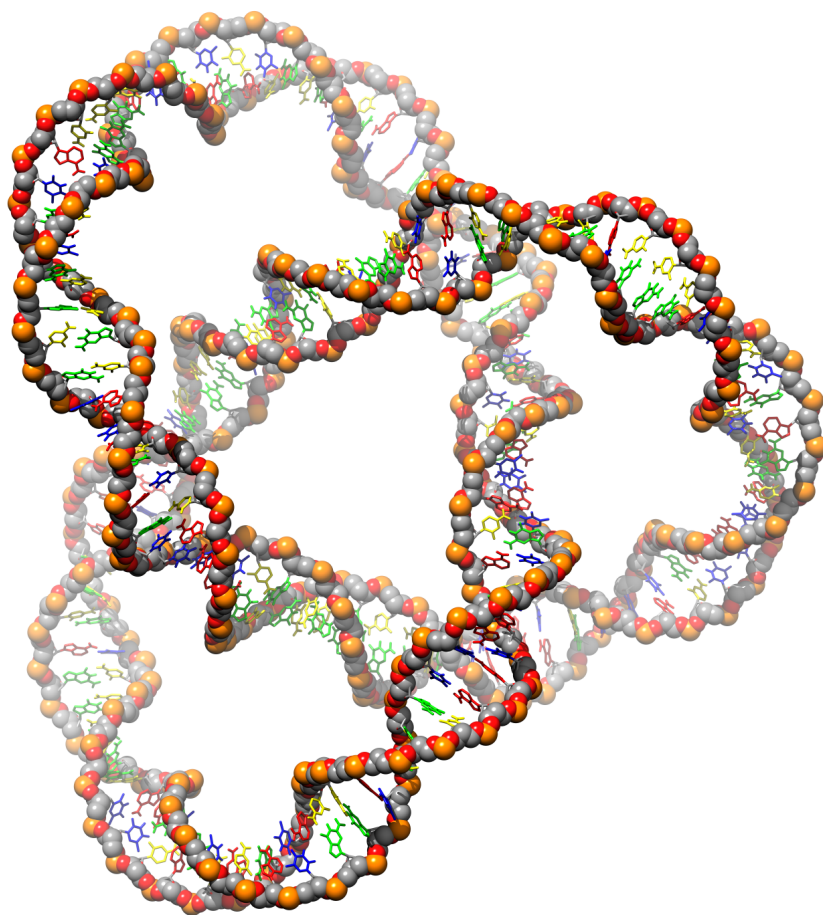


AmberTools12

Reference Manual



AmberTools12 Reference Manual

AmberTools consists of several independently developed packages that work well with Amber itself. The main components of AmberTools are listed below.

NAB (Nucleic Acid Builder)

Thomas J. Macke, W.A. Svrcek-Seiler,
Russell A. Brown, István Kolossváry,
Yannick J. Bomble, Ramu Anandakrishnan,
David A. Case

LEaP

Wei Zhang, Tingjun Hou, Christian
Schafmeister, Wilson S. Ross, David A. Case

antechamber

Junmei Wang

amberlite

Romain M. Wolf

ptraj

Thomas E. Cheatham, III, *et al.* (see
<http://ambermd.org/contributors.html>)

cpptraj

Daniel R. Roe, *et al.* (see
<http://ambermd.org/contributors.html>)

pbsa

Jun Wang, Qin Cai, Xiang Ye, Meng-Juei
Hsieh, Chuck Tan, Ray Luo

sqm

Ross C. Walker, Michael F. Crowley, Scott
Brozell, Tim Giese, Andreas W. Götz,
Tai-Sung Lee, David A. Case

CHAMBER

Michael F. Crowley, Mark Williamson, Ross
C. Walker

3D-RISM

Tyler Luchko⁵, David A. Case, Sergey
Gusarov, Andriy Kovalenko

mdgx

David S. Cerutti

MMPBSA.py

Jason Swails, T. Dwight McGee Jr., Bill
Miller III

MTK++, MCPB

Martin Peters, Kenneth Ayers, Andrew
Wollacott, Duane E. Williams, Benjamin P.
Roberts, Kenneth M. Merz, Jr.

paramfit

Ross C. Walker, Robin Betz

Notes

- Most of the programs included here can be redistributed and/or modified under the terms of the GNU General Public License; a few components have other open-source licenses. See the *amber11/AmberTools/LICENSE* file for details. The programs are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
- Some of the force field routines were adapted from similar routines in the MOIL program package: R. Elber, A. Roitberg, C. Simmerling, R. Goldstein, H. Li, G. Verkhivker, C. Keasar, J. Zhang and A. Ulitsky, "MOIL: A program for simulations of macromolecules" *Comp. Phys. Commun.* **91**, 159-189 (1995).
- The "trifix" routine for random pairwise metrization is based on an algorithm designed by Jay Ponder and was adapted from code in the Tinker package; see M.E. Hodsdon, J.W. Ponder, and D.P. Cistola, *J. Mol. Biol.* **264**, 585-602 (1996) and <http://dasher.wustl.edu/tinker/>.
- The "molsurf" routines for computing molecular surface areas were adapted from routines written by Paul Beroza. The "sasad" routine for computing derivatives of solvent accessible surface areas was kindly provided by S. Sridharan, A. Nicholls and K.A. Sharp. See *J. Computat. Chem.* **8**, 1038-1044 (1995).
- Some of the "pb_exmol" routines for mapping molecular surface to finite-difference grids were adapted from routines written by Michael Gilson and Malcolm Davis in UHBD. See *Comp. Phys. Comm.* **91**, 57-95 (1995).
- The *cifparse* routines to deal with mmCIF formatted files were written by John Westbrook, and are distributed with permission. See *cifparse/README* for details.
- Sun, Sun Microsystems and Sun Performance Library are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Cover Illustration

The cover symbolizes the entangled, endless (yet finite) and somewhat imperfect nature of the code at hand. This is a (2, 3) torus knot (http://en.wikipedia.org/wiki/Torus_knot) generated from a slightly modified program #9 from NAB, using a parametric representation of the knot, and rendered using *chimera* [1] and *povray* (<http://www.povray.org>) by Volodymyr Babin.

Contents

Contents	3
1. Getting started	17
1.1. Information flow in Amber	17
1.1.1. Preparatory programs	18
1.1.2. Simulation programs	19
1.1.3. Analysis programs	19
1.2. Installation	20
1.3. Combining AmberTools12 with Amber11 or Amber10	23
1.4. Testing the installation	24
1.5. Applying Updates	24
1.5.1. Advanced options	25
1.6. Contacting the developers	25
1.7. List of programs	26
2. Specifying a force field	31
2.1. Specifying which force field you want in LEaP	32
2.2. The ff12SB force field	33
2.2.1. Backbone modifications	34
2.2.2. Side chain modifications	34
2.3. The ff10 force field	34
2.3.1. Variants of ff10	35
2.4. The AMOEBA potentials	36
2.5. The Duan et al. (2003) force field	36
2.6. The Yang et al. (2003) united-atom force field	37
2.7. The 2002 polarizable force fields	37
2.8. Force fields related to semi-empirical QM	38
2.9. The GLYCAM-06 and GLYCAM-06EP force fields for carbohydrates and lipids	38
2.9.1. File versioning	39
2.9.2. Atom type name changes in the current versions	40
2.9.3. General information regarding parameter development	41
2.9.4. Scaling of electrostatic and nonbonded interactions	41
2.9.5. Development of partial atomic charges	41
2.9.6. Carbohydrate parameters for use with the TIP5P water model	42
2.9.7. Carbohydrate Naming Convention in GLYCAM	42
2.10. LIPID11: A modular lipid force field	47
2.10.1. LIPID11 PDB format	48

CONTENTS

2.11. Ions	48
2.12. Solvent models	49
2.13. CHAMBER	50
2.13.1. Usage	54
2.13.2. Validation	55
2.13.3. Known limitations / Issues	56
3. LEaP	57
3.1. Introduction	57
3.2. Concepts	57
3.2.1. Commands	57
3.2.2. Variables	58
3.2.3. Objects	59
3.3. Basic instructions for using LEaP	63
3.3.1. Building a Molecule For Molecular Mechanics	63
3.3.2. Amino Acid Residues	64
3.3.3. Nucleic Acid Residues	64
3.4. Commands	65
3.4.1. add	65
3.4.2. addAtomTypes	66
3.4.3. addIons	66
3.4.4. addIons2	66
3.4.5. addPath	66
3.4.6. addPdbAtomMap	67
3.4.7. addPdbResMap	67
3.4.8. alias	68
3.4.9. bond	68
3.4.10. bondByDistance	68
3.4.11. check	69
3.4.12. combine	69
3.4.13. copy	70
3.4.14. createAtom	70
3.4.15. createResidue	70
3.4.16. createUnit	70
3.4.17. deleteBond	70
3.4.18. desc	71
3.4.19. groupSelectedAtoms	72
3.4.20. help	72
3.4.21. impose	72
3.4.22. list	74
3.4.23. loadAmberParams	75
3.4.24. loadAmberPrep	75
3.4.25. loadOff	75
3.4.26. loadMol2	75
3.4.27. loadPdb	76

3.4.28. loadPdbUsingSeq	76
3.4.29. logFile	76
3.4.30. measureGeom	77
3.4.31. quit	77
3.4.32. remove	77
3.4.33. saveAmberParm	77
3.4.34. saveMol2	78
3.4.35. saveOff	78
3.4.36. savePdb	78
3.4.37. sequence	78
3.4.38. set	79
3.4.39. solvateBox and solvateOct	81
3.4.40. solvateCap	81
3.4.41. solvateShell	82
3.4.42. source	82
3.4.43. transform	82
3.4.44. translate	83
3.4.45. verbosity	83
3.4.46. zMatrix	83
3.5. Building oligosaccharides and lipids	84
3.5.1. Procedures for building oligosaccharides using the GLYCAM-06 pa- rameters	85
3.5.2. Procedures for building a lipid using GLYCAM-06 parameters	89
3.5.3. Procedures for building a glycoprotein in LEaP.	90
4. Antechamber and MCPB	93
4.1. Principal programs	94
4.1.1. antechamber	94
4.1.2. parmchk	97
4.2. A simple example for antechamber	98
4.3. Programs called by antechamber	102
4.3.1. atomtype	102
4.3.2. am1bcc	102
4.3.3. bondtype	103
4.3.4. prepgen	104
4.3.5. espgen	105
4.3.6. respgen	105
4.4. Miscellaneous programs	107
4.4.1. acdoctor	107
4.4.2. parmcal	107
4.4.3. residuegen	107
4.5. New Development of Antechamber And GAFF	108
4.5.1. Extensive Test of AM1-BCC Charges	109
4.5.2. New Development of GAFF	109

CONTENTS

4.6. Metal Center Parameter Builder (MCPB)	110
4.6.1. Introduction	110
4.6.2. Running MCPB	110
5. amberlite: Some AMBER-Tools-Based Utilities	113
5.1. Introduction	113
5.1.1. Installation	114
5.1.2. Python Scripts	114
5.1.3. NAB Applications	115
5.2. Coordinates and Parameter-Topology Files	116
5.3. <i>pytleap</i> : Creating Coordinates and Parameter- Topology Files	117
5.3.1. Running <i>pytleap</i>	117
5.3.2. Output from <i>pytleap</i>	120
5.3.3. Error Checking	121
5.4. Energy Checking Tool: <i>ffgbsa</i>	121
5.5. Energy Minimizer: <i>minab</i>	121
5.6. Molecular Dynamics "Lite": <i>mdnab</i>	123
5.7. MM(GB)(PB)/SA Analysis Tool: <i>pymdpbsa</i>	124
5.7.1. Brief Overview on MM(GB)(PB)/SA Concepts	124
5.7.2. Pitfalls and Error Sources	125
5.7.3. Some Technical Remarks on <i>pymdpbsa</i>	126
5.7.4. Running <i>pymdpbsa</i>	126
5.7.5. Details on Internal Workings and Output of <i>pymdpbsa</i>	128
5.7.6. Using <i>pymdpbsa</i> for Single-Point Interaction Energy	131
5.7.7. Remark Concerning Poisson-Boltzmann Options <code>--solv 3</code> and <code>--solv</code> <code>4</code>	131
5.8. Appendix A: Preparing PDB Files	132
5.8.1. Cleaning up Protein PDB Files for AMBER	132
5.8.2. Special Residues, Name Conventions, Chain Terminations	133
5.8.3. Chains, Residue Numbering, Missing Residues	135
5.9. Appendix B: Atom and Residue Selections	135
5.9.1. Amber Masks	136
5.9.2. "Atom Expressions" in NAB Applications	137
5.10. Appendix C: Examples and Test Cases	138
5.10.1. Example 1: Generating AMBER Files for Crambin with Disulfide Bonds	138
5.10.2. Example 2: Energy Minimization of the Crambin Structure	139
5.10.3. Example 3: Preparation of a Complex between P38 MAP Kinase and Ligand	141
5.10.4. Example 4: Interaction Energy between P38 and Ligand in the Unre- fined (Original) Complex	143
5.10.5. Example 5: Minimization of P38 Complex with <i>minab</i> and Resulting Interaction Energy	144
5.10.6. Example 6: Generate MD Trajectory for the P38-Ligand Complex with <i>mdnab</i>	146

5.10.7. Example 7: Running <i>pymdpbsa</i> on the P38/Ligand Complex Trajectory	147
6. sqm: Semi-empirical quantum chemistry	149
6.1. Available Hamiltonians	149
6.2. Charge-dependent exchange-dispersion corrections of vdW interactions	151
6.3. Dispersion and hydrogen bond correction	151
6.4. Usage	153
7. cpptraj	161
7.1. Comparison to ptraj - Important Differences	161
7.1.1. Actions and multiple topologies	162
7.1.2. Datasets and datafiles	163
7.1.3. Enhanced features for actions using the “out” keyword	163
7.2. Running cpptraj	163
7.2.1. Command Line Syntax	163
7.2.2. Mask Syntax	164
7.2.3. Ranges	165
7.2.4. OpenMP Parallelization	165
7.3. Ptraj actions/analysis in Cpptraj	166
7.4. General Commands	167
7.4.1. activeref	167
7.4.2. debug	167
7.4.3. noexitonerror	168
7.4.4. noprogess	168
7.4.5. select	168
7.5. Parameter and Trajectory File Commands	168
7.5.1. parm	169
7.5.2. parminfo	169
7.5.3. parmwrite	170
7.5.4. parmstrip	170
7.5.5. box parmbox	170
7.5.6. parmbondinfo	170
7.5.7. parmmolinfo	171
7.5.8. parmresinfo	171
7.5.9. bondsearch nobondsearch	171
7.5.10. molsearch nomolsearch	171
7.5.11. trajin	171
7.5.12. trajout	173
7.5.13. reference	175
7.6. Datafile Commands (actions using the “out” keyword only)	175
7.6.1. datafile create	175
7.6.2. datafile xlabel	176
7.6.3. datafile ylabel	176
7.6.4. datafile invert	176
7.6.5. datafile noxcol	176

CONTENTS

7.6.6.	datafile noheader	176
7.6.7.	datafile precision	177
7.7.	Commands that modify the state	177
7.7.1.	atommap	177
7.7.2.	center	178
7.7.3.	closest	178
7.7.4.	image	179
7.7.5.	runavg	180
7.7.6.	strip	180
7.7.7.	unstrip	180
7.7.8.	unwrap	181
7.8.	Action Commands	181
7.8.1.	angle	181
7.8.2.	atomicfluct	181
7.8.3.	average	182
7.8.4.	avgcoord	182
7.8.5.	check	183
7.8.6.	clusterdihedral	183
7.8.7.	cluster	184
7.8.8.	contacts	186
7.8.9.	diffusion	186
7.8.10.	dihedral	186
7.8.11.	dipole	186
7.8.12.	distance	186
7.8.13.	drmsd (distance RMSD)	186
7.8.14.	dnaiontracker	187
7.8.15.	grid	187
7.8.16.	gfe (GridFreeEnergy)	187
7.8.17.	hbond	187
7.8.18.	jcoupling	188
7.8.19.	mask	189
7.8.20.	molsurf	190
7.8.21.	nastruct	190
7.8.22.	outtraj	191
7.8.23.	principal	191
7.8.24.	projection	191
7.8.25.	pucker	191
7.8.26.	radgyr	192
7.8.27.	radial	192
7.8.28.	randomizeions	193
7.8.29.	rmsavgcorr	193
7.8.30.	rmsd	193
7.8.31.	rms2d	195
7.8.32.	rotdif	195
7.8.33.	scale	197

7.8.34. secstruct	197
7.8.35. surf	198
7.8.36. watershell	198
7.9. Matrix and Vector Actions	198
7.9.1. matrix	198
7.9.2. vector	198
7.10. Analysis Commands	198
7.10.1. corr / analyze correlationcoe	199
7.10.2. analyze crank	199
7.10.3. hist(ogram)	199
7.10.4. analyze matrix	200
7.10.5. analyze modes	200
7.10.6. analyze stat	200
7.10.7. analyze timecorr	200
8. PBSA	201
8.1. Introduction	201
8.1.1. Numerical solutions of the PB equation	202
8.1.2. Numerical interpretation of energy and forces	203
8.1.3. Numerical accuracy and related issues	204
8.2. Usage and keywords	205
8.2.1. File usage	205
8.2.2. Basic input options	205
8.2.3. Options to define the physical constants	207
8.2.4. Options for Implicit Membranes	209
8.2.5. Options to select numerical procedures	209
8.2.6. Options to compute energy and forces	211
8.2.7. Options for visualization and output	212
8.2.8. Options to select a non-polar solvation treatment	213
8.2.9. Options to enable active site focusing	214
8.2.10. Options to enable multiblock focusing	215
8.3. Example inputs and demonstrations of functionalities	216
8.3.1. Single-point calculation of solvation free energies	216
8.3.2. Implicit membrane model	217
8.3.3. Single point calculation of forces	217
8.3.4. Comparing with <i>Delphi</i> results	218
8.4. Visualization functions in <i>pbsa</i>	219
8.4.1. Visulization of electrostatic potential using <i>PyMol</i>	219
8.4.2. Writting electrostatic potential to DX format volumetric data file	220
8.4.3. Loading DX format electrostatic potential data in <i>VMD</i>	220
8.4.4. Changing the representation model	221
8.4.5. Adjusting the color scale of the color map	222
8.4.6. Changing the color scale range	222
8.4.7. Loading electrostatic potential data into an existing molecule	223
8.4.8. Using the electrostatic potential data as a color map	223

CONTENTS

8.4.9.	Loading and displaying the level set map	224
8.4.10.	Visualizing the molecular surface as an isosurface of the level set	225
8.4.11.	Visualizing interior channels, voids, and solvent pockets	226
8.4.12.	Importing / Modifying Atomic Radii to VMD from the prmtop file	227
8.5.	<i>pbsa</i> in <i>sander</i> and NAB	228
8.5.1.	Electrostatic forces/gradients in <i>pbsa</i>	228
8.5.2.	Example for <i>pbsa</i> in <i>sander</i>	229
8.5.3.	Example for <i>pbsa</i> in NAB	229
9.	Reference Interaction Site Model	231
9.1.	Introduction	231
9.1.1.	1D-RISM	233
9.1.2.	3D-RISM	234
9.1.3.	Analytic Temperature Derivatives	236
9.2.	Practical Considerations	237
9.2.1.	Computational Requirements and Parallel Scaling	237
9.2.2.	Output	238
9.2.3.	Numerical Accuracy	238
9.3.	Work Flow	239
9.4.	<i>rism1d</i>	239
9.4.1.	Parameters	239
9.4.2.	Example	243
9.5.	3D-RISM in NAB	243
9.5.1.	Solvation Box Size	243
9.5.2.	I/O	244
9.5.3.	Examples	244
9.5.4.	Thermodynamic Output	245
9.5.5.	Compiling MPI 3D-RISM	246
9.6.	<i>rism3d.snglpnt</i>	246
9.6.1.	Usage	247
9.7.	File Formats	249
9.7.1.	MDL	249
9.7.2.	XVV	250
9.7.3.	Site-site functionals	252
9.7.4.	Thermodynamics	253
9.7.5.	Total excess values	254
9.7.6.	DX volumetric data	254
9.7.7.	XYZV volumetric data	255
10.	MMPBSA.py	257
10.1.	Introduction	257
10.2.	Preparing for an MM/PB(GB)SA calculation	258
10.2.1.	Building Topology Files	258
10.2.2.	Using ante-MMPBSA.py	259
10.2.3.	Running Molecular Dynamics	260

10.3. Running MMPBSA.py	260
10.3.1. The input file	260
10.3.2. Calling MMPBSA.py from the command-line	268
10.3.3. Running MMPBSA.py	270
10.3.4. Types of calculations you can do	271
10.3.5. The Output File	272
10.3.6. Temporary Files	272
10.3.7. Advanced Options	275
11. mdgx: A Developmental Molecular Simulation Engine in Simple C Code	277
11.1. Input and Output	277
11.2. Installation	279
11.3. Special Algorithmic Features of mdgx	279
11.4. Customizable Virtual Site Support in mdgx	280
11.5. Restrained Electrostatic Potential Fitting in mdgx	283
11.6. Thermodynamic Integration	287
11.7. Future Directions and Goals of the mdgx Project	288
12. paramfit	291
12.1. Usage	291
12.2. The Job Control File	292
12.2.1. General options	292
12.2.2. Creating quantum input files	293
12.2.3. Specifying parameters	294
12.2.4. Fitting options	294
12.2.5. Algorithm options	295
12.2.6. Bounds Checking	297
12.3. Examples	298
12.3.1. Setting up to fit	298
12.3.2. Fitting K	298
12.3.3. Improving a fit iteratively	299
12.3.4. Evaluating Results	299
13. Miscellaneous utilities	301
13.1. ambpdb	301
13.2. reduce	303
13.2.1. Running reduce	303
13.2.2. General input flags	304
13.2.3. Fixing an orientation	305
13.2.4. Cliques	306
13.2.5. Contact	306
13.3. elsize	306
13.4. Utilities for Molecular Crystal Simulations	307
13.4.1. UnitCell	308
13.4.2. PropPDB	308

CONTENTS

13.4.3. AddToBox	308
13.4.4. ChBox	310
13.5. ParmEd	310
13.5.1. Running parmed.py	310
13.5.2. ParmEd commands (they are all case-insensitive)	311
13.5.3. Examples	318
13.5.4. xparmed.py	320
13.5.5. Advanced Options	321
14. NAB: Introduction	327
14.1. Background	328
14.1.1. Conformation build-up procedures	329
14.1.2. Base-first strategies	329
14.2. Methods for structure creation	330
14.2.1. Rigid-body transformations	330
14.2.2. Distance geometry	331
14.2.3. Molecular mechanics	332
14.3. Compiling nab Programs	333
14.4. Parallel Execution	333
14.5. First Examples	334
14.5.1. B-form DNA duplex	334
14.5.2. Superimpose two molecules	335
14.5.3. Place residues in a standard orientation	336
14.6. Molecules, Residues and Atoms	337
14.7. Creating Molecules	338
14.8. Residues and Residue Libraries	339
14.9. Atom Names and Atom Expressions	341
14.10. Looping over atoms in molecules	343
14.11. Points, Transformations and Frames	344
14.11.1. Points and Vectors	344
14.11.2. Matrices and Transformations	344
14.11.3. Frames	345
14.12. Creating Watson Crick duplexes	346
14.12.1. bdna() and fd_helix()	347
14.12.2. wc_complement()	347
14.12.3. wc_helix() Overview	349
14.12.4. wc_basepair()	349
14.12.5. wc_helix() Implementation	352
14.13. Structure Quality and Energetics	356
14.13.1. Creating a Parallel DNA Triplex	356
14.13.2. Creating Base Triads	357
14.13.3. Finding the lowest energy triad	359
14.13.4. Assembling the Triads into Dimers	361

15. NAB: Language Reference	367
15.1. Introduction	367
15.2. Language Elements	367
15.2.1. Identifiers	367
15.2.2. Reserved Words	367
15.2.3. Literals	368
15.2.4. Operators	368
15.2.5. Special Characters	369
15.3. Higher-level constructs	369
15.3.1. Variables	369
15.3.2. Attributes	370
15.3.3. Arrays	372
15.3.4. Expressions	373
15.3.5. Regular expressions	374
15.3.6. Atom Expressions	374
15.3.7. Format Expressions	375
15.4. Statements	377
15.4.1. Expression Statement	377
15.4.2. Delete Statement	378
15.4.3. If Statement	378
15.4.4. While Statement	378
15.4.5. For Statement	379
15.4.6. Break Statement	380
15.4.7. Continue Statement	380
15.4.8. Return Statement	380
15.4.9. Compound Statement	380
15.5. Structures	380
15.6. Functions	382
15.6.1. Function Definitions	382
15.6.2. Function Declarations	383
15.7. Points and Vectors	383
15.8. String Functions	384
15.9. Math Functions	385
15.10 System Functions	385
15.11 I/O Functions	385
15.11.1. Ordinary I/O Functions	385
15.11.2. matrix I/O	388
15.12 Molecule Creation Functions	389
15.13 Creating Biopolymers	390
15.14 Fiber Diffraction Duplexes in NAB	391
15.15 Reduced Representation DNA Modeling Functions	392
15.16 Molecule I/O Functions	392
15.17 Other Molecular Functions	394
15.18 Debugging Functions	395
15.19 Time and date routines	396

CONTENTS

15.20	Computational resource consumption functions	396
16.	NAB: Rigid-Body Transformations	397
16.1.	Transformation Matrix Functions	397
16.2.	Frame Functions	397
16.3.	Functions for working with Atomic Coordinates	398
16.4.	Symmetry Functions	398
16.4.1.	Matrix Creation Functions	399
16.4.2.	Matrix I/O Functions	400
16.5.	Symmetry server programs	401
16.5.1.	matgen	401
16.5.2.	Symmetry Definition Files	401
16.5.3.	matmerge	403
16.5.4.	matmul	404
16.5.5.	matextract	404
16.5.6.	transform	404
17.	NAB: Distance Geometry	405
17.1.	Metric Matrix Distance Geometry	405
17.2.	Creating and manipulating bounds, embedding structures	406
17.3.	Distance geometry templates	411
17.4.	Bounds databases	414
18.	NAB: Molecular mechanics and dynamics	417
18.1.	Basic molecular mechanics routines	417
18.2.	Typical calling sequences	428
18.3.	Second derivatives and normal modes	429
18.4.	Low-MODE (LMOD) optimization methods	431
18.4.1.	LMOD conformational searching	431
18.4.2.	LMOD Procedure	432
18.4.3.	XMIN	433
18.4.4.	Sample XMIN program	435
18.4.5.	LMOD	438
18.4.6.	Sample LMOD program	441
18.4.7.	Tricks of the trade of running LMOD searches	445
18.5.	Using the Hierarchical Charge Partitioning (HCP) method	446
18.5.1.	Level 1 HCP approximation	446
18.5.2.	Level 2 and 3 HCP approximation	447
19.	NAB: Sample programs	449
19.1.	Duplex Creation Functions	449
19.2.	nab and Distance Geometry	450
19.2.1.	Refine DNA Backbone Geometry	452
19.2.2.	RNA Pseudoknots	454
19.2.3.	NMR refinement for a protein	456

19.3. Building Larger Structures	460
19.3.1. Closed Circular DNA	461
19.3.2. Nucleosome Model	465
19.4. Wrapping DNA Around a Path	468
19.4.1. Interpolating the Curve	468
19.4.2. Driver Code	472
19.4.3. Wrap DNA	473
19.5. Other examples	476
Bibliography	477
Bibliography	477
Index	496
A. ptraj	503
A.1. Understanding ptraj commands and actions	504
A.2. ptraj coordinate input/output commands	505
A.3. ptraj commands that override the molecular information specified	507
A.4. ptraj <i>action</i> commands - short list	508
A.5. ptraj <i>action</i> commands - detailed discussion	508
A.6. Correlation and fluctuation facility	523
A.7. Examples	527
A.7.1. Calculating and analyzing matrices and modes	527
A.7.2. Projecting snapshots onto modes	527
A.7.3. Calculating time correlation functions	528
A.8. Hydrogen bonding facility	528
A.9. rdparm	530
B. Obsolete force field files	533
B.1. The Cornell et al. (1994) force field	533
B.2. The Weiner et al. (1984,1986) force fields	534
B.3. The Wang et al. (1999) force field	534

1. Getting started

AmberTools is a set of programs for biomolecular simulation and analysis. They are designed to work well with each other, and with the “regular” Amber suite of programs. You can perform many simulation tasks with AmberTools, and you can do more extensive simulations with the combination of AmberTools and Amber itself.

Most components of AmberTools are released under the GNU General Public License (GPL). A few components are in the public domain or have other open-source licenses. See the *README_at* and *LICENSE_at* files for more information. We hope to add new functionality to AmberTools as additional programs become available. If you have suggestions for what might be added, please contact us.

Everyone should read (or at least skim) Section 1.1. Even if you are an experienced Amber user, there may be things you have missed, or new features, that will help.

If you are installing this package see Section 1.2. There are also tips and examples on the Amber Web pages at <http://ambermd.org>. Although Amber may appear dauntingly complex at first, it has become easier to use over the past few years, and overall is reasonably straightforward once you understand the basic architecture and option choices. In particular, we have worked hard on the tutorials to make them accessible to new users. Thousands of people have learned to use Amber; don’t be easily discouraged.

If you want to learn more about basic biochemical simulation techniques, there are a variety of good books to consult, ranging from introductory descriptions,[2, 3] to standard works on liquid state simulation methods,[4, 5] to multi-author compilations that cover many important aspects of biomolecular modelling.[6–8] Looking for “paradigm” papers that report simulations similar to ones you may want to undertake is also generally a good idea.

1.1. Information flow in Amber

Understanding where to begin in AmberTools is primarily a problem of managing the flow of information in this package — see Fig. 1.1. You first need to understand what information is needed by the simulation programs (*sander*, *pmemd*, *mdgx* or *nab*). You need to know where it comes from, and how it gets into the form that the energy programs require. This section is meant to orient the new user and is not a substitute for the individual program documentation.

Information that all the simulation programs need:

1. Cartesian coordinates for each atom in the system. These usually come from X-ray crystallography, NMR spectroscopy, or model-building. They should be in Protein Data Bank (PDB) or Tripos “mol2” format. The program *LEaP* provides a platform for carrying out many of these modeling tasks, but users may wish to consider other programs as well.

1. Getting started

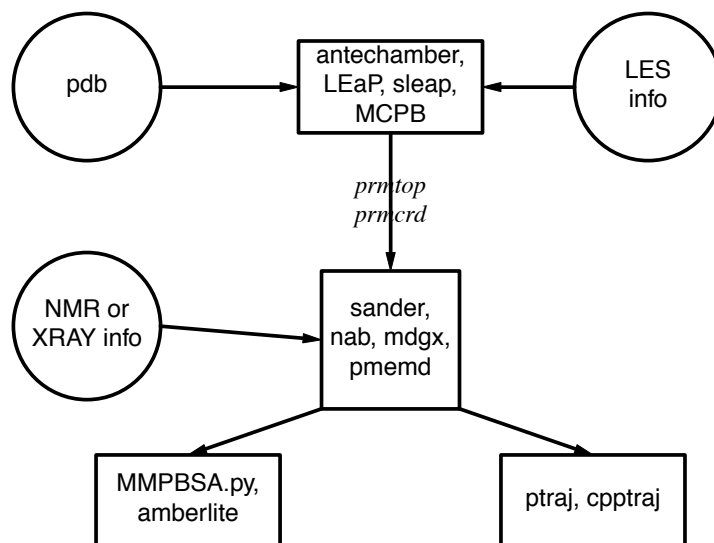


Figure 1.1.: Basic information flow in Amber

2. Topology: Connectivity, atom names, atom types, residue names, and charges. This information comes from the database, which is found in the `$AMBERHOME/dat/leap/leap` directory, and is described in Chapter 2. It contains topology for the standard amino acids as well as N- and C-terminal charged amino acids, DNA, RNA, and common sugars. The database contains default internal coordinates for these monomer units, but coordinate information is usually obtained from PDB files. Topology information for other molecules (not found in the standard database) is kept in user-generated “residue files”, which are generally created using *antechamber*.
3. Force field: Parameters for all of the bonds, angles, dihedrals, and atom types in the system. The standard parameters for several force fields are found in the `$AMBERHOME/dat/leap/parm` directory; see Chapter 2 for more information. These files may be used “as is” for proteins and nucleic acids, or users may prepare their own files that contain modifications to the standard force fields.
4. Commands: The user specifies the procedural options and state parameters desired. These are specified in “driver” programs written in the NAB language.

1.1.1. Preparatory programs

LEaP is the primary program to create a new system in Amber, or to modify existing systems. It is available as the command-line program *tleap* or the GUI *xleap*. It combines the functionality of *prep*, *link*, *edit* and *parm* from earlier versions.

antechamber is the main program from the Antechamber suite. If your system contains more than just standard nucleic acids or proteins, this may help you prepare the input for LEaP.

MCPB provides a means to rapidly build, prototype, and validate MM models of metalloproteins. It uses the bonded plus electrostatics model to expand existing pairwise additive force fields.

paramfit iteratively fits individual force field parameters for a given system. This can be useful when *antechamber* fails to find parameters, or if existing force fields fail to properly characterize the system.

1.1.2. Simulation programs

NAB (Nucleic Acid Builder) is a language that can be used to write programs to perform non-periodic simulations, most often using an implicit solvent force field.

sander (part of Amber) is the basic energy minimizer and molecular dynamics program. This program relaxes the structure by iteratively moving the atoms down the energy gradient until a sufficiently low average gradient is obtained. The molecular dynamics portion generates configurations of the system by integrating Newtonian equations of motion. MD will sample more configurational space than minimization, and will allow the structure to cross over small potential energy barriers. Configurations may be saved at regular intervals during the simulation for later analysis, and basic free energy calculations using thermodynamic integration may be performed. More elaborate conformational searching and modeling MD studies can also be carried out using the SANDER module. This allows a variety of constraints to be added to the basic force field, and has been designed especially for the types of calculations involved in NMR structure refinement.

pmemd (part of Amber) is a version of *sander* that is optimized for speed and for parallel scaling. The name stands for “Particle Mesh Ewald Molecular Dynamics,” but this code can now also carry out generalized Born simulations. The input and output have only a few changes from *sander*.

mdgx is a molecular dynamics engine with functionality that mimics some of the features in *sander* and *pmemd*, but featuring simple C code and an atom sorting routine that simplifies the flow of information during force calculations. The principal purpose of *mdgx* is to provide a tool for radical redesign of the basic molecular dynamics algorithms and models.

1.1.3. Analysis programs

ptraj is a general purpose utility for analyzing and processing trajectory or coordinate files created from MD simulations (or from various other sources), carrying out superpositions, extractions of coordinates, calculation of bond/angle/dihedral values, atomic positional fluctuations, correlation functions, analysis of hydrogen bonds, etc. The same executable, when named *rdparm* (from which *ptraj* evolved), can examine and modify prmtop files.

1. Getting started

cpptraj is a trajectory analysis utility (written in C++) that is similar to *ptraj*. It has almost all the functionality of *ptraj* and generally better performance, particularly when processing NetCDF trajectories. A few key features of *cpptraj* are the ability to process multiple prmtop files at once, ability to specify a separate reference mask during RMSD calculations, support for multiple output trajectory files, native support for compressed (gzip or bzip2) trajectories and prmtop files, and output of stripped prmtop files.

pbsa is an analysis program for solvent-mediated energetics of biomolecules. It can be used to perform both electrostatic and non-electrostatic continuum solvation calculations with input coordinate files from molecular dynamics simulations and other sources. The electrostatic solvation is modeled by the Poisson-Boltzmann equation. Both linear and full nonlinear numerical solvers are implemented. The nonelectrostatic solvation is modeled by two separate terms: dispersion and cavity.

MMPBSA.py is a python script that automates energy analysis of snapshots from a molecular dynamics simulation using ideas generated from continuum solvent models. (There is also an older perl script, called *mm_pbsa*, that is a part of Amber.)

amberlite is small set of NAB programs and python scripts that implement a limited set of MD simulations and mm-pbsa (or mm-gbsa) analysis, aimed primarily at the analysis of protein-ligand interactions. These tools can be useful in their own right, or as a good introduction to Amber and a starting point for more complex calculations. Detailed instructions are in Chapter 5.

1.2. Installation

We have worked hard in this release to simplify the installation of Amber, and there are some differences from earlier releases. First, if you have both *Amber* and *AmberTools*, both will be installed and tested with a single command. Second, the configure script automatically checks for bugfixes, and installs them if you ask it to. A third (minor) change is that the configure script is now run from the \$AMBERHOME directory (not from \$AMBERHOME/AmberTools/src, as in the past.)

The Amber web page (<http://ambermd.org>) has some specific instructions and hints for various common operating systems. Look for the “Running Amber on” links. Once you have downloaded the distribution files, do the following:

1. First, extract the files in some location (we use */home/myname* as an example here):

```
cd /home/myname
tar xvfj AmberTools12.tar.bz2
tar xvfj Amber12.tar.bz2      # (only if you have licensed Amber 12!)
```

2. Next, set your AMBERHOME environment variable:

```
export AMBERHOME=/home/myname/amber12      # (for bash, zsh, ksh, etc.)
setenv AMBERHOME /home/myname/amber12      # (for csh, tcsh)
```

Be sure to change the “/home/myname” above to whatever directory is appropriate for your machine, and be sure that you have write permissions in the directory tree you choose. You should also add \$AMBERHOME/bin to your PATH.

3. Next, you may need to install some compilers and other libraries. Details depend on what OS you have, and what is already installed. Package managers can greatly simplify this task. For example for Debian-based Linux systems (such as Ubuntu), the following command should get you what you need:

```
sudo apt-get install csh flex gfortran g++ xorg-dev \
    zlib1g-dev libbz2-dev
```

Other Linux distributions will have a similar command, but with a package manager different than `apt-get`. For example, the following should work for Fedora Core and similar systems:

```
sudo yum install gcc flex tcsh zlib-devel bzip2-devel \
    libXt-devel libXext-devel libXdmcp-devel
```

For Macintosh OSX, *MacPorts* (<http://www.macports.org>) serves a similar purpose. You would download and install the *port* program, then issue commands like this:

```
sudo port install gcc46
```

MacPorts is useful because the “Xcode” compilers provided by Apple will not work to compile Amber, since no Fortran compiler is provided. Amber cross-links Fortran and C/C++ code, so a “full” GCC installation is necessary.

4. Now, in the *AMBERHOME* directory, run the configure script:

```
cd $AMBERHOME
./configure --help
```

will show you the options. Choose the compiler and flags you want; for most systems, the following should work:

```
./configure gnu
```

Don’t choose any parallel options at this point. (You may need to edit the resulting *config.h* file to change any variables that don’t match your compilers and OS. The comments in the *config.h* file should help.) This step will also check to see if there are any bugfixes that have not been applied to your installation, and will apply them (unless you ask it not to). If the configure step finds missing libraries, go back to Step 3.

5. Then,

```
make install
```

will compile the codes. If this step fails, try to read the error messages carefully to identify the problem.

6. This can be followed by

```
make test
```

1. Getting started

which will run tests and will report successes or failures.

Where "possible FAILURE" messages are found, go to the indicated directory under \$AMBERHOME/AmberTools/test or \$AMBERHOME/test, and look at the "*.dif" files. Differences should involve round-off in the final digit printed, or occasional messages that differ from machine to machine (see below for details). As with compilation, if you have trouble with individual tests, you may wish to comment out certain lines in the Makefiles (i.e., \$AMBERHOME/AmberTools/test/Makefile or \$AMBERHOME/test/Makefile), and/or go directly to the test subdirectories to examine the inputs and outputs in detail. For convenience, all of the failure messages and differences are collected in the \$AMBERHOME-/logs directory; you can quickly see from these if there is anything more than round-off errors.

Note: If you have untarred the `Amber12.tar.bz2` file, then steps 1-6 will install and test both *AmberTools* and *Amber*; otherwise it will just install and test *AmberTools*. If you license *Amber* later, just come back and repeat steps 1-6 again.

7. If you are new to Amber, you should look at the tutorials and this manual and become familiar with how things work. If and when you wish to compile parallel (MPI) versions of *Amber*, do this:

```
cd $AMBERHOME
./configure -mpi <...other options...> <compiler-choice>
make install
# Note the value below may depend on your MPI implementation
export DO_PARALLEL="mpirun -np 2"
make test
# Note, some tests, like the replica exchange tests, require more
# than 2 threads, so we suggest that you test with either 4 or 8
# threads as well
export DO_PARALLEL="mpirun -np 8"
make test
```

This assumes that you have installed MPI, that you have set your `MPI_HOME` environment variable to the MPI installation path, and that *mpicc* and *mpif90* are in your `PATH`. Some MPI installations are tuned to particular hardware (such as infiniband), and you should use those versions if you have such hardware. Most people can use standard versions of either *mpich2* or *openmpi*. To install one of these, use one of the simple scripts that we have prepared:

```
cd $AMBERHOME/AmberTools/src
./configure_mpich2 <compiler-choice>    OR
./configure_openmpi <compiler-choice>
```

Follow the instructions of these scripts, then return to beginning of step 7.

Note: Parallel versions of *AmberTools* are rather specialized, and many users will skip this step. Consider the following points before compiling and using the MPI version:

- a) The MPI version of *nab* is called *mpinab*, by analogy with *mpicc* or *mpif90*: *mpinab* is a compiler that will produce an MPI-enabled executable from source code

1.3. Combining AmberTools12 with Amber11 or Amber10

written in the NAB language. Before compiling *mpinab*, be sure that you are familiar with the serial version of *nab* and that you really need a parallel version. If you have shared-memory nodes, the OpenMP version might be a better alternative. See Section 14.4 for more information. (Note that *mpinab* is primarily designed to write driver routines that call MPI versions of the energy functions; it is not set up to write your own, novel, parallel codes.)

- b) The MPI version of *MMPBSA.py* is called *MMPBSA.py.MPI*, and requires the package *mpi4py* to run. If it is not present in your system Python installation already, it will be built along with *MMPBSA.py.MPI* and placed in the `$AMBERHOME/bin` directory. If you have problems with *MMPBSA.py.MPI*, see if you get the same problems with the serial version, *MMPBSA.py*, to see if it is an issue with the parallel version or *MMPBSA.py* in general.

8. NAB and Cpptraj can also be compiled using OpenMP:

```
./configure --openmp <....other options....> <compiler-choice>
make openmp
```

Note that the OpenMP versions of NAB and Cpptraj have the same name as the single-threaded version. See Section 14.4 for information on running the OpenMP version of NAB and section 7.2.4 for information on running the OpenMP version of Cpptraj.

1.3. Combining AmberTools12 with Amber11 or Amber10

It is certainly feasible to combine AmberTools12 with earlier versions of Amber. Here is the outline of what to do:

1. Unpack *AmberTools12.tar.bz2* into a directory tree whose head will be `amber12`. Point your `AMBERHOME` environment to this directory, and install and test as described above. Don't do anything to your existing Amber files, which will be in a directory tree headed by `amber11` (or `amber10...`).
2. Set your `PATH` variable to have `$AMBERHOME/bin` ahead of the old amber directories. For example:

```
export PATH="$AMBERHOME/bin:/home/myname/amber11/bin:$PATH" # (for bash, zsh, ksh,
setenv PATH "$AMBERHOME/bin:/home/myname/amber11/bin:$PATH" # (for csh)
```

In this way, your `PATH` will see the AmberTools12 codes first, but will also find the earlier versions of *sander*, *pmemd*, and other Amber codes. You will probably want to put one of the above commands (along with the definition of `AMBERHOME`) into your startup script (e.g. `~/.bashrc`, `~/.zshrc`, `~/.cshrc`, etc.)

1.4. Testing the installation

We have installed and tested Amber on a number of platforms, using UNIX, Linux, Microsoft Windows or Macintosh OSX operating systems. However, owing to time and access limitations, not all combinations of code, compilers, and operating systems have been tested. Therefore we recommend running the test suites.

The distribution contains a validation suite that can be used to help verify correctness. The nature of molecular dynamics, is such that the course of the calculation is very dependent on the order of arithmetical operations and the machine arithmetic implementation, *i.e.*, the method used for round-off. Because each step of the calculation depends on the results of the previous step, the slightest difference will eventually lead to a divergence in trajectories. As an initially identical dynamics run progresses on two different machines, the trajectories will eventually become completely uncorrelated. Neither of them are "wrong;" they are just exploring different regions of phase space. Hence, states at the end of long simulations are not very useful for verifying correctness. Averages are meaningful, provided that normal statistical fluctuations are taken into account. "Different machines" in this context means any difference in floating point hardware, word size, or rounding modes, as well as any differences in compilers or libraries. Differences in the order of arithmetic operations will affect round-off behavior; $(a + b) + c$ is not necessarily the same as $a + (b + c)$. Different optimization levels will affect operation order, and may therefore affect the course of the calculations.

All initial values reported as integers should be identical. The energies and temperatures on the first cycle should be identical. The RMS and MAX gradients reported in sander are often more precision sensitive than the energies, and may vary by 1 in the last figure on some machines. In minimization and dynamics calculations, it is not unusual to see small divergences in behavior after as little as 100-200 cycles.

1.5. Applying Updates

AmberTools 12 and Amber 12 have introduced a new method for downloading and applying 'feature adjustments' as they become available. It also keeps track of which patch files have been installed to avoid applying patches multiple times. It can also be used to determine which bugs have been fixed, etc. When you run *configure*, *patch_amber.py* is automatically run to check for available updates (unless you explicitly disable that). If any are available, you will be asked if you want them downloaded and applied. The Python script that does this is called *patch_amber.py* (Python2-compatible). This script resides in \$AMBERHOME and can be executed from anywhere (it will verify that AMBERHOME is set properly), but if moved from AMBERHOME, it will not work. There are 3 main operating modes, or actions, that you can perform with them:

- `$AMBERHOME/patch_amber.py --check-updates` : This option will query the Amber website for any bug fixes that have been posted for AmberTools12 or Amber12 that have not been applied to your installation. If you think you have found a bug, this is helpful to try first before emailing with problems since your bug may have already been fixed.
- `$AMBERHOME/patch_amber.py --patch-level` : This option will return which patches

have been applied to the current tree so far. When emailing the Amber list with problems, it is important to have the output of this command, since that lets us know which bug fixes have already been applied.

- `$AMBERHOME/patch_amber.py --update-tree` : This option will go to the Amber website, download all updates that have not been applied to your installation, and apply them to the source code. Note that you will have to recompile for the changes to take effect

1.5.1. Advanced options

`patch_amber.py` has additional functionality as well that allows more intimate control over the patching process. For a full list of options, use the `--help` or `-h` command-line options. These are considered advanced options.

- `$AMBERHOME/patch_amber.py --download-patches` : Only download patches, do not apply them
- `$AMBERHOME/patch_amber.py --apply-patch=PATCH_FILE` : Apply an individual patch file. It can be an external patch from a third-party source if desired. (For patch file creators, `patch_amber.py` applies patches using the `patch` program from `$AMBERHOME`, so the patch file must be designed accordingly)
- `$AMBERHOME/patch_amber.py --reverse-patch=PATCH_FILE` : Reverses an installed patch file. It will update its list of applied patches and remove the unapplied one from the list of applied patches.
- `$AMBERHOME/patch_amber.py --show-applied-patches` : Shows details about each patch that has been applied.

`patch_amber.py` will also provide varying amounts of information about each patch based on the verbosity setting. There are 3 different settings in decreasing order of verbosity:

1. `--verbose` : This flag will print out all information about a particular patch, including the name of the author, the programs that it fixes, the date it was created, and the full description of the bug that it targets.
2. `--quiet` : This flag will print out some information about a particular patch, including the name of the author, the program it fixes, and the date it was created (it will NOT print a description of the bug that it targets).
3. `--dead-silent` : This flag is actually a bit of a misnomer. It will print out the patch number (or patch name, like *bugfix.4*), but nothing else. This is the default setting.

1.6. Contacting the developers

Please send suggestions and questions to amber@ambermd.org. You need to be subscribed to post there; to subscribe, go to <http://lists.ambermd.org/mailman/listinfo/amber>.

1.7. List of programs

AmberTools is comprised of a large number of programs designed to aid you in your computational studies of chemical systems, and the number of released tools grows regularly. This section provides a comprehensive list of programs included with *AmberTools*. Each program included in the suite is listed here with a very brief description of its main function along with which chapter in the manual a more thorough description can be found.

AddToBox A program for adding solvent molecules to a crystal cell. See [13.4.3](#)

ChBox A program for changing the box dimensions of an Amber restart file. See [13.4.4](#)

CheckMD A program for automated checking of an MD simulation. Run the program without options for usage statement.

MCPB A semi-automated tool for metalloprotein parametrization. See [4.6](#)

MMPBSA.py A program to post-process trajectories to calculate binding free energies according to the MM/PBSA approximation. See [10](#)

MTKppConstants Lists the constants used in MTK++. Run the program without arguments to get the full list.

PropPDB A program for propagating a PDB structure. See [13.4.2](#)

UnitCell A program for recreating a crystallographic unit cell from a PDB structure. See [13.4.1](#)

acdoctor A tool to diagnose what may be causing antechamber to fail. See [4.4.1](#)

add_pdb A program to add sections to a topology file that correspond to PDB information. The input files are given by *-i <input_prmtop>* and *-p <matching_pdb>*, with the output prmtop being given by the flag *-o <output_prmtop>*. It adds the sections RESIDUE_NUMBER, RESIDUE_CHAINID, and ATOM_ELEMENT to the topology file, leaving the rest unchanged. Note that the *input_prmtop* and *matching_pdb* must match each other.

am1bcc A program called by antechamber to calculate AM1-BCC charges during ligand parametrization. It can be used as a standalone program, with the options printed when you enter the program name with no arguments. See [4.3](#)

ambpdb A program to convert an Amber system (prmtop and inpcrd/restart) into a PDB, MOL2, or PQR file. See [13.1](#)

ante-MMPBSA.py A program to create the necessary, self-consistent prmtop files for *MMPBSA* with a single starting topology file. See [10.2.2](#)

antechamber A program for parametrizing ligands and other small molecules. See [4](#)

atomtype A program called by *antechamber* to judge the atom types in an input structure. It can be used as a standalone program. If you provide no arguments, it prints out the usage statement. See [4.3](#)

- bondtype** A program called by *antechamber* to judge what types of bonds exist in a given input structure. It can be used as a standalone program. If you provide no arguments, it prints out the usage statement. See [4.3](#)
- calcpka** A program that calculates fraction protonation, and Hendersen-Hasselbalch pKas from constant pH simulations conducted with *Amber*. If you provide no arguments, it prints out the usage statement.
- capActiveSite** A program to cap the active site of a protein using a cutoff. The flag *-h* prints the usage message.
- chamber** A program to convert a CHARMM psf file to an Amber topology (prmtop) file. See [2.13](#)
- charmmlipid2amber.x** A script that converts a PDB created with the CHARMM-GUI lipid builder into one recognized by Amber and AmberTools programs. If you provide no arguments, it prints out the usage statement.
- cpinutil** A program to create a constant pH input (CPin) file from a PDB file. If you provide no arguments, you get the usage statement.
- cpptraj** A versatile program for trajectory post-processing similar to ptraj. It has some overlapping, and some different features. See [7](#)
- elsize** A program that estimates the effective electrostatic size of a given input structure. See [13.3](#)
- espgen** A program called by *antechamber* to generate ESP files during ligand or small molecule parametrization. If you provide no arguments, it prints out the usage statement.
- frcmod2xml** A program that converts an Amber frcmod file to an XML file that can be interpreted by MTK++ (used by MCPB and related programs). Providing no arguments returns the usage message.
- func** A program that determines the functional groups in ligands. Providing no arguments returns the usage message. (Part of the MCPB and related programs)
- hcp_getpdb** A program that adds necessary sections to a topology (prmtop) file so it can be used for the HCP GB approximation. See [18.5](#)
- lmodprmtop** A program that adds VDW walls to all atoms for LMOD searches. See [18.4.7](#).
Use it as
lmodprmtop <input_prmtop> <output_prmtop>
- matextract** Part of the symmetry definition programs, used to print matrices dumped to stdin to stdout. See [16.5.5](#)
- matgen** Generate symmetry-transformation matrices. Part of the symmetry definition programs. See [16.5.1](#)

1. Getting started

matmerge Merges symmetry-transformation matrices into one matrix transformation matrix. Part of the symmetry definition programs. See [16.5.3](#)

matmul Multiplies matrices. Part of the symmetry definition programs. See [16.5.4](#)

mdgx An explicit solvent, PME molecular dynamics engine. See [11](#)

mmE A program that calculates Amber energies and gradients. Part of the MCPB/MTK++ packages. Providing no arguments prints the usage message.

mmpbsa_py_energy A NAB program written to calculate energies for *MMPBSA* using either GB or PB solvent models. It can be used as a standalone program that mimics the *imin=5* functionality of *sander*, but it is called automatically inside *MMPBSA*. See *MMPBSA* mdin files as example input files for this program. Providing the *-help* or *-h* flags prints the usage message.

mmpbsa_py_nabnmode A NAB program written to calculate normal mode entropic contributions for *MMPBSA*. This can really only be used by *MMPBSA*.

molsurf A program that calculates a molecular surface area based on input PQR files and a probe radius. Providing no arguments prints the usage message.

nab Stands for Nucleic Acid Builder. NAB is really a compiler that provides a convenient molecular programming language loosely based on C. See [14](#) and other related chapters.

ncdump Program to dump all of the data from NetCDF files (mdcrd, mdvel, etc.) in a human-readable format to stdout. This is built with NetCDF, so consult their documentation for detailed usage instructions. Basic usage is as follows:
ncdump <netcdf_file>

ncgen A program that generates NetCDF files. This is built with NetCDF, so consult their documentation for usage details.

paramfit Improves force field parameters by fitting to quantum data. See [12](#)

parmcal Calculates parameters for given angles and bonds interactively. See [4.4.2](#)

parmchk A program that analyzes an input force field library file (mol2 or amber prep), and extracts relevant parameters into an frcmod file. See [4.1.2](#)

parmed.py A program for querying and manipulating prmtop files. See [13.5](#)

pbsa A finite difference Poisson-Boltzmann solver. See [8](#)

pdbSearcher Searches a local PDB database. Part of the MCPB/MTK++ package. Use no arguments to get the usage message

prep2xml Converts Amber prep file to an XML file format that can be understood by MCPB/MTK++. Use no arguments to get the usage message.

- prepgen** A program used as part of *antechamber* that generates an Amber prep file. Use no arguments to print the usage message. See [4.3](#)
- process_mdout.perl** A perl script that parses the mdout files from a molecular dynamics simulation and dumps statistics that can be plotted. It is used extensively in the online tutorials found on the amber website (<http://ambermd.org/tutorials/>)
- process_minout.perl** A perl script just like *process_mdout.perl* for minimization output files.
- protonator** A program to add protons to chemical systems. Part of the MCPB/MTK++ package. Use no arguments to print a usage message.
- ptraj** A trajectory post-processing tool. See [A](#)
- rdparm** A program to parse and provide details about a given prmtop file. See [A.9](#)
- reduce** A program for adding or removing hydrogen atoms to a PDB. See [13.2](#)
- residuegen** A program to automate the generation of an Amber residue template (i.e. Amber prep file). See [4.4.3](#)
- resp** A program typically called by *antechamber* and R.E.D. tools to perform a Restrained ElectroStatic Potential calculation for calculating partial atomic charges. Use no arguments to get the usage message
- respgen** A program called by *antechamber* to generate RESP input files. See [4.3](#)
- rism1d** A 1D-RISM solver. See [9.4](#)
- rism3d.snglpnt** A 3D-RISM solver for single point calculations. See [9.6](#)
- sequenceAligner** A program for sequence alignment and structural superimposition. Part of the MCPB/MTK++ package. Use no arguments to get a usage message.
- softcore_setup.py** A program to aid in softcore TI setup. Use no arguments to get a usage message.
- sqm** Semiempirical (or Stand-alone) Quantum Mechanics solver. See [6](#)
- stats** A simple statistics program. Part of the MCPB/MTK++ package.
- stdLib2Sdf** A program to convert a standard XML library file into an SDF library file. Part of the MCPB/MTK++ package. Use no arguments to get a usage message.
- superimposer** A program to do structural superimposition. Part of the MCPB/MTK++ package. Use no arguments to get a usage message directly.
- tleap** A script that calls teLeap with specific setup command-line arguments. See [3](#)
- transform** Applies matrix transformations to a structure. Part of the symmetry definition programs. See [16.5.6](#)

1. Getting started

tss_init A program to do some matrix stuff. See [16.5](#)

tss_main A program to do some matrix stuff. See [16.5](#)

tss_next A program to do some matrix stuff. See [16.5](#)

ucpp A program to do some preprocessing. You should never actually use this program.

xaLeap A graphical program for creating Amber topology files. This program is called through the xleap script, so you should never actually invoke this program directly.

xleap A script that calls xaLeap with specific setup command-line arguments. See [3](#)

xparmed.py A graphical front-end to ParmEd functionality (i.e., parameter file editing and querying). See [13.5](#)

Parallel programs (they operate the same as their serial counterparts):

mpinab2c MPI version of *nab2c*

mpinab MPI version of *nab*

MMPBSA.py.MPI MPI version of *MMPBSA*

2. Specifying a force field

Amber is designed to work with several simple types of force fields, although it is most commonly used with parametrizations developed by Peter Kollman and his co-workers. There are now several such parametrization, with no obvious “default” value. The “traditional” parametrization uses fixed partial charges, centered on atoms. The current recommended force fields of this type is *ff12*, although *ff03.r1* is also commonly used; descriptions are given below. Less extensively used, but very promising, modifications add polarizable dipoles to atoms, so that the charge description depends upon the environment; such potentials are called “polarizable” or “non-additive”. Examples are *ff02* and *ff02EP*: the former has atom-based charges (as in the traditional parametrization), and the latter adds in off-center charges (or “extra points”), primarily to help describe better the angular dependence of hydrogen bonds. Major updates to these are under development, but were not ready for the Amber12 release in March, 2012.

An alternative is to use force fields originally developed for the CHARMM codes; this requires a completely different setup procedure, which is described in Section 2.13, below.

In order to tell LEaP which force field is being used, the four types of information described below need to be provided. This is generally accomplished by selecting an appropriate *leaprc* file, which loads the information needed for a specific force field (see also section 2.2, below).

1. A listing of the atom types, what elements they correspond to, and their hybridizations. This information is encoded as a set of LEaP commands, and is normally read from a *leaprc* file.
2. Residue descriptions (or “residue topologies”) that describe the chemical nature of amino acids, nucleotides, and so on. These files specify the connectivities, atom types, charges, and other information. These files have a “prep” format (a now-obsolete part of Amber) and the extension “.in”. Standard libraries of residue descriptions are in the *\$AMBERHOME/dat/leap/lep* directory. The *antechamber* program may be used to generate prep files for other organic molecules.
3. Parameter files give force constants, equilibrium bond lengths and angles, Lennard-Jones parameters, and the like. Standard files have a “.dat” extension, and are found in *\$AMBERHOME/dat/leap/parm*.
4. Extensions or changes to the parameters can be included in *frmod* files. The expectation is that the user will load a large, “standard” parameter file, and, if required, a smaller *frmod* file that describes any needed supplementary parameters or adjustments to the standard parameters. The *frmod* files for changing the default water model (which is TIP3P) into other water models are in files like *\$AMBERHOME/dat/leap/parm/frmod.tip4p*. The *parmchk* program (part of *antechamber*) can also generate *frmod* files.

2. Specifying a force field

2.1. Specifying which force field you want in LEaP

Various combinations of the above files make sense, and we have moved to an “ff” (force field) nomenclature to identify these; examples would then be *ff94* (which was the default in Amber 5 and 6), *ff99*, etc. The most straightforward way to specify which force field you want is to use one of the leaprc files in *\$AMBERHOME/dat/leap/cmd*. The syntax is

```
xleap -s -f <filename>
```

Here, the *-s* flag tells LEaP to ignore any leaprc file it might find, and the *-f* flag tells it to start with commands for some other file. Here are the combinations we support and recommend:

File name	Topology	Parameters
leaprc.ff12SB	Cornell <i>et al.</i> , 1994	see Sec. 2.2
leaprc.ff03.r1	Duan <i>et al.</i> 2003	parm99.dat+frcmod.ff03
leaprc.ff03ua	Yang <i>et al.</i> 2003	parm99.dat+frcmod.ff03+frcmod.ff03ua
leaprc.ff02	reduced charges	parm99.dat+frcmod.ff02pol.r1
leaprc.gaff	none	gaff.dat
leaprc.GLYCAM_06h	Woods <i>et al.</i>	GLYCAM_06h.dat
leaprc.GLYCAM_06EPb	"	GLYCAM_06EPb.dat
leaprc.amoeba	Ren & Ponder	Ren & Ponder
leaprc.lipid11	Skjevik <i>et al.</i> , 2012	lipid11.dat (see Ref. [9])

Notes:

1. There is no default leaprc file. If you make a link from one of the files above to a file named *leaprc*, then that will become the default. For example:

```
cd $AMBERHOME/dat/leap/cmd
ln -s leaprc.ff12SB leaprc
```

will provide a good default for many users; after this you could just invoke *tleap* or *xleap* without any arguments, and it would automatically load the *ff12SB* force field. A file named *leaprc* in the working directory overrides any other such files that might be present in the search path.

2. Most of the choices in the above table are for additive (non-polarizable) simulations; you should use *saveAmberParm* to save the prmtop file, and keep the default *ipol=0* in *sander* or *pmemd*.
3. The *ff02* entries in the above table are for non-additive (polarizable) force fields. Use *saveAmberParmPol* to save the prmtop file, and set *ipol=1* in the sander input file. Note that POL3 is a polarizable water model, so you need to use *saveAmberParmPol* for it as well.
4. There is also a *leaprc.gaff* file, which sets you up for the GAFF (“general” Amber) force field. This is primarily for use with Antechamber (see 4), and does not load any topology files.

	ff99SB	ff10	ff12SB
<i>proteins</i>	= ff99 + backbone torsion modifications	no change from ff99SB	= ff99SB + new backbone and sidechain torsions
<i>DNA</i>	= ff99	= ff99 + “Barcelona” backbone torsion modifications	no change from ff10
<i>RNA</i>	= ff99	= ff99 + “Barcelona” backbone changes + “OL3” changes for χ	no change from ff10

Table 2.1.: Changes in recent fixed-charge forcefields for proteins and nucleic acids

- There are some leaprc files for older force fields in the `$AMBERHOME/dat/leap/cmd/oldff` directory. We no longer recommend these combinations, but we recognize that there may be reasons to use them, especially for comparisons to older simulations.
- Nucleic acid residues in *ff12SB* use the new (version 3) PDB nomenclature: “DC” is used for deoxy-cytosine, and “C” for cytosine in RNA, etc. Earlier force fields (which are *not* recommended!) use “RC” for the RNA version. If you want a single, nucleoside, use “CN”, etc. For a single nucleotide, use the following command in LEaP:

```
cnuc = sequence { OHE C3 }
```

and analogs for other bases. Note that this will construct a protonated 5' phosphate group, which may not be what you want.

- The General Amber Force Field (**gaff**) is discussed in Chap. 4.
- Information on some older (now obsolete) force fields is given in Appendix B.

2.2. The ff12SB force field

<code>leaprc.ff12SB</code>	This will load the files listed below
<code>parm10.dat</code>	ff10 force field parameters
<code>frcmod.ff12SB</code>	ff12SB modifications to parm10.dat
<code>amino12.lib</code>	topologies and charges for amino acids
<code>amino12nt.lib</code>	same, for N-terminal amino acids
<code>amino12ct.lib</code>	same, for C-terminal amino acids
<code>nucleic12.lib</code>	topologies and charges for nucleic acids

For proteins, the **ff12SB** force field starts with **ff10** and updates the protein backbone and side chain dihedral corrections, as described below. For nucleic acids, ff12 is the same as ff10; see

2. Specifying a force field

Table 2.1 for a summary of recent changes in the Amber fixed-charge force fields. **Note:** ff12SB calculations must be carried out with Amber12 if using GB7, GB8, or GBSA.

2.2.1. Backbone modifications

ff99SB has been demonstrated to understabilize helical conformations of transiently folded peptides. Therefore, a principal goal of ff12SB was to predict accurate secondary structure propensities. Of candidate force fields adjusting the ϕ' and ψ' parameters to enhance α /ppII stability, modification of only ϕ' most accurately reproduced the delicate balance of secondary structure indicated by experiments. We extensively tested three candidate force fields in a diverse range of systems modifying this torsional term. The one that best reproduces secondary structure, order parameters, and vicinal scalar couplings is distributed here.

2.2.2. Side chain modifications

The side chain dihedral parameters of ff99SB are the same as those of ff94. Residues such as isoleucine, leucine, aspartate, and asparagine (*cf.* ff99SBildn) sample conformations different from those indicated by experiments. We therefore refined the dihedral corrections of the amino acid side chains by fitting energy profiles to match *ab initio* quantum data.

A key objective in the ff12SB fitting was to develop parameters that are robust with variation of the local environment, including backbone conformation, of which the training set possesses a limited number, and solvent, notably absent from the training. Since side-chain preferences reproducibly vary with backbone conformation, we employed multiple backbone conformations of each amino acid to partially account for energy backbone-dependence. We also did not preferentially solve our corrections for certain side chain conformers that happen to be stable at a particular backbone conformation of a dipeptide *in vacuo*.

Where particularly strong non-bonded interactions occur, minor deficiencies in non-bonded models may manifest as significant, structurally-dependent energy errors. This is especially true since Amber charges are not particularly *in vacuo* charges. Strong non-bonded interactions may also induce strain, exposing errors in bond length or angle representation far away from the ground state. Since the goal is to fit robust parameters describing local dihedral torsion effects that are appropriate as other structural features may change, we removed from our training any structures where atoms not in a bond, angle, or torsion with each other were particularly close. We also restrained all backbone dihedrals, including hydrogens, to further avoid overly strong vacuum non-bonded interactions.

Together with new corrections for the backbone and the four amino acids addressed in ff99SBildn, this work offers updated side chain dihedral corrections for lysine, arginine, glutamate, glutamine, methionine, serine, threonine, valine, tryptophan, cysteine, phenylalanine, tyrosine, and histidine. ff12SB enhances reproduction of experimentally indicated geometries over ff99SB.

2.3. The ff10 force field

```
leaprc.ff10
```

```
This will load the files listed below
```

<code>parm10.dat</code>	Basic force field parameters
<code>amino10.in</code>	topologies and charges for amino acids
<code>amino10nt.in</code>	same, for N-terminal amino acids
<code>amino10ct.in</code>	same, for C-terminal amino acids
<code>nucleic10.in</code>	topologies and charges for nucleic acids

The **ff10** force field collects a variety of updates and modifications to the (generally rather successful) *ff99* force field, which is described below in Section B.3. It incorporates the following additions and changes:

1. **ff99SB for proteins.** Several groups have noticed that *ff99* (and *ff94* as well) do not provide a good energy balance between helical and extended regions of peptide and protein backbones. Another problem is that many of the *ff94* variants had incorrect treatment of glycine backbone parameters. *ff99SB* is the recent attempt to improve this behavior, and was developed in the Simmerling group.[10] It presents a careful reparametrization of the backbone torsion terms in *ff99* and achieves much better balance of four basic secondary structure elements (PP II, β , α_L , and α_R). A detailed explanation of the parametrization as well as an extensive comparison with many other variants of fixed-charge Amber force fields is given in the reference above. Briefly, dihedral term parameters were obtained through fitting the energies of multiple conformations of glycine and alanine tetrapeptides to high-level *ab initio* QM calculations. We have shown that this force field provides much improved proportions of helical versus extended structures. In addition, it corrected the glycine sampling and should also perform well for β -turn structures, two things which were especially problematic with most previous Amber force field variants.
2. **ff99bsc0 and chi.OL3 for nucleic acids.** The nucleic acid force fields have recently been updated from those in *ff99*, in order to address a tendency of DNA double helices to convert (after fairly long simulations) to extended forms in the α and γ backbone torsion angles gauche+, trans and twisting of the sugar into the plane of the base.[11] Also, in order to improve the χ glycosidic torsional potential for RNA, recent work has proposed additional force field modifications[12, 13] (see also an alternative set of χ modifications below). These force field modifications lead to better representations of both DNA and RNA; note that the χ modifications only apply to RNA.
3. **Updated ion parameters.** Recently, Joung and Cheatham have created a more consistent set of parameters, fitting solvation free energies, radial distribution functions, ion-water interaction energies and crystal lattice energies and lattice constants for non-polarizable spherical ions.[14, 15] These have been separately parametrized for each of three popular water models, as indicated above. Please note: you need to load an additional `frmod` file specific to the water model you are using; see Section 2.11.
4. **New atom and residue names.** The residue and atom names in *ff10* now comply with PDB format version 3.

2.3.1. Variants of ff10

1. **Torsional modifications to ff99SB.** Two recent modifications to *ff99SB* for proteins have been proposed. The first, *ff99SBildn*, changes side chain torsions for the amino acids

2. Specifying a force field

isoleucine, leucine, aspartate and asparagine.[16] A second, *ff99SBnmr*, suggests modifications to backbone torsions based on fits to NMR data. [17] Please see the cited papers for further information. These are not in *ff10*: you need to load the topology and parameter files listed below to access them.

<code>ions08.lib</code>	updated parameters for ions
<code>frcmod.ff99SBildn</code>	modified side-chain torsions
<code>all_amino94ildn.lib</code>	topologies for the "ff99SBildn" force field
<code>frcmod.ff99SBnmr</code>	modified backbone dihedrals, based on NMR analyses

2. **Alternate glycosidic torsions for RNA.** Another parametrization of the χ torsion profiles in RNA has been developed by Yildirim et al. [18, 19]. You can use this set (instead of the "OL3" set in *ff10*) by loading `leaprc.parmCHI_YIL.bsc`; as the name suggests, this modification to *ff99* also includes the "bsc0" α and γ updates as well as modified torsional parameters for the χ torsion in RNA.

2.4. The AMOEBA potentials

The **amoeba** force field for proteins, ions, organic solvents and water, developed by Ponder and Ren [20–24], is available in *sander* and *pmemd.amoeba*. This force field is specified by setting *iamoeba* to 1 in the input file. Right now, setting up the system is a bit complex: you need to set up the system in Tinker (version 4.3), then run the *tinker-to-amber* program to convert to Amber prmtop and coordinate files. Some examples are in `$AMBERHOME/src/pmemd.amoeba/build_amoeba`. But keep checking the Amber web page, since we hope to provide a simpler path soon.

2.5. The Duan et al. (2003) force field

<code>frcmod.ff03</code>	For proteins: changes to <code>parm99.dat</code> , primarily in the phi and psi torsions.
<code>all_amino03.in</code>	Charges and atom types for proteins
<code>all_aminont03.in</code>	For N-terminal amino acids
<code>all_aminoc03.in</code>	For C-terminal amino acids

The **ff03** force field [25, 26] is a modified version of *ff99* (described below). The main changes are that charges are now derived from quantum calculations that use a continuum dielectric to mimic solvent polarization, and that the ϕ and ψ backbone torsions for proteins are modified, with the effect of decreasing the preference for helical configurations. The changes are just for proteins; nucleic acid parameters are the same as in *ff99*.

The original model used the old (*ff94*) charge scheme for N- and C-terminal amino acids. This was what was distributed with Amber 9, and can still be activated by using `oldff/leaprc.ff03`. More recently, new libraries for the terminal amino acids have been constructed, using the same charge scheme as for the rest of the force field. This newer version (which is recommended for all new simulations) is accessed by using `leaprc.ff03.r1`.

2.6. The Yang et al. (2003) united-atom force field

<code>frcmod.ff03ua</code>	For proteins: changes to <code>parm99.dat</code> , primarily in the introduction of new united-atom carbon types and new side chain torsions.
<code>uni_amino03.in</code>	Amino acid input for building database
<code>uni_aminont03.in</code>	NH3+ amino acid input for building database.
<code>uni_aminoc03.in</code>	COO- amino acid input for building database.

The **ff03ua** force field [27] is the united-atom counterpart of *ff03*. This force field uses the same charging scheme as *ff03*. In this force field, the aliphatic hydrogen atoms on all amino acid side-chains are united to their corresponding carbon atoms. The aliphatic hydrogen atoms on all alpha carbon atoms are still represented explicitly to minimize the impact of the united-atom approximation on protein backbone conformations. In addition, aromatic hydrogens are also explicitly represented. Van der Waals parameters of the united carbon atoms are refitted based on solvation free energy calculations. Due to the use of an all-atom protein backbone, the ϕ and ψ backbone torsions from *ff03* are left unchanged. The sidechain torsions involving united carbon atoms are all refitted. In this parameter set, nucleic acid parameters are still in all atom and kept the same as in *ff99*.

2.7. The 2002 polarizable force fields

<code>parm99.dat</code>	Force field, for amino acids and some organic molecules; can be used with either additive or non-additive treatment of electrostatics.
<code>parm99EP.dat</code>	Like <code>parm99.dat</code> , but with "extra-points": off-center atomic charges, somewhat like lone-pairs.
<code>frcmod.ff02pol.r1</code>	Updated torsion parameters for <i>ff02</i> .
<code>all_nuc02.in</code>	Nucleic acid input for building database, for a non-additive (polarizable) force field without extra points.
<code>all_amino02.in</code>	Amino acid input ...
<code>all_aminoc02.in</code>	COO- amino acid input ...
<code>all_aminont02.in</code>	NH3+ amino acid input
<code>all_nuc02EP.in</code>	Nucleic acid input for building database, for a non-additive (polarizable) force field with extra points.
<code>all_amino02EP.in</code>	Amino acid input ...
<code>all_aminoc02EP.in</code>	COO- amino acid input ...
<code>all_aminont02EP.in</code>	NH3+ amino acid input

The **ff02** force field is a polarizable variant of *ff99*. (See Ref. [28] for a recent overview of polarizable force fields.) Here, the charges were determined at the B3LYP/cc-pVTZ//HF/6-31G* level, and hence are more like "gas-phase" charges. During charge fitting the correction for

2. Specifying a force field

intramolecular self polarization has been included.[29] Bond polarization arising from interactions with a condensed phase environment are achieved through polarizable dipoles attached to the atoms. These are determined from isotropic atomic polarizabilities assigned to each atom, taken from experimental work of Applequist. The dipoles can either be determined at each step through an iterative scheme, or can be treated as additional dynamical variables, and propagated through dynamics along with the atomic positions, in a manner analogous to Car-Parinello dynamics. Derivation of the polarizable force field required only minor changes in dihedral terms and a few modification of the van der Waals parameters.

Recently, a set up updated torsion parameters has been developed for the *ff02* polarizable force field.[30] These are available in the *frmod.ff02pol.r1* file.

The user also has a choice to use the polarizable force field with extra points on which additional point charges are located; this is called **ff02EP**. The additional points are located on electron donating atoms (e.g. O,N,S), which mimic the presence of electron lone pairs.[31] For nucleic acids we chose to use extra interacting points only on nucleic acid bases and not on sugars or phosphate groups.

There is not (yet) a full published description of this, but a good deal of preliminary work on small molecules is available.[29, 32] Beyond small molecules, our initial tests have focused on small proteins and double helical oligonucleotides, in additive TIP3P water solution. Such a simulation model, (using a polarizable solute in a non-polarizable solvent) gains some of the advantages of polarization at only a small extra cost, compared to a standard force field model. In particular, the polarizable force field appears better suited to reproduce intermolecular interactions and directionality of H-bonding in biological systems than the additive force field. Initial tests show *ff02EP* behaves slightly better than *ff02*, but it is not yet clear how significant or widespread these differences will be.

2.8. Force fields related to semi-empirical QM

ParmAM1 and **parmPM3** are classical force field parameter sets that reproduce the geometry of proteins minimized at the semi-empirical AM1 or PM3 level, respectively.[33] These new force fields provide an inexpensive, yet reliable, method to arrive at geometries that are more consistent with a semi-empirical treatment of protein structure. These force fields are meant only to reproduce AM1 and PM3 geometries (warts and all) and were not tested for use in other instances (e.g., in classical MD simulations, etc.) Since the minimization of a protein structure at the semi-empirical level can become cost-prohibitive, a “preminimization” with an appropriately parametrized classical treatment will facilitate future analysis using AM1 or PM3 Hamiltonians.

2.9. The GLYCAM-06 and GLYCAM-06EP force fields for carbohydrates and lipids

The latest releases of GLYCAM parameters, prep files, libraries, leaprc files and other documentation can be obtained from the Woods group’s GLYCAM Force Field Parameters

2.9. The GLYCAM-06 and GLYCAM-06EP force fields for carbohydrates and lipids

Download Page (glycam.org/params). Researchers are strongly encouraged to obtain the most recent files for their projects. Online file names might vary slightly from those listed in this document.

GLYCAM 2006 force field

GLYCAM_06h.dat	Parameters for oligosaccharides (Check www.glycam.org for more recent versions)
GLYCAM_06h.prep	Structures and charges for glycosyl residues
GLYCAM_lipids_06h.prep	Structures and charges for sample lipid residues (Check www.glycam.org for additional residues)
leaprc.GLYCAM_06h	LEaP configuration file for GLYCAM-06
GLYCAM_amino_06h.lib	Glycoprotein library for centrally-positioned residues
GLYCAM_aminoc_06h.lib	Glycoprotein library for C-terminal residues
GLYCAM_aminot_06h.lib	Glycoprotein library for N-terminal residues

GLYCAM 2006EP force field using lone pairs (extra points)

GLYCAM_06EPb.dat	Parameters for oligosaccharides
GLYCAM_06EPb.prep	Structures and charges for glycosyl residues
leaprc.GLYCAM_06EPb	LEaP configuration file for GLYCAM-06EP

GLYCAM Force Field Parameters Download Page

<http://www.glycam.org/params>

GLYCAM_06h.prep contains prep entries for all carbohydrate residues and GLYCAM_lipids_06h.prep contains prep entries for lipid residues. GLYCAM_06EPb.prep contains prep entries for all carbohydrate residues available for modeling with extra points.

For linking glycans to proteins, libraries containing modified amino acid residues (Ser, Thr, Hyp, and Asn) must be loaded. GLYCAM_amino_06h.lib GLYCAM_aminot_06h.lib and GLYCAM_aminoc_06h.lib contain entries for centrally located, N-terminal and C-terminal amino acids, respectively. Amino acid libraries designed for linking carbohydrates modeled with extra points are not currently available.

2.9.1. File versioning

Beginning on 15 September, 2011, a new versioning system was implemented for Glycam parameters. Files produced before that date will not necessarily conform to the new system. In the new system, all files containing parameters are versioned. Since they are provided merely as a convenience, the “leaprc” files, will not be versioned. Users should check their contents and modify them with recent parameters as appropriate.

The new versioning system employs letters and numbers. If a parameter set contains new functionality (e.g., the addition of new parameters) or fundamental changes (e.g., atom type

2. Specifying a force field

Version	Release Date	Contributors	Change Summary
h	20 Oct., 2010	MBT, BLF	<i>*Changed atom type naming to be orthogonal to other force fields.</i> Added HO van der Waals parameters. Set protein-related parameter values to their parm99 counterparts. Updated N-sulfation parameters.
g	20 Oct., 2010 HERE	MBT	* 1,4-scaling terms added to parameter file. Angle and torsion updates for pyranose rings, N-sulfate, phosphate and sialic acid.
f	3 Feb., 2009	MBT	* Corrected a typo in O-Acetyl term
e	28 May, 2008	MBT	* Updated glycosidic linkage terms to optimize ring puckering in pyranoses
d	12 May, 2008	SPK, MBT, ABY	Terms for thiol glycosidic linkages
c	21 Feb., 2008	MBT, ABY	* Additional (published) terms for lipid simulations[34]
b	10 Jan., 2008	MBT, ABY	Alkanes, alkenes, amide and amino groups for lipid simulations[34]
a	24 Apr., 2005	ABY	Sulfates & phosphates for carbohydrates

Table 2.2.: *Version change summary for the GLYCAM-06 force field. *Previously released parameters were changed. See full release notes at glycam.org/params. SPK: Sameer P. Kawatkar. MBT: Matthew B. Tessier. ABY: Austin B. Yongye. BLF: B. Lachele Foley.*

name reassignments), a letter will be appended to its name. If the new version contains corrections (e.g., for typographical errors), its name will be appended with a number. See glycam.org/params for more documentation and examples.

Researchers are also encouraged to read the version change documentation available on the GLYCAM Parameters download page under “Documents.” In this document, the changes specific to each version release are detailed. The changes are also summarized here in Table 2.2.

2.9.2. Atom type name changes in the current versions

Beginning with the current versions, Glycam atom type names will adopt a standard designed to keep them from overlapping with other force fields. In most cases, Glycam’s type names will consist of two characters, one upper-case followed by one lower-case. Because of this, leaprc files, lib files and prep files from previous versions will be incompatible with the current version.

Note that some type names will not change, despite being present in the Glycam force field files. This will be the case where the interface to some other force field is needed, for example when linking to amino acid residues. In these cases, Glycam will use the type name appropriate to the external force field. Parameters will be introduced only to the extent necessary to provide a link. Since the associated parameters will also include Glycam types, they should only affect the intersections between the two force fields.

2.9.3. General information regarding parameter development

In GLYCAM-06,[35] the torsion terms have now been entirely developed by fitting to quantum mechanical data (B3LYP/6-31++G(2d,2p)//HF/6-31G(d)) for small-molecules. This has converted GLYCAM-06 into an additive force field that is extensible to diverse molecular classes including, for example, lipids and glycolipids. The parameters are self-contained, such that it is not necessary to load any AMBER parameter files when modeling carbohydrates or lipids. To maintain orthogonality with AMBER parameters for proteins, notably those involving the CT atom type, tetrahedral carbon atoms in GLYCAM are called Cg (C-GLYCAM, CG in previous releases). Thus, GLYCAM and AMBER may be combined for modeling carbohydrate-protein complexes and glycoproteins. More information on atom type names is available in 2.9.2 . Because the GLYCAM-06 torsion terms were derived by fitting to data for small, often highly symmetric molecules, asymmetric phase shifts were not required in the parameters. This has the significant advantage that it allows one set of torsion terms to be used for both α - and β -carbohydrate anomers regardless of monosaccharide ring size or conformation. A molecular development suite of more than 75 molecules was employed, with a test suite that included carbohydrates and numerous smaller molecular fragments. The GLYCAM-06 force field has been validated against quantum mechanical and experimental properties, including: gas-phase conformational energies, hydrogen bond energies, and vibrational frequencies; solution-phase rotamer populations (from NMR data); and solid-phase vibrational frequencies and crystallographic unit cell dimensions.

2.9.4. Scaling of electrostatic and nonbonded interactions

As in previous versions of GLYCAM,[36] the parameters were derived for use without scaling 1-4 non-bonded and electrostatic interactions. Thus, in *sander*, *pmemd*, and so on, the simulation parameters *scnb* and *scee* should typically be set to unity. We have shown that this is essential in order to properly treat internal hydrogen bonds, particularly those associated with the hydroxymethyl group, and to correctly reproduce the rotamer populations for the C5-C6 bond.[37] Beginning with Amber 11, it is now possible to employ mixed scaling of the *scnb* and *scee* parameters. Anyone wishing to simulate systems containing both carbohydrates and proteins should use the new mixed scaling capability. To do this, any scaling factors that differ from the default must be included in the parameter file. Beginning with the GLYCAM_06g parameter file shipped with Amber 11, these factors are already included. Anyone wishing to employ earlier parameter sets must modify the files.

2.9.5. Development of partial atomic charges

As in previous versions of GLYCAM, the atomic partial charges were determined using the RESP formalism, with a weighting factor of 0.01,[35, 38] from a wavefunction computed at the HF/6-31G(d) level. To reduce artifactual fluctuations in the charges on aliphatic hydrogen atoms, and on the adjacent saturated carbon atoms, charges on aliphatic hydrogens (types HC, H1, H2, and H3) were set to zero while the partial charges were fit to the remaining atoms.[39] It should be noted that aliphatic hydrogen atoms typically carry partial charges that fluctuate around zero when they are included in the RESP fitting, particularly when averaged over con-

2. Specifying a force field

formational ensembles.[35, 40] In order to account for the effects of charge variation associated with exocyclic bond rotation, particularly associated with hydroxyl and hydroxymethyl groups, partial atomic charges for each sugar were determined by averaging RESP charges obtained from 100 conformations selected evenly from 10-50 ns solvated MD simulations of the methyl glycoside of each monosaccharide, thus yielding an ensemble averaged charge set.[35, 40]

2.9.6. Carbohydrate parameters for use with the TIP5P water model

In order to extend GLYCAM to simulations employing the TIP-5P water model, an additional set of carbohydrate parameters, GLYCAM-06EP, has been derived in which lone pairs (or extra points, EPs) have been incorporated on the oxygen atoms.[41] The optimal O-EP distance was located by obtaining the best fit to the HF/6-31g(d) electrostatic potential. In general, the best fit to the quantum potential coincided with a negligible charge on the oxygen nuclear position. The optimal O-EP distance for an sp³ oxygen atom was found to be 0.70 Å; for an sp² oxygen atom a shorter length of 0.3 Å was optimal. When applied to water, this approach to locating the lone pair positions and assigning the partial charges yielded a model that was essentially indistinguishable from TIP-5P. Therefore, we believe this model is well suited for use with TIP-5P.[41] The new files are named 06EP (originally 04EP), as they have been corrected for numerous typographical errors and updated to match current naming and residue structure conventions.

2.9.7. Carbohydrate Naming Convention in GLYCAM

In order to incorporate carbohydrates in a standardized way into modeling programs, as well as to provide a standard for X-ray and NMR protein database files (pdb), we have developed a three-letter code nomenclature. The restriction to three letters is based on standards imposed on protein data bank (PDB) files by the RCSB PDB Advisory Committee (www.rcsb.org/pdb/pdbac.html), and for the practical reason that all modeling and experimental software has been developed to read three-letter codes, primarily for use with protein and nucleic acids.

As a basis for a three-letter PDB code for monosaccharides, we have introduced a one-letter code for monosaccharides (Table 2.4).[42] Where possible, the letter is taken from the first letter of the monosaccharide name. Given the endless variety in monosaccharide derivatives, the limitation of 26 letters ensures that no one-letter (or three-letter) code can be all encompassing. We have therefore allocated single letters firstly to all 5- and 6-carbon, non-derivatized monosaccharides. Subsequently, letters have been assigned on the order of frequency of occurrence or biological significance.

Using three letters (Tables 2.5 to 2.7), the present GLYCAM residue names encode the following content: carbohydrate residue name (Glc, Gal, etc.), ring form (pyranosyl or furanosyl), anomeric configuration (α or β , enantiomeric form (D or L) and occupied linkage positions (2-, 2,3-, 2,4,6-, etc.). Incorporation of linkage position is a particularly useful addition, since, unlike amino acids, the linkage cannot otherwise be inferred from the monosaccharide name. Further, the three-letter codes were chosen to be orthogonal to those currently employed for amino acids.

2.9. The GLYCAM-06 and GLYCAM-06EP force fields for carbohydrates and lipids

Carbohydrate	Pyranose	Furanose
	α/β , D/L	α/β , D/L
Arabinose	yes	yes
Lyxose	yes	yes
Ribose	yes	yes
Xylose	yes	yes
Allose	yes	
Altrose	yes	
Galactose	yes	<i>a</i>
Glucose	yes	<i>a</i>
Gulose	yes	
Idose	<i>a</i>	
Mannose	yes	
Talose	yes	
Fructose	yes	yes
Psicose	yes	yes
Sorbose	yes	yes
Tagatose	yes	yes
Fucose	yes	
Quinovose	yes	
Rhamnose	yes	
Galacturonic Acid	yes	
Glucuronic Acid	yes	
Iduronic Acid	yes	
<i>N</i> -Acetylgalactosamine	yes	
<i>N</i> -Acetylglucosamine	yes	
<i>N</i> -Acetylmannosamine	yes	
Neu5Ac	yes, <i>b</i>	yes, <i>b</i>
KDN	<i>a, b</i>	<i>a, b</i>
KDO	<i>a, b</i>	<i>a, b</i>

Table 2.3.: Current Status of Monosaccharide Availability in GLYCAM. (*a*) Currently under development. (*b*) Only one enantiomer and ring form known.

2. Specifying a force field

	Carbohydrate ^a	One letter code ^b	Common Abbreviation
1	D-Arabinose	A	Ara
2	D-Lyxose	D	Lyx
3	D-Ribose	R	Rib
4	D-Xylose	X	Xyl
5	D-Allose	N	All
6	D-Altrose	E	Alt
7	D-Galactose	L	Gal
8	D-Glucose	G	Glc
9	D-Gulose	K	Gul
10	D-Idose	I	Ido
11	D-Mannose	M	Man
12	D-Talose	T	Tal
13	D-Fructose	C	Fru
14	D-Psicose	P	Psi
15	D-Sorbose	B ^d	Sor
16	D-Tagatose	J	Tag
17	D-Fucose (6-deoxy D-galactose)	F	Fuc
18	D-Quinovose (6-deoxy D-glucose)	Q	Qui
19	D-Rhamnose (6-deoxy D-mannose)	H	Rha
20	D-Galacturonic Acid	O ^d	GalA
21	D-Glucuronic Acid	Z ^d	GlcA
22	D-Iduronic Acid	U ^d	IdoA
23	D-N-Acetylgalactosamine	V ^d	GalNac
24	D-N-Acetylglucosamine	Y ^d	GlcNac
25	D-N-Acetylmannosamine	W ^d	ManNac
26	N-Acetyl-neuraminic Acid	S ^d	NeuNac, Neu5Ac
	KDN	KN ^{c,d}	KDN
	KDO	KO ^{c,d}	KDO
	N-Glycolyl-neuraminic Acid	SG ^{c,d}	NeuNGc, Neu5Gc

Table 2.4.: *The one-letter codes that form the core of the GLYCAM residue names for monosaccharides* ^a*Users requiring prep files for residues not currently available may contact the Woods group (www.glycam.org) to request generation of structures and ensemble averaged charges.* ^b*Lowercase letters indicate L-sugars, thus L-Fucose would be “f”, see Table 2.7 .* ^c*Less common residues that cannot be assigned a single letter code are accommodated at the expense of some information content.* ^d*Nomenclature involving these residues will likely change in future releases.[42] Please visit www.glycam.org for the most updated information.*

2.9. The GLYCAM-06 and GLYCAM-06EP force fields for carbohydrates and lipids

	α -D-Glcp	β -D-Galp	α -D-Arap	β -D-Xylp
Linkage Position	Residue Name	Residue Name	Residue Name	Residue Name
Terminal ^b	0GA ^b	0LB	0AA	0XB
1- ^c	1GA ^c	1LB	1AA	1XB
2-	2GA	2LB	2AA	2XB
3-	3GA	3LB	3AA	3XB
4-	4GA	4LB	4AA	4XB
6-	6GA	6LB		
2,3-	ZGA ^d	ZLB	ZAA	ZXB
2,4-	YGA	YLB	YAA	YXB
2,6-	XGA	XLB		
3,4-	WGA	WLB	WAA	WXB
3,6-	VGA	VLB		
4,6-	UGA	ULB		
2,3,4-	TGA	TLB	TAA	TXB
2,3,6-	SGA	SLB		
2,4,6-	RGA	RLB		
3,4,6-	QGA	QLB		
2,3,4,6-	PGA	PLB		

Table 2.5.: Specification of linkage position and anomeric configuration in D-hexo- and D-pentopyranoses in three-letter codes based on the GLYCAM one-letter code ^aIn pyranoses A signifies α -configuration; B = β . ^bPreviously called GA, the zero prefix indicates that there are no oxygen atoms available for bond formation, i.e., that the residue is for chain termination. ^cIntroduced to facilitate the formation of a 1-1' linkage as in α -D-Glc-1-1'- α -D-Glc {1GA 0GA}. ^dFor linkages involving more than one position, it is necessary to avoid employing prefix letters that would lead to a three-letter code that was already employed for amino acids, such as ALA.

	α -D-Glcf	β -D-Manf	α -D-Araf	β -D-Xylf
Linkage position	Residue name	Residue name	Residue name	Residue name
Terminal	0GD	0MU	0AD	0XU
1-	1GD	1MU	1AD	1XU
2-	2GD	2MU	2AD	2XU
3-	3GD	3MU	3AD	3XU
...
etc.	etc.	etc.	etc.	etc.

Table 2.6.: Specification of linkage position and anomeric configuration in D-hexo- and D-pentofuranoses in three-letter codes based on the GLYCAM one-letter code. In furanoses D (down) signifies α ; U (up) = β .

2. Specifying a force field

	α -L-Glcp	β -L-Manp	α -L-Arap	β -L-Xylp
Linkage position	Residue name	Residue name	Residue name	Residue name
Terminal	0gA	0mB	0aA	0xB
1-	1gA	1mB	1aA	1xB
2-	2gA	2mB	2aA	2xB
3-	3gA	3mB	3aA	3xB
...
etc.	etc.	etc.	etc.	etc.

Table 2.7.: Specification of linkage position and anomeric configuration in L-hexo- and L-pentofuranoses in three-letter codes.

2.10. LIPID11: A modular lipid force field

Relevant files:

```
leaprc.lipid11    loads the files below
lipid11.lib       atoms, charges, and topologies for LIPID11 residues
lipid11.dat       LIPID11 force field parameters
```

Usage:

```
source leaprc.lipid11
```

LIPID11 is a new modular force field for the simulation of phospholipids and cholesterol designed to be compatible with the other pairwise additive Amber force fields. Phospholipids are divided into interchangeable head group and tail group “residues.”

Currently, there are seven tail group residues and eight head group residues supported, as well as cholesterol. LEaP supports any combination of lipid residues. The supported LIPID11 residues and their residue names are listed in the following table:

Table of LIPID11 residue names

	Description	LIPID11 Residue Name
Acyl chain	Palmitoyl (16:0)	PA
	Stearoyl (18:0)	ST
	Oleoyl (18:1 n-9)	OL
	Linoleoyl (18:2 n-6)	LEO
	Linolenoyl (18:3 n-3)	LEN
	Arachidonoyl (20:4 n-6)	AR
	Docosahexanoyl (22:6 n-3))	DHA
Head group	Phosphatidylcholine	PC
	Phosphatidylethanolamine	PE
	Phosphatidylserine	PS
	Phosphatidic acid (PHO4-)	PH-
	Phosphatidic acid (PO42-)	P2-
	R-phosphatidylglycerol	PGR
	S-phosphatidylglycerol	PGS
	Phosphatidylinositol	PI
Other	Cholesterol	CHL

LIPID11 can be used alone or in conjunction with other Amber force fields. For example, to load ff12SB and LIPID11 in LEaP use

```
source leaprc.ff12SB
source leaprc.lipid11
```

A properly formatted lipid PDB can then be loaded into LEaP. Please see the following section about the LIPID11 PDB format. For example, to load a combination protein-lipid system PDB simply use the standard

```
proteinWithDPPC = loadPdb proteinWithDPPC.pdb
```

Additional information:

Further information about the LIPID11 force field can be found in Ref. [9].

2. Specifying a force field

2.10.1. LIPID11 PDB format

Each phospholipid molecule in LIPID11 is made up of three residues. The head group and tail residues are linked together by the LEaP program after loading the lipid PDB file. In order for LEaP to correctly define linker atoms the following residue order must be followed for each molecule:

LIPID11 residue order in each molecule:

```
sn-1 tail residue
head group residue
sn-2 tail residue
TER card
```

The connectivity (CONECT records) section of the PDB is redundant and should be removed prior to loading into LEaP.

charmm lipid2amber.x:

A simple script called *charmm lipid2amber.x* is available to convert a CHARMM-GUI (<http://www.charmm-gui.org/>) membrane builder pdb file to a lipid11 pdb file ready to be loaded in LEaP for Amber simulations.

Usage:

```
charmm lipid2amber.x input_CHARMM-GUI.pdb output_LIPID11.pdb
```

2.11. Ions

<code>frcmod.ionsjc_tip3p</code>	Joung/Cheatham ion parameters for TIP3P water
<code>frcmod.ionsjc_spce</code>	same, but for SPC/E water
<code>frcmod.ionsjc_tip4pew</code>	same, but for TIP4P/EW water
<code>frcmod.ionsff99_tip3p</code>	Older monovalent ion parameters from ff94/ff99
<code>ions08.lib</code>	topologies for ions with the new naming scheme
<code>ions94.lib</code>	topologies for ions with the old naming scheme

In the past, for alkali ions with TIP3P waters, Amber has provided the values of Aqvist,[43] adjusted for Amber’s nonbonded atom pair combining rules to give the same ion-OW potentials as in the original (which were designed for SPC water); these values reproduce the first peak of the radial distribution for ion-OW and the relative free energies of solvation in water of the various ions. Note that these values would have to be changed if a water model other than TIP3P were to be used. Rather arbitrarily, Amber also included chloride parameters from Dang.[44] These are now known not to work all that well with the Aqvist cation parameters, particularly for the K/Cl pair. Specifically, at concentrations above 200 mM, KCl will spontaneously crystallize; this is also seen with NaCl at concentrations above 1 M.[45] These “older” parameters are now collected in *frcmod.ionsff99_tip3p*, but are not recommended except to reproduce older simulations.

Recently, Joung and Cheatham have created a more consistent set of parameters, fitting solvation free energies, radial distribution functions, ion-water interaction energies and crystal lattice

energies and lattice constants for non-polarizable spherical ions.[14, 15] These have been separately parametrized for each of three popular water models, as indicated above. Please note: *Most leaprc files still load the “old” ion parameters*; to use the newer versions, you will need to load the *ions08.lib* file as well as the appropriate *frcmod* file. Even for *ff10*, which automatically loads *ions08.lib*, you will need to choose a *frcmod* file that matches the water model you are using.

2.12. Solvent models

<code>solvents.lib</code>	library for water, methanol, chloroform, NMA, urea
<code>frcmod.tip4p</code>	Parameter changes for TIP4P.
<code>frcmod.tip4pew</code>	Parameter changes for TIP4PEW.
<code>frcmod.tip5p</code>	Parameter changes for TIP5P.
<code>frcmod.spce</code>	Parameter changes for SPC/E.
<code>frcmod.pol3</code>	Parameter changes for POL3.
<code>frcmod.meoh</code>	Parameters for methanol.
<code>frcmod.chcl3</code>	Parameters for chloroform.
<code>frcmod.nma</code>	Parameters for N-methacetamide.
<code>frcmod.urea</code>	Parameters for urea (or urea-water mixtures).

Amber now provides direct support for several water models. The default water model is TIP3P.[46] This model will be used for residues with names HOH or WAT. If you want to use other water models, execute the following leap commands after loading your leaprc file:

```
WAT = PL3 (residues named WAT in pdb file will be POL3)
loadAmberParams frcmod.pol3 (sets the HW,OW parameters to POL3)
```

(The above is obviously for the POL3 model.) The *solvents.lib* file contains TIP3P,[46] TIP3P/F,[47] TIP4P,[46, 48] TIP4P/Ew,[49, 50] TIP5P,[51] POL3[52] and SPC/E[53] models for water; these are called TP3, TPF, TP4, T4E, TP5, PL3 and SPC, respectively. By default, the residue name in the prmtop file will be WAT, regardless of which water model is used. If you want to change this (for example, to keep track of which water model you are using), you can change the residue name to whatever you like. For example,

```
WAT = TP4
set WAT.1 name "TP4"
```

would make a special label in PDB and prmtop files for TIP4P water. Note that Brookhaven format files allow at most three characters for the residue label, which is why the residue names above have to be abbreviated.

Amber has two flexible water models, one for classical dynamics, SPC/Fw[54] (called “SPF”) and one for path-integral MD, qSPC/Fw[55] (called “SPG”). You would use these in the following manner:

```
WAT = SPG
loadAmberParams frcmod.qspcfw
set default FlexibleWater on
```

2. Specifying a force field

Then, when you load a PDB file with residues called WAT, they will get the parameters for qSPC/Fw. (Obviously, you need to run some version of quantum dynamics if you are using qSPC/Fw water.)

The *solvents.lib* file, which is automatically loaded with many leaprc files, also contains pre-equilibrated boxes for many of these water models. These are called POL3BOX, QSPCFWBOX, SPCBOX, SPCFWBOX, TIP3PBOX, TIP3PFBOX, TIP4PBOX, TIP4PEWBOX, and TIP5PBOX. These can be used as arguments to the *solvateBox* or *solvateOct* commands in LEaP.

In addition, non-polarizable models for the organic solvents methanol, chloroform and N-methylacetamide are provided,[29] along with a box for an 8M urea-water mixture. The input files for a single molecule are in *\$AMBERHOME/dat/leap/prep*, and the corresponding frcmod files are in *\$AMBERHOME/dat/leap/parm*. Pre-equilibrated boxes are in *\$AMBERHOME/dat/leap/lib*. For example, to solvate a simple peptide in methanol, you could do the following:

```
source leaprc.ff99SB (get a standard force field)
loadAmberParams frcmod.meoh (get methanol parameters)
peptide = sequence { ACE VAL NME } (construct a simple peptide)
solvateBox peptide MEOHBOX 12.0 0.8 (solvate the peptide with meoh)
saveAmberParm peptide prmtop prmcrd
quit
```

Similar commands will work for other solvent models.

2.13. CHAMBER

CHAMBER (CHARMM↔AMBER) is a tool which enables the use of the CHARMM force field within AMBER's molecular dynamics engines (MDEs). If you make use of this tool, please cite the following [56]. There are two components to CHAMBER:

1. The tool (*\$AMBERHOME/bin/chamber*) which converts a CHARMM psf, associated coordinated file, parameter and topology to a CHARMM force field enabled version of AMBER's prmtop and inpcrd.
2. The additional code within *sander* and *pmemd* to evaluate the extra CHARMM energies and forces.

AMBER[57] and CHARMM[58, 59] are two approaches to the parametrization of classical force fields that find extensive use in the modeling of biological systems. The high similarity in the functional form of the two potential energy functions used by these force fields, Eq.(2.1 and 2.2), gives rise to the possible use of one force field within the other MDE.

$$\begin{aligned} V_{\text{AMBER}} = & \sum_{\text{bonds}} k(r - r_{eq})^2 + \sum_{\text{angles}} k(\theta - \theta_{eq})^2 + \sum_{\text{dihedrals}} \frac{V_n}{2} [1 + \cos(n\phi - \gamma)] / \\ & + \sum_{i < j} \left[\frac{A_{ij}}{R_{ij}^{12}} - \frac{B_{ij}}{R_{ij}^6} \right] + \sum_{i < j} \left[\frac{q_i q_j}{\epsilon R_{ij}} \right] \end{aligned} \quad (2.1)$$

$$\begin{aligned}
V_{\text{CHARMM}} = & \sum_{\text{bonds}} k_b (b - b_0)^2 + \sum_{\text{angles}} k_\theta (\theta - \theta_0)^2 + \sum_{\text{dihedrals}} k_\phi [1 + \cos(n\phi - \delta)] \\
& + \sum_{\text{Urey-Bradley}} k_u (u - u_0)^2 + \sum_{\text{impropers}} k (\omega - \omega_0)^2 + \sum_{\phi, \psi} V_{\text{CMAP}} \\
& + \sum_{\text{nonbonded}} \epsilon \left[\left(\frac{R_{\text{min}ij}}{r_{ij}} \right)^{12} - \left(\frac{R_{\text{min}ij}}{r_{ij}} \right)^6 \right] + \frac{q_i q_j}{\epsilon r_{ij}} \quad (2.2)
\end{aligned}$$

In the case of the CHARMM force field, its MDE is also called CHARMM[60, 61]. For the implementation of the CHARMM force field within Amber, parameters that are of the same energy term can be directly translated. However, there are differences in the functional forms of the two potentials, with CHARMM having three additional bonded terms. With respect to the 1-4 non-bonded interactions, CHARMM scales these in a different manner: the electrostatic scaling factor (*scee*) is 1.0 in CHARMM but 1.2 in Amber, while the van der Waals scaling factor (*scnb*) is 1.0 within CHARMM but 2.0 in Amber. Additionally, CHARMM uses a different set of parameters in the Lennard-Jones equation for the van der Waals interaction if the two atoms are bonded 1-4 to each other.

The first additional bonded term is CHARMM’s two-body Urey-Bradley term, which extends over all 1-3 bonds. The second is a four-body quadratic improper term. The final additional term is a cross term, named CMAP, [62, 63], which is a function of two sequential protein backbone dihedrals. This term originates from differences observed between classically calculated two-dimensional ϕ/ψ peptide free energy surfaces using the CHARMM22 force field and those of experiment. CMAP is a numerical energy correction which essentially transforms the 2D ϕ/ψ classical energy map to match that of a QM calculated map.

Support for these extra terms has required the development of extra sections to Amber’s extensible prmtop format to accommodate this new information as well as modifications of the precision of existing sections. For example, the CHARMM parameter file stores the equilibrium angle (θ_0 , Eq.2.2) parameter in degrees in its parameter file, while Amber stores it in radians in the prmtop. However, during the conversion with *chamber*, this becomes inexact when converted to radians. Within CHARMM this is done internally at runtime and the inexactness is determined by the variable type that will hold the result of this conversion. However, for Amber, this conversion is done at the *chamber* execution stage, and as a result is limited by the precision to which that specific parameter is written to the prmtop file. Hence the precision of the ANGLE_EQUIL_VALUE has been increased; similar changes were carried out for the CHARGE and VDW sections for the same reasons. Specifically, the modified sections of the prmtop format and the additions to it are as follows:

```
%FLAG CTITLE
```

The keyword CTITLE is used in place of TITLE to specify that this is a CHAMBER prmtop.

```
%FLAG FORCE FIELD TYPE
```

```
%FORMAT(i2,a78)
```

```
1 CHARMM 31 *>>>>>>>CHARMM22 All-Hydrogen Topology File for Proteins <<
```

This section described the force field in use. The initial integer specifies the number of lines to be read. The keyword CHARMM here indicates that this is the CHARMM force field.

2. Specifying a force field

```
%FLAG CHARGE
%COMMENT Atomic charge multiplied by sqrt(332.0716D0) (CCELEC)
%FORMAT(3e24.16)
```

The default format for charge has been changed from 5e16.8 to 3e24.16

```
%FLAG CHARMM_UREY_BRADLEY_COUNT
%COMMENT V(ub) = K_ub(r_ik - R_ub)**2
%COMMENT Number of Urey Bradley terms and types
%FORMAT(2i8)
```

This additional section describes the number of CHARMM Urey-Bradley terms present and the total number of Urey-Bradley types in use.

```
%FLAG CHARMM_UREY_BRADLEY
%COMMENT List of the two atoms and its parameter index
%COMMENT in each UB term: i,k,index
%FORMAT(10i8)
```

This additional section lists the atom indexes and parameter lookup index for each of the Urey-Bradley terms.

```
%FLAG CHARMM_UREY_BRADLEY_FORCE_CONSTANT
%COMMENT K_ub: kcal/mole/A**2
%FORMAT(5e16.8)
```

This additional section lists the force constant for each of the Urey-Bradley types.

```
%FLAG CHARMM_UREY_BRADLEY_EQUIL_VALUE
%COMMENT r_ub: A
%FORMAT(5e16.8)
```

This additional section lists the equilibrium value for each of the Urey-Bradley types.

```
%FLAG SCEE_SCALE_FACTOR
%FORMAT(5e16.8)
```

This additional section lists a unique value of scee for each dihedral. This overrides the default or &cntrl values set for scee and in the case of the CHARMM force field will always be 1.0 for all dihedrals.

```
%FLAG SCNB_SCALE_FACTOR
%FORMAT(5e16.8)
```

This is the analogous additional term for scnb

```
%FLAG CHARMM_NUM_IMPROPERS
%COMMENT Number of terms contributing to the
%COMMENT quadratic four atom improper energy term:
%COMMENT V(improper) = K_psi(psi - psi_0)**2
%FORMAT(10i8)
```

This additional section lists the number of CHARMM improper terms present.

```
%FLAG CHARMM_IMPROPERS
%COMMENT List of the four atoms in each improper term
%COMMENT i,j,k,l,index i,j,k,l,index
%COMMENT where index is into the following two lists:
%COMMENT CHARMM_IMPROPER_{FORCE_CONSTANT,IMPROPER_PHASE}
%FORMAT(10i8)
```

This additional section lists the atom indices and index into the parameter arrays for each of the CHARMM improper terms.

```
%FLAG CHARMM_NUM_IMPR_TYPES
%COMMENT Number of unique parameters contributing to the
%COMMENT quadratic four atom improper energy term
%FORMAT(i8)
```

This additional section lists the number of types present for the CHARMM impropers.

```
%FLAG CHARMM_IMPROPER_FORCE_CONSTANT
%COMMENT K_psi: kcal/mole/rad**2
%FORMAT(5e16.8)
```

This additional section lists the force constant for each CHARMM improper types.

```
%FLAG CHARMM_IMPROPER_PHASE
%COMMENT psi: degrees
%FORMAT(5e16.8)
```

This additional section lists the equilibrium phase angle for each of the CHARMM improper types.

```
%FLAG LENNARD_JONES_ACOEF
%FORMAT(3e24.16)
```

The default format for the Lennard Jones A and B coefficients has been changed from 5e16.8 to 3e24.16.

```
%FLAG LENNARD_JONES_14_ACOEF
%FORMAT(3e24.16)
```

This additional section and the corresponding BCOEF section provide the alternative parameters for 1-4 VDW interactions in the CHARMM force field.

In concert with these prmtop additions, the appropriate modifications have to be made within *sander* and *pmemd* to enable the calculation of the energy and derivatives corresponding to these new terms. The intention behind the approach of creating a CHARMM enabled prmtop file is that the use of this prmtop file should be transparent to the user. Once a CHARMM prmtop file is produced by *chamber*, the *sander* and *pmemd* dynamics engines automatically detect the presence of CHARMM parameters in the prmtop file and automatically select the correct parameters and code paths.

2. Specifying a force field

WARNING: The use of an unpatched Amber molecular dynamics engine with a chamber-generated prmtop file will give undefined behavior, leading to incorrect results. If you see the following error at runtime:

```
ERROR: Flag "TITLE" not found in PARM file
```

it most likely means that you are using an old pmemd or sander executable.

A difficulty that has been encountered with the chamber generated prmtop files is visualisation with VMD. The format of the chamber generated prmtop is valid with respect to AMBER's prmtop %FLAG, %FORMAT paradigm, however, VMD does not take into account a flag's corresponding format specification since it has, a priori, set each flag to a specific format. Hence, when the format of an existing flag is modified in a prmtop, VMD fails to recognise this and incorrectly uses its hardcoded value instead.

As of AT 1.6, chamber has the ability to write an additional version of the prmtop (vmd_prmtop) file, that is compatible with VMD. The general strategy here, is to use this additional vmd_prmtop file only for viewing purposes with VMD, and use the correct prmtop for calculations with SANDER and PMEMD. The compatible vmd_prmtop file is correct with respect to topology, but an incorrect with respect to certain parameters; for example %CHARGE has been truncated to the old format and %COMMENT has been removed.

If one specifies the -vmd flag, an additional prmtop file, named vmd_prmtop, is generated. This can then be used with VMD in the following ways:

```
vmd -parm7 vmd_prmtop -rst7 file.inpcrd
vmd -parm7 vmd_prmtop -mdcrd trajectory.mdcrd
vmd -parm7 vmd_prmtop -netcdf trajectory.nc
```

2.13.1. Usage

Here is the set of options returned from running the chamber binary:

```
Usage: chamber [args]
args for input are <default>
    -top <top_all127_prot_na.rtf>
    -param <par_all127_prot_na.prm>
    -psf <psf.psf>
    -crd <chmpdb.pdb>
Note: -crd can specify a pdb, a CHARMM crd or CHARMM rst file.
The filetype is auto detected.
args for output are      <default>
    -p      <prmtop>
    -inpcrd <inpcrd>
args for options are:
    -cmap / -nocmap (Required option. Specifies
                     whether CMAP terms should be
```



```

                                included or excluded.)
-tip3_flex (allow angle in water)
-box a b c
    Set the Orthorhombic lattice parameters a b c
    for the generated inpcrd file.
-verbose    (lots of progress messages)
-vmd
    Write a VMD compatible form of the prmtop file
-radius_set (GB radius set) options are: <default>
    0 Bondi radii (bondi)
    1 Amber 6 modified Bondi radii (amber6)
    <2> modified Bondi radii (mbondi)
    6 H(N)-modified Bondi radii (mbondi2)
arg for help (this message) -h

```

Typical usage would be as follows:

```

$AMBERHOME/bin/chamber -cmap -top top_all122_prot.inp \
-param par_all122_prot.inp -psf foo.psf -crd foo.coor \
-p foo.prmtop -inpcrd foo.inpcrd -box 48.37 40.15 35.21

```

2.13.2. Validation

A force field is defined by its specific potential energy equation and its specific set of associated parameters; it is independent of the MDE that it is expressed in. For a faithful reproduction of a force field that exists in a reference MDE, one needs to be able to reproduce the following in another engine to within a specific precision:

1. The same total potential energy of the system.
2. The same energy gradients on each atom in the system.

However, as soon as dynamics are explored using a force field, external attributes such as thermostat, long range electrostatic treatment and cutoffs come into play and are specific to the MDE; these are considered outside of the definition of a force field and more closely linked to the type of simulation being run and the MDE.

Starting with version c36a2 of CHARMM, a command (**fredump**) has been implemented which provides a validation route for alternate implementations of the CHARMM force fields. For a given system, this command writes the various force field potential energy contributions, as well as the energy gradient experienced by each atom, to a file using a specific format and to a high precision. The same formatted output can also be generated by the AMBER MDEs to facilitate comparison and to validate that the CHARMM force field is being implemented correctly in Amber's MDEs.

An example section of a charmm script that will write this output to a file called **charmm_gold_c36a2** is as follows:

2. Specifying a force field

```
open unit 20 form write name charmm_gold_c36a2
frcdump unit 20
close unit 20
```

The analogous mdin section for Amber is as follows:

```
&debugf
  do_charmm_dump_gold = 1,
/
```

Given this directive, the Amber MDE will stop after evaluating the potential energy of a system and write the energy and forces pertaining to this to a (hardcoded) file called **charmm_gold** in the same directory as the mdin file. The reader is invited to examine the various example test calculations within the `$AMBERHOME/test/chamber/dev_tests/` directory for in depth examples of the above. For such testing, it is recommended that both the CHARMM binary and the Amber MDE binaries be compiled with the same compiler. Given that CHARMM support within Amber and the *chamber* software is still somewhat experimental, the user is advised to carry out such a comparison before running a long production run.

2.13.3. Known limitations / Issues

This is a non-exhaustive list of the current known bugs and/or limitations with *chamber*:

- CHARMM polarization models are not supported. (**IPOL** /= 0)
- The ability to read CHARMM restart files is not currently supported.
- The mdout file will contain extra potential energy fields pertaining to the CHARMM terms. This may break or confuse third party scripts that parse such outputs.
- Third party scripts and/or tools which do not correctly parse the extensible prmtop format may have issues with a *chamber*-generated prmtop file.
- The potential energy decomposition components (self, reciprocal, direct, adjusted) of the Particle Mesh Ewald energy generated in the **charmm_gold** file when the **do_charmm_dump_gold = 1** mdin option in Amber do not match with the breakdown used in CHARMM, however, the summation and resulting forces do match.

If other issues are found, the *chamber* authors would be very grateful if these could be reported to them, either via the Amber mailing list and/or directly to the authors. Please ensure that prior to reporting an issue, the *chamber* binary passes the test cases provided with AmberTools. Please provide a standalone example of the problem with all input files present and a script reproducing the sequence of commands that triggers the problem. The posting of large files (> 2 MB) to the Amber mailing list is not recommended; instead one should make the files available on a website somewhere and provide a link to it with the posting to the list.

3. LEaP

3.1. Introduction

LEaP is the basic tool to construct force field files (see Fig. 1.1). Using *tleap*, the user can:

```
Read AMBER PREP input files
Read Amber PARM format parameter sets
Read and write Object File Format files (OFF)
Read and write PDB files
Construct new residues and molecules using simple commands
Link together residues and create nonbonded complexes of molecules
Modify internal coordinates within a molecule
Generate files that contain topology and parameters for AMBER and NAB
usage: tleap [ -I<dir> ] [ -f <file>|- ]
```

The command *tleap* is a simple shell script that calls *teLeap* with a number of standard arguments. Directories to be searched are indicated by one or more “-I” flags; standard locations are provided in the *tleap* script. The “-f” flag is used to tell *tleap* to take its input from a file (or from *stdin* if “-f -” is specified). If there is no “-f” flag, input is taken interactively from the terminal.

A key command for LEaP is *loadPdb*, which inputs sequence and structure information from Protein Databank Files. *Be sure to read Section 5.8 for information on how to “clean up” PDB files before loading them.*

3.2. Concepts

In order to effectively use LEaP it is necessary to understand the philosophy behind the program, especially the concepts of LEaP commands, variables, and objects. In addition to exploring these concepts, this section also addresses the use of external files and libraries with the program.

3.2.1. Commands

A researcher uses LEaP by entering commands that manipulate objects. An object is just a basic building block; some examples of objects are ATOMs, RESIDUEs, UNITs, and PARM-SETs. The commands that are supported within LEaP are described throughout the manual and are defined in detail in the “Command Reference” section.

3. LEaP

The heart of LEaP is a command-line interface that accepts text commands which direct the program to perform operations on objects. All LEaP commands have one of the following two forms:

```
command argument1 argument2 argument3 ...  
variable = command argument1 argument2 ...
```

For example:

```
edit ALA trypsin = loadPdb trypsin.pdb
```

Each command is followed by zero or more arguments that are separated by whitespace. Some commands return objects which are then associated with a variable using an assignment (=) statement. Each command acts upon its arguments, and some of the commands modify their arguments' contents. The commands themselves are case-insensitive. That is, in the above example, `edit` could have been entered as `Edit`, `eDiT`, or any combination of upper and lower case characters. Similarly, `loadPdb` could have been entered a number of different ways, including `loadpdb`. In this manual, we frequently use a mixed case for commands. We do this to enhance the differences between commands and as a mnemonic device. Thus, while we write `createAtom`, `createResidue`, and `createUnit` in the manual, the user can use any case when entering these commands into the program.

The arguments in the command text may be objects such as `NUMBERS`s, `STRING`s, or `LIST`s, or they may be variables. These two subjects are discussed next.

3.2.2. Variables

A variable is a handle for accessing an object. A variable name can be any alphanumeric string whose first character is an alphabetic character. Alphanumeric means that the characters of the name may be letters, numbers, or special symbols such as “*”. The following special symbols should not be used in variable names: dollar sign, comma, period (full stop), pound sign (hash), equals sign, space, semicolon, double quote, or the curly braces { and }. LEaP commands should not be used as variable names. Unlike commands, variable names are case-sensitive: “ARG” and “arg” are different variables. Variables are associated with objects using an assignment statement not unlike that found in conventional programming languages such as Fortran or C.

```
mole = 6.02E23  
MOLE = 6.02E23  
myName = "Joe Smith"  
listOf7Numbers = { 1.2 2.3 3.4 4.5 6 7 8 }
```

In the above examples, both `mole` and `MOLE` are variable names, whose contents are the same (6.02×10^{23}). Despite the fact that both `mole` and `MOLE` have the same contents, they are not the same variable. This is due to the fact that variable names are case-sensitive. LEaP maintains a list of variables that are currently defined. This list can be displayed using the `list` command. The contents of a variable can be printed using the `desc` command.

3.2.3. Objects

The object is the fundamental entity in LEaP. Objects range from the simple, such as NUMBERS and STRINGS, to the complex, such as UNITS, RESIDUES and ATOMs. Complex objects have properties that can be altered using the `set` command, and some complex objects can contain other objects. For example, RESIDUES are complex objects that can contain ATOMs and have the properties: residue name, connect atoms, and residue type.

NUMBERS

NUMBERS are simple objects holding double-precision floating point numbers. They serve the same function as “double precision” variables in Fortran and “double” variables in C.

STRINGS

STRINGS are simple objects that are identical to character arrays in C and similar to character strings in Fortran. STRINGS store sequences of characters which may be delimited by double quote characters. Example strings are:

```
"Hello there"
"String with a " (quote) character"
"Strings contain letters and numbers:1231232"
```

LISTs

LISTs are made up of sequences of other objects delimited by LIST open and close characters. The LIST open character is an open curly bracket ({) and the LIST close character is a close curly bracket (}). LISTs can contain other LISTs and be nested arbitrarily deep. Example LISTs are:

```
{ 1 2 3 4 }
{ 1.2 "string" }
{ 1 2 3 { 1 2 } { 3 4 } }
```

LISTs are used by many commands to provide a more flexible way of passing data to the commands. The `zMatrix` command has two arguments, one of which is a LIST of LISTs where each subLIST contains between three and eight objects.

PARMSETs (Parameter Sets)

PARMSETs are objects that contain bond, angle, torsion, and non-bonding parameters for AMBER force field calculations. They are normally loaded from force field data files, such as *parm94.dat*, and *frmod* files.

3. LEaP

ATOMs

ATOMs are complex objects that do not contain any other objects. The ATOM object corresponds to the chemical concept of an atom. Thus, it is a single entity that may be bonded to other ATOMs and used as a building block for creating molecules. ATOMs have many properties that can be changed using the `set` command. These properties are defined below.

name This is a case-sensitive STRING property and it is the ATOM's name. The names for all ATOMs in a RESIDUE should be unique. The name has no relevance to molecular mechanics force field parameters; it is chosen arbitrarily as a means to identify ATOMs. Ideally, the name should correspond to the PDB standard, being 3 characters long except for hydrogens, which can have an extra digit as a 4th character.

type This is a STRING property. It defines the AMBER force field atom type. It is important that the character case match the canonical type definition used in the appropriate force field data (*.dat) or frcmod file. For smooth operation, all atom types must have element and hybridization defined by the `addAtomTypes` command. The standard AMBER force field atom types are added by the selected leaprc file.

charge The charge property is a NUMBER that represents the ATOM's electrostatic point charge to be used in a molecular mechanics force field.

element The atomic element provides a simpler description of the atom than the type, and is used only for LEaP's internal purposes (typically when force field information is not available). The element names correspond to standard nomenclature; the character "?" is used for special cases.

position This property is a LIST of NUMBERS. The LIST must contain three values: the (X, Y, Z) Cartesian coordinates of the ATOM.

RESIDUES

RESIDUES are complex objects that contain ATOMs. RESIDUES are collections of ATOMs, and are either molecules (e.g., formaldehyde) or are linked together to form molecules (e.g., amino acid monomers). RESIDUES have several properties that can be changed using the `set` command. (Note that database RESIDUES are each contained within a UNIT having the same name; the residue GLY is referred to as GLY.1 when setting properties. When two of these single-UNIT residues are joined, the result is a single UNIT containing the two RESIDUES.)

One property of RESIDUES is connection ATOMs. Connection ATOMs are ATOMs that are used to make linkages between RESIDUES. For example, in order to create a protein, the N-terminus of one amino acid residue must be linked to the C-terminus of the next residue. This linkage can be made within LEaP by setting the N ATOM to be a connection ATOM at the N-terminus and the C ATOM to be a connection ATOM at the C-terminus. As another example, two CYX amino acid residues may form a disulfide bridge by crosslinking a connection atom on each residue.

There are several properties of RESIDUES that can be modified using the `set` command. The properties are described below:

connect0 This defines the first of up to three ATOMs that are used to make links to other RESIDUEs. In UNITs containing single RESIDUEs, the RESIDUE's connect0 ATOM is usually defined as the UNIT's head ATOM. (This is how the standard library UNITs are defined.) For amino acids, the convention is to make the N-terminal nitrogen the connect0 ATOM.

connect1 This defines the second of up to three ATOMs that are used to make links to other RESIDUEs. In UNITs containing single RESIDUEs, the RESIDUE's connect1 ATOM is usually defined as the UNIT's tail ATOM. (This is done in the standard library UNITs.) For amino acids, the convention is to make the C-terminal oxygen the connect1 ATOM.

connect2 This defines the third of up to three ATOMs that are used to make links to other RESIDUEs. In amino acids, the convention is that this is the ATOM to which disulfide bridges are made.

restype This property is a STRING that represents the type of the RESIDUE. Currently, it can have one of the following values: "undefined", "solvent", "protein", "nucleic", or "saccharide". Some of the LEaP commands behave in different ways depending on the type of a residue. For example, the solvate commands require that the solvent residues be of type "solvent". It is important that the proper character case be used when defining this property.

name The RESIDUE name is a STRING property. It is important that the proper character case be used when defining this property.

UNITs

UNITs are the most complex objects within LEaP, and the most important. They may contain RESIDUEs and ATOMs. UNITs, when paired with one or more PARMSETs, contain all of the information required to perform a calculation using AMBER. UNITs can be created using the `createUnit` command. RESIDUEs and ATOMs can be added or deleted from a UNIT using the `add` and `remove` commands. UNITs have the following properties, which can be changed using the `set` command:

head

tail These define the ATOMs within the UNIT that are connected when UNITs are joined together using the `sequence` command or when UNITs are joined together with the PDB or PREP file reading commands. The tail ATOM of one UNIT is connected to the head ATOM of the next UNIT in any sequence. (Note: a TER card in a PDB file causes a new UNIT to be started.)

box This property can either be null, a NUMBER, or a LIST. The property defines the bounding box of the UNIT. If it is defined as null then no bounding box is defined. If the value is a single NUMBER, the bounding box will be defined to be a cube with each side being *box* Å across. If the value is a LIST, it must contain three NUMBERS, the lengths of the three sides of the bounding box.

3. LEaP

cap This property can either be null or a LIST. The property defines the solvent cap of the UNIT. If it is defined as null, no solvent cap is defined. If it is a LIST, it must contain four NUMBERS. The first three define the Cartesian coordinates (X, Y, Z) of the origin of the solvent cap in Å, while the fourth defines the radius of the solvent cap, also in Å.

Examples of setting the above properties are:

```
set dipeptide head dipeptide.1.N
set dipeptide box { 5.0 10.0 15.0 }
set dipeptide cap { 15.0 10.0 5.0 8.0 }
```

The first example makes the amide nitrogen in the first RESIDUE within “dipeptide” the head ATOM. The second example places a rectangular bounding box around the origin with the (X, Y, Z) dimensions of (5.0, 10.0, 15.0) in Å. The third example defines a solvent cap centered at (15.0, 10.0, 5.0) Å with a radius of 8.0 Å. Note: the `set cap` command does not actually solvate, it just sets an attribute. See the `solvateCap` command for a more practical case.

Complex objects and accessing subobjects

UNITs and RESIDUEs are complex objects. Among other things, this means that they can contain other objects. There is a loose hierarchy of complex objects and what they are allowed to contain. The hierarchy is as follows:

- UNITs can contain RESIDUEs and ATOMs.
- RESIDUEs can contain ATOMs.

The hierarchy is loose because it does not forbid UNITs from containing ATOMs directly. However, the convention that has evolved within LEaP is to have UNITs directly contain RESIDUEs which directly contain ATOMs.

Objects that are contained within other objects can be accessed using dot “.” notation. An example would be a UNIT which describes a dipeptide ALA-PHE. The UNIT contains two RESIDUEs each of which contain several ATOMs. If the UNIT is referenced (named) by the variable `dipeptide`, then the RESIDUE named ALA can be accessed in two ways. The user may type one of the following commands to display the contents of the RESIDUE:

```
desc dipeptide.ALA
desc dipeptide.1
```

The first command translates to “describe some RESIDUE named ALA within the UNIT named dipeptide”. The second form translates as “describe the RESIDUE with sequence number 1 within the UNIT named dipeptide”. The second form is more useful because every subobject within an object is guaranteed to have a unique sequence number. If the first form is used and there is more than one RESIDUE with the name ALA, then an arbitrary residue with the name ALA is returned. To access ATOMs within RESIDUEs, either of the following forms of command may be used:

```
desc dipeptide.1.CA
desc dipeptide.1.3
```


Assuming that the ATOM with the name CA has a sequence number 3 within RESIDUE 1, then both of the above commands will print a description of the α -carbon of RESIDUE dipeptide.ALAL or dipeptide.1. The reader should keep in mind that dipeptide.1.CA is the ATOM, an object, contained within the RESIDUE named ALA within the variable dipeptide. This means that dipeptide.1.CA can be used as an argument to any command that requires an ATOM as an argument. However dipeptide.1.CA is not a variable and cannot be used on the left hand side of an assignment statement.

3.3. Basic instructions for using LEaP

This section gives an overview of how LEaP is most commonly used. Detailed descriptions of all the commands are given in the following section.

3.3.1. Building a Molecule For Molecular Mechanics

In order to prepare a molecule within LEaP for AMBER, three basic tasks need to be completed.

1. Any needed UNIT or PARMSET objects must be loaded;
2. The molecule must be constructed within LEaP;
3. The user must output topology and coordinate files from LEaP to use in AMBER.

The most typical command sequence is the following:

```
source leaprc.ff99SB (load a force field)
x = loadPdb trypsin.pdb (load in a structure)
... add in cross-links, solvate, etc.
saveAmberParm x prmtop prmcrd (save files)
```

There are a number of variants of this:

1. Although `loadPdb` is by far the most common way to enter a structure, one might use `loadOff`, or `loadAmberPrep`, or use the `zmat` command to build a molecule from a Z-matrix. See the Commands section below for descriptions of these options. If you do not have a starting structure (in the form of a PDB file), LEaP can be used to build the molecule; you will find, however, that this is not always a straightforward process. Many experienced Amber users turn to other (commercial and non-commercial) programs to create their initial structures.
2. Be very attentive to any errors produced in the `loadPdb` step; these generally mean that LEaP has misread the file. A general rule of thumb is to keep editing your input PDB file until LEaP stops complaining. It is often convenient to use the `addPdbAtomMap` or `addPdbResMap` commands to make systematic changes from the names in your PDB files to those in the Amber topology files; see the `leaprc` files in `$AMBERHOME/dat/leap/cmd` for examples of this. *Be sure to read Section 5.8 for information on how to “clean up” PDB files before loading them.*

3. LEaP

3. The `saveAmberParm` command cited above is appropriate for most force fields; for polarizable calculations you will need to use `saveAmberParmPol`.

3.3.2. Amino Acid Residues

For each of the amino acids found in the LEaP libraries, there has been created an N-terminal and a C-terminal analog. The N-terminal amino acid UNIT/RESIDUE names and aliases are prefaced by the letter N (e.g., NALA) and the C-terminal amino acids by the letter C (e.g., CALA). If the user models a peptide or protein within LEaP, they may choose one of three ways to represent the terminal amino acids. The user may use (1) standard amino acids, (2) protecting groups (ACE/NME), or (3) the charged C- and N-terminal amino acid UNITS/RESIDUES. If the standard amino acids are used for the terminal residues, then these residues will have incomplete valences. These three options are illustrated below:

```
{ ALA VAL SER PHE }  
{ ACE ALA VAL SER PHE NME }  
{ NALA VAL SER CPHE }
```

The default for loading from PDB files is to use N- and C-terminal residues; this is established by the `addPdbResMap` command in the default `leaprc` files. To force incomplete valences with the standard residues, one would have to define a sequence (“`x = { ALA VAL SER PHE }`”) and use `loadPdbUsingSeq`, or use `clearPdbResMap` to completely remove the mapping feature.

Histidine can exist either as the protonated species or as a neutral species with a hydrogen at the δ or ϵ position. For this reason, the histidine UNIT/RESIDUE name is either HIP, HID, or HIE (but not HIS). The default “`leaprc`” file assigns the name HIS to HIE. Thus, if a PDB file is read that contains the residue HIS, the residue will be assigned to the HIE UNIT object. This feature can be changed within one’s own `leaprc` file.

The AMBER force fields also differentiate between the residue cysteine (CYS) and the similar residue which participates in disulfide bridges, cystine (CYX). The user will have to explicitly define, using the `bond` command, the disulfide bond for a pair of cystines, as this information is not read from the PDB file. In addition, the user will need to load the PDB file using the `loadPdbUsingSeq` command, substituting CYX for CYS in the sequence wherever a disulfide bond will be created.

3.3.3. Nucleic Acid Residues

The “D” prefix can be used to distinguish between deoxyribose and ribose units. Residue names like “A” or “DA” can be followed by a “5” or “3” (“DA5”, “DA3”) for residues at the ends of chains; this is also the default established by `addPdbResMap`, even if the “5” or “3” are not added in the PDB file. The “5” and “3” residues are “capped” by a hydrogen; the plain and “3” residues include a “leading” phosphate group. Neutral residues (nucleotides) capped by hydrogens end their names with “N”, as in “DAN”.

3.4. Commands

The following is a description of the commands that can be accessed using the command line interface in *tleap*, or through the command line editor in *xleap*. Whenever an argument in a command line definition is enclosed in square brackets (e.g., [arg]), then that argument is optional. When examples are shown, the command line is prefaced by “>”, and the program output is shown without this character preface.

Some commands that are almost never used have been removed from this description to save space. You can use the “help” facility to obtain information about these commands; most only make sense if you understand what the program is doing behind the scenes.

3.4.1. add

```
add a b
```

UNIT/RESIDUE/ATOM a,b

Add the object b to the object a. This command is used to place ATOMs within RESIDUEs, and RESIDUEs within UNITs. This command will work only if b is not contained by any other object.

The following example illustrates both the `add` command and the way the TIP3P water molecule is created for the LEaP distribution.

```
> h1 = createAtom H1 HW 0.417
> h2 = createAtom H2 HW 0.417
> o = createAtom O OW -0.834
>
> set h1 element H
> set h2 element H
> set o element O
>
> r = createResidue TIP3
> add r h1
> add r h2
> add r o
>
> bond h1 o
> bond h2 o
> bond h1 h2
>
> TIP3 = createUnit TIP3
>
> add TIP3 r
> set TIP3.1 retype solvent
> set TIP3.1 imagingAtom TIP3.1.O
>
> zMatrix TIP3 {
```

3. LEaP

```
> { H1 O 0.9572 }
> { H2 O H1 0.9572 104.52 }
> }
>
> saveOff TIP3 water.lib
Saving TIP3.
Building topology.
Building atom parameters.
```

3.4.2. addAtomTypes

```
addAtomTypes { { type element hybrid } { ... } ... }
```

Define element and hybridization for force field atom types. This command for the standard force fields can be seen in the default leaprc files. The STRINGS are most safely rendered using quotation marks. If atom types are not defined, confusing messages about hybridization can result when loading PDB files.

3.4.3. addIons

```
addIons unit ion1 numIon1 [ion2 numIon2]
```

Adds counterions in a shell around *unit* using a Coulombic potential on a grid. If *numIon1* is 0, then the unit is neutralized. In this case, *numIon1* must be opposite in charge to *unit* and *numIon2* must not be specified. If solvent is present, it is ignored in the charge and steric calculations, and if an ion has a steric conflict with a solvent molecule, the ion is moved to the center of that solvent molecule, and the latter is deleted. (To avoid this behavior, either solvate *_after_ addions*, or use *addIons2*.) Ions must be monatomic. This procedure is not guaranteed to globally minimize the electrostatic energy. When neutralizing regular-backbone nucleic acids, the first cations will generally be placed between phosphates, leaving the final two ions to be placed somewhere around the middle of the molecule. The default grid resolution is 1 Å, extending from an inner radius of (*maxIonVdwRadius* + *maxSoluteAtomVdwRadius*) to an outer radius 4 Å beyond. A distance-dependent dielectric is used for speed.

3.4.4. addIons2

```
addIons2 unit ion1 numIon1 [ion2 numIon2]
```

Same as *addIons*, except solvent and solute are treated the same.

3.4.5. addPath

```
addPath path
```

Add the directory in *path* to the list of directories that are searched for files specified by other commands. The following example illustrates this command.

```
> addPath /disk/howard
/disk/howard added to file search path.
```

After the above command is entered, the program will search for a file in this directory if a file is specified in a command. Thus, if a user has a library named “/disk/howard/rings.lib” and the user wants to load that library, one only needs to enter `load rings.lib` and not `load /disk/howard/rings.lib`.

3.4.6. addPdbAtomMap

```
addPdbAtomMap list
```

The atom Name Map is used to try to map atom names read from PDB files to atoms within residue UNITS when the atom name in the PDB file does not match an atom in the residue. This enables PDB files to be read in without extensive editing of atom names. Typically, this command is placed in the LEaP startup file, “leaprc”, so that assignments are made at the beginning of the session. *list* should be a LIST of LISTS. Each sublist should contain two entries to add to the Name Map. Each entry has the form:

```
{ string string }
```

where the first string is the name within the PDB file, and the second string is the name in the residue UNIT.

3.4.7. addPdbResMap

```
addPdbResMap list
```

The Name Map is used to map RESIDUE names read from PDB files to variable names within LEaP. Typically, this command is placed in the LEaP startup file, “leaprc”, so that assignments are made at the beginning of the session. The LIST is a LIST of LISTS. Each sublist contains two or three entries to add to the Name Map. Each entry has the form:

```
{ double string1 string2 }
```

where *double* can be 0 or 1, *string1* is the name within the PDB file, and *string2* is the variable name to which *string1* will be mapped. To illustrate, the following is part of the Name Map that exists when LEaP is started from the “leaprc” file included in the distribution:

```
ADE --> DADE
: :
0 ALA --> NALA
0 ARG --> NARG
: :
1 ALA --> CALA
1 ARG --> CARG
: :
1 VAL --> CVAL
```

3. LEaP

Thus, the residue ALA will be mapped to NALA if it is the N-terminal residue and CALA if it is found at the C-terminus. The above Name Map was produced using the following (edited) command line:

```
> addPdbResMap {  
> { 0 ALA NALA } { 1 ALA CALA }  
> { 0 ARG NARG } { 1 ARG CARG } : :  
> { 0 VAL NVAL } { 1 VAL CVAL }  
> : :  
> { ADE DADE } : :  
> }
```

3.4.8. alias

```
alias [ string1 [ string2 ] ]
```

This command will add or remove an entry to the Alias Table or list entries in the Alias Table. If both strings are present, then *string1* becomes the alias to *string2*, the original command. If only one string is used as an argument, then that string will be removed from the Alias Table. If no arguments are given to the command, the current aliases stored in the Alias Table will be listed.

The proposed alias is first checked for conflict with the LEaP commands and rejected if a conflict is found. A proposed alias will replace an existing alias with a warning being issued. The alias can stand for more than a single word, but also as an entire string so the user can quickly repeat entire lines of input.

3.4.9. bond

```
bond atom1 atom2 [ order ]
```

Create a bond between *atom1* and *atom2*. Both of these ATOMs must be contained by the same UNIT. By default, the bond will be a single bond. By specifying “-”, “=”, “#”, or “:” as the optional argument, order, the user can specify a single, double, triple, or aromatic bond, respectively. Example:

```
bond trx.32.SG trx.35.SG
```

3.4.10. bondByDistance

```
bondByDistance container [ maxBond ]
```

Create single bonds between all ATOMs in the UNIT *container* that are within *maxBond* Å of each other. If *maxBond* is not specified, a default distance will be used. This command is especially useful in building molecules. Example:

```
bondByDistance alkylChain
```

3.4.11. check

```
check unit [ parms ]
```

This command can be used to check *unit* for internal inconsistencies that could cause problems when performing calculations. This is a very useful command that should be used before a UNIT is saved with `saveAmberParm` or its variants. Currently it checks for the following possible problems:

- long bonds
- short bonds
- non-integral total charge of the UNIT
- missing force field atom types
- close contacts ($< 1.5 \text{ \AA}$) between nonbonded ATOMs

The user may collect any missing molecular mechanics parameters in a PARMSET for subsequent editing. In the following example, the alanine UNIT found in the amino acid library has been examined by the check command:

```
> check ALA
Checking 'ALA'....
Checking parameters for unit 'ALA'.
Checking for bond parameters.
Checking for angle parameters.
Unit is OK.
```

3.4.12. combine

```
variable = combine list
```

Combine the contents of the UNITs within *list* into a single UNIT. The new UNIT is placed in *variable*. This command is similar to the sequence command except it does not link the ATOMs of the UNITs together. In the following example, the input and output should be compared with the example given for the sequence command.

```
> tripeptide = combine { ALA GLY PRO }
Sequence: ALA
Sequence: GLY
Sequence: PRO
> desc tripeptide
UNIT name: ALA !! bug: this should be tripeptide!
Head atom: .R<ALA 1>.A<N 1>
Tail atom: .R<PRO 3>.A<C 13>
Contents:
R<ALA 1>
R<GLY 2>
R<PRO 3>
```

3. LEaP

3.4.13. copy

```
newvariable = copy variable
```

Creates an exact duplicate of the object *variable*. Since *newvariable* is not pointing to the same object as *variable*, changing the contents of one object will not alter the other object.

Example:

```
> tripeptide = sequence { ALA GLY PRO }
> tripeptideSol = copy tripeptide
> solvateBox tripeptideSol WATBOX216 8 2
```

In the above example, *tripeptide* is a separate object from *tripeptideSol* and is not solvated. Had the user instead entered

```
> tripeptide = sequence { ALA GLY PRO }
> tripeptideSol = tripeptide
> solvateBox tripeptideSol WATBOX216 8 2
```

then both *tripeptide* and *tripeptideSol* would be solvated since they would both refer to the same object.

3.4.14. createAtom

```
variable = createAtom name type charge
```

Return a new and empty ATOM with *name*, *type*, and *charge* as its atom name, atom type, and electrostatic point charge. (See the `add` command for an example of the `createAtom` command.)

3.4.15. createResidue

```
variable = createResidue name
```

Return a new and empty RESIDUE with the name *name*. (See the `add` command for an example of the `createResidue` command.)

3.4.16. createUnit

```
variable = createUnit name
```

Return a new and empty UNIT with the name *name*. (See the `add` command for an example of the `createUnit` command.)

3.4.17. deleteBond

```
deleteBond atom1 atom2
```

Delete the bond between the ATOMs *atom1* and *atom2*. If no bond exists, an error will be displayed.

3.4.18. desc

```
desc variable
```

Print a description of the object *variable*. In the following example, the alanine UNIT found in the amino acid library has been examined by the `desc` command:

```
> desc ALA
UNIT name: ALA
Head atom: .R<ALA 1>.A<N 1>
Tail atom: .R<ALA 1>.A<C 9>
Contents: R<ALA 1>
```

Now, the `desc` command is used to examine the first residue (1) of the alanine UNIT:

```
> desc ALA.1
RESIDUE name: ALA
RESIDUE sequence number: 1
Type: protein
Connection atoms:
Connect atom 0: A<N 1>
Connect atom 1: A<C 9>
Contents:
A<N 1>
A<HN 2>
A<CA 3>
A<HA 4>
A<CB 5>
A<HB1 6>
A<HB2 7>
A<HB3 8>
A<C 9>
A<O 10>
```

Next, we illustrate the `desc` command by examining the ATOM N of the first residue (1) of the alanine UNIT:

```
> desc ALA.1.N
ATOM Name: N
Type: N
Charge: -0.463
Element: N
Atom flags: 20000|posfxd- posblt- posdrn- sel- pert- notdisp- tchd-
             posknwn+ int - nmin- nbld-
Atom position: 3.325770, 1.547909, -0.000002
Atom velocity: 0.000000, 0.000000, 0.000000
Bonded to .R<ALA 1>.A<HN 2> by a single bond.
Bonded to .R<ALA 1>.A<CA 3> by a single bond.
```

3. LEaP

Since the N ATOM is also the first atom of the ALA residue, the following command will give the same output as the previous example:

```
> desc ALA.1.1
```

3.4.19. groupSelectedAtoms

```
groupSelectedAtoms unit name
```

Create a group within *unit* with the name *name*, using all of the ATOMs within *unit* that are selected. If the group has already been defined then overwrite the old group. The `desc` command can be used to list groups. Example:

```
groupSelectedAtoms TRP sideChain
```

An expression like “TRP@sideChain” returns a LIST, so any commands that require LISTS can take advantage of this notation. After assignment, one can access groups using the “@” notation. Examples:

```
select TRP@sideChain  
center TRP@sideChain
```

The latter example will calculate the center of the atoms in the “*sideChain*” group. (See the `select` command for a more detailed example.)

3.4.20. help

```
help [string]
```

This command prints a description of the command in *string*. If no argument is given, a list of help topics is provided.

3.4.21. impose

```
impose unit seqlist internals
```

The `impose` command allows the user to impose internal coordinates on *unit*. The list of RESIDUEs to impose the internal coordinates upon is in *seqlist*. The internal coordinates to impose are in *internals*, which is an object of type LIST.

The command works by looking into each RESIDUE within *unit* that is listed in *seqlist* and attempts to apply each of the internal coordinates within *internals*. The *seqlist* argument is a LIST of NUMBERS that represent sequence numbers or ranges of sequence numbers. A range of sequence numbers is represented by two element LISTS that contain the first and last sequence number in the range. The user can specify sequence number ranges that are larger than what is found in *unit*, in which case the range will stop at the beginning or end of *unit* as appropriate. For example, the range { 1 999 } will include all RESIDUEs in a 200 RESIDUE UNIT.

The *internals* argument is a LIST of LISTS. Each sublist contains a sequence of ATOM names which are of type STRING followed by the value of the internal coordinate. An example of the `impose` command would be:

```
impose peptide { 1 2 3 } { { "N" "CA" "C" "N" -40.0 } { "C" "N" "CA" "C" -60.0 } }
```

This would cause the RESIDUE with sequence numbers 1, 2, and 3 within the UNIT *peptide* to assume an α -helical conformation. The command

```
impose peptide { 1 2 { 5 10 } 12 } { { "CA" "CB" 5.0 } }
```

will impose on the residues with sequence numbers 1, 2, 5, 6, 7, 8, 9, 10, and 12 within the UNIT *peptide* a bond length of 5.0 Å between the α and β carbon atoms. RESIDUES without an ATOM named CB, such as glycine, will be unaffected.

It is important to understand that the impose command attempts to perform the intended action on all residues in the *seqlist*, but does not necessarily limit itself to acting only upon *internals* contained within those residues. That is, the list does not limit the residues to consider. Rather, it is a list of all starting points to consider. In other words, to specify a *seqlist* of { 3 4 } tells *impose* to attempt to set two torsions, one starting in residue 3 and the other starting in residue 4. It does not specify that the torsion should only be set if the atoms are found within residues 3 and/or 4.

Because of this, one must be careful when setting torsions between two residues. It is necessary to know which atoms are contained in which residues. Consider the following trisaccharide:



To build it most simply in leap requires the following directive. Note that the build order in leap is the reverse of the standard order in which the residues are written above.

```
glycan = sequence { ROH 6LB 6MB 0GA }
```

A proper build of a 1-6 oligosaccharide linkage often requires setting three torsions. In the manner that residues are defined in the Glycam force fields, the atoms describing two of those torsions, ϕ and ψ , span two residues. However, the atoms in the third, ω , exist entirely within one residue. In fact, they exist within all three glycan residues in the example above. The following commands will set only the three torsions in the glycosidic linkage between residues 4 (0GA) and 3 (6MB).

```
impose glycan { 4 } { { "H1" "C1" "O6" "C6" -60.0 } } # O6 & C6 are in residue 3
impose glycan { 4 } { { "C1" "O6" "C6" "C5" 180.0 } } # only C1 is in residue 4
impose glycan { 3 } { { "O6" "C6" "C5" "O5" 60.0 } } # all are in residue 3
```

The common misconception that the *seqlist* sets a limit on the residues affected can cause trouble in this case. For example, this command

```
impose glycan { 4 3 } { { "H1" "C1" "O6" "C6" -60.0 } }
```

will find all sequences beginning in residue 4 and in residue 3 that contain the serially bonded atoms H1 C1 O6 and C6. Therefore, in this case, it will set the specified torsions between residues 4 and 3 as well as between 3 and 2. Similarly, this command

3. LEaP

```
impose peptide { 4 } { { "O6" "C6" "C5" "O5" 60.0 } }
```

will not affect any inter-residue linkage, but instead will set the C5-C6 torsion in the glucopyranoside (OGA) at the non-reducing end of the oligosaccharide.

The ordering and content within the *internals* list is important as well. For these examples, consider the simple peptide sequence:

```
peptide = sequence { ALA ALA ALA ALA }
```

The ordering of the *internals* specifies the atoms to which the torsion set is applied. The *impose* command will find the first atom in the *internals* list, check for the presence of a bonded second atom, and so forth. It will then apply the action, here a torsion, to those four atoms. For example, this command:

```
impose peptide { 3 } { { "N" "CA" "C" "N" -40.0 } } # between 3 and 4
```

will set the torsion between residues 3 and 4. However, this one:

```
impose peptide { 3 } { { "N" "C" "CA" "N" -40.0 } } # between 3 and 2
```

will set the torsion between residues 3 and 2.

If at any point, the *impose* command does not find an atom bonded to a previous atom in an *internals* list, it will silently ignore the command. This is likely to occur in two instances. One, the atom simply might not exist in the residue:

```
impose peptide { 3 } { { "N" "CA" "CB" "HB4" 10.0 } } # no effect, silent
```

Here, of course, there is no atom named HB4 in alanine. Similarly, improper torsions are ignored. For example, this command also has no effect:

```
impose peptide { 3 } { { "N" "HB1" "CA" "CB" 10.0 } } # no effect, silent
```

because HB1 is not bonded to N.

Three types of conformational change are supported: Bond length changes, bond angle changes, and torsion angle changes. If the conformational change involves a torsion angle, then all dihedrals around the central pair of atoms are rotated. The entire list of *internals* is applied to each RESIDUE.

It is also important to note that the *impose* command performs its actions entirely using internal coordinates. Because of this, it is difficult to predict the resulting behavior when the coordinates are translated back to cartesian, for example when writing a PDB file.

3.4.22. list

List all of the variables currently defined. To illustrate, the following (edited) output shows the variables defined when LEaP is started from the leaprc file included in the distribution:

```
> list A ACE ALA ARG ASN : : VAL W WAT Y
```

3.4.23. loadAmberParams

```
variable = loadAmberParams filename
```

Load an AMBER format parameter set file and place it in *variable*. All interactions defined in the parameter set will be contained within *variable*. This command causes the loaded parameter set to be included in LEaP's list of parameter sets that are searched when parameters are required. General proper and improper torsion parameters are modified during the command execution with the LEaP general type "?" replacing the AMBER general type "X"

```
> parm91 = loadAmberParams parm91X.dat
> saveOff parm91 parm91.lib
```

3.4.24. loadAmberPrep

```
loadAmberPrep filename [ prefix ]
```

This command loads an AMBER PREP input file. For each residue that is loaded, a new UNIT is constructed that contains a single RESIDUE and a variable is created with the same name as the name of the residue within the PREP file. If the optional argument *prefix* (a STRING) is provided, its contents will be prefixed to each variable name; this feature is used to prefix UATOM residues, which have the same names as AATOM residues with the string "U" to distinguish them.

```
> loadAmberPrep cra.in
Loaded UNIT: CRA
```

3.4.25. loadOff

```
loadOff filename
```

This command loads the OFF library within the file named *filename*. All UNITS and PARMSETs within the library will be loaded. The objects are loaded into LEaP under the variable names the objects had when they were saved. Variables already in existence that have the same names as the objects being loaded will be overwritten. Any PARMSETs loaded using this command are included in LEaP's library of PARMSETs that is searched whenever parameters are required (the old AMBER format is used for PARMSETs rather than the OFF format in the default configuration). Example command line:

```
> loadOff parm91.lib
Loading library: parm91.lib
Loading: PARAMETERS
```

3.4.26. loadMol2

```
variable = loadMol2 filename
```

Load a Sybyl MOL2 format file into *variable*, a UNIT. This command is very much like *loadOff*, except that it only creates a single UNIT.

3. LEaP

3.4.27. loadPdb

```
variable = loadPdb filename
```

Load a Protein Data Bank (PDB) format file with the file name *filename* into *variable*, a UNIT. The sequence numbers of the RESIDUES will be determined from the order of residues within the PDB file ATOM records. This function will search the variables currently defined within LEaP for variable names that map to residue names within the ATOM records of the PDB file. If a matching variable name is found then the contents of the variable are added to the UNIT that will contain the structure being loaded from the PDB file. Adding the contents of the matching UNIT into the UNIT being constructed means that the contents of the matching UNIT are copied into the UNIT being built and that a bond is created between the connect0 ATOM of the matching UNIT and the connect1 ATOM of the UNIT being built. The UNITs are combined in the same way UNITs are combined using the sequence command. As atoms are read from the ATOM records their coordinates are written into the correspondingly named ATOMs within the UNIT being built. If the entire residue is read and it is found that ATOM coordinates are missing, then external coordinates are built from the internal coordinates that were defined in the matching UNIT. This allows LEaP to build coordinates for hydrogens and lone-pairs which are not specified in PDB files.

```
> crambin = loadPdb 1crn
```

3.4.28. loadPdbUsingSeq

```
loadPdbUsingSeq filename unitlist
```

This command reads a PDB format file named *filename*. This command is identical to `loadPdb` except it does not use the residue names within the PDB file. Instead, the sequence is defined by the user in *unitlist*. For more details see `loadPdb`.

```
> peptSeq = { UALA UASN UILE UVAL UGLY }  
> pept = loadPdbUsingSeq pept.pdb peptSeq
```

In the above example, a variable is first defined as a LIST of united atom RESIDUES. A PDB file is then loaded, in this sequence order, from the file "pept.pdb".

3.4.29. logFile

```
logFile filename
```

This command opens the file with the file name *filename* as a log file. User input and all output is written to the log file. Output is written to the log file as if the verbosity level were set to 2. An example of this command is

```
> logfile /disk/howard/leapTrpSolvate.log
```

3.4.30. measureGeom

```
measureGeom atom1 atom2 [ atom3 [ atom4 ] ]
```

Measure the distance, angle, or torsion between two, three, or four ATOMs, respectively.

In the following example, we first describe the RESIDUE ALA of the ALA UNIT in order to find the identity of the ATOMs. Next, the `measureGeom` command is used to determine a distance, simple angle, and a dihedral angle. As shown in the example, the ATOMs may be identified using atom names or numbers.

```
> desc ALA.ALA
RESIDUE name: ALA
RESIDUE sequence number: 1
Type: protein ....
```

3.4.31. quit

Quit the LEaP program.

3.4.32. remove

```
remove a b
```

Remove the object *b* from the object *a*. If *a* does not contain *b*, an error message will be displayed. This command is used to remove ATOMs from RESIDUEs, and RESIDUEs from UNITs. If the object represented by *b* is not referenced by any other variable name, it will be destroyed.

```
> dipeptide = combine { ALA GLY }
Sequence: ALA
Sequence: GLY
> desc dipeptide
UNIT name: ALA !! bug: this should be dipeptide!
Head atom: .R<ALA 1>.A<N 1>
Tail atom: .R<GLY 2>.A<C 6>
Contents: R<ALA 1> R<GLY 2>
> remove dipeptide dipeptide.2
> desc dipeptide UNIT name: ALA !! bug: this should be dipeptide!
Head atom: .R<ALA 1>.A<N 1>
Tail atom: null
Contents: R<ALA 1>
```

3.4.33. saveAmberParm

```
saveAmberParm unit topologyfilename coordinatefilename
```

3. LEaP

Save the Amber/NAB topology and coordinate files for *unit* into the files named *topologyfilename* and *coordinatefilename* respectively. This command will cause LEaP to search its list of PARMSETs for parameters defining all of the interactions between the ATOMs within *unit*. It produces topology files and coordinate files that are identical in format to those produced by Amber PARM and can be read into Amber and NAB for calculations. The output of this operation can be used for minimizations, dynamics, and thermodynamic perturbation calculations.

In the following example, the topology and coordinates from the all_amino94.lib UNIT ALA are generated:

```
> saveamberparm ALA ala.top ala.crd
```

3.4.34. saveMol2

```
saveMol2 unit filename type-flag
```

Write *unit* to the file *filename* as a Tripos mol2 format file. If *type-flag* is 0, the Tripos (Sybyl) atom types will be used; if *type-flag* is 1, the Amber atom types present in *unit* will be used. Generally, you would want to set *type-flag* to 1, unless you need the Sybyl atom types for use in some program outside Amber; Amber itself has no force fields that use Sybyl atom types.

3.4.35. saveOff

```
saveOff object filename
```

The `saveOff` command allows the user to save UNITS and PARMSETs to a file named *filename*. The file is written using the Object File Format (off) and can accommodate an unlimited number of uniquely named objects. The names by which the objects are stored are the variable names specified within the *object* argument. If the file *filename* already exists, the new objects will be added to it. If there are objects within the file with the same names as objects being saved then the old objects will be overwritten. The argument *object* can be a single UNIT, a single PARMSET, or a LIST of mixed UNITS and PARMSETs. (See the `add` command for an example of the `saveOff` command.)

3.4.36. savePdb

```
savePdb unit filename
```

Write *unit* to the file *filename* as a PDB format file. In the following example, the PDB file from the “all_amino94.lib” UNIT ALA is generated:

```
> savepdb ALA ala.pdb
```

3.4.37. sequence

```
variable = sequence list
```


The `sequence` command is used to combine the contents of *list*, which should be a LIST of UNITS, into a new, single UNIT. This new UNIT is constructed by taking each UNIT in *list* in turn and copying its contents into the UNIT being constructed. As each new UNIT is copied, a bond is created between the tail ATOM of the UNIT being constructed and the head ATOM of the UNIT being copied, if both connect ATOMs are defined. If only one is defined, a warning is generated and no bond is created. If neither connection ATOM is defined then no bond is created. As each RESIDUE is copied into the UNIT being constructed it is assigned a sequence number which represents the order the RESIDUES are added. Sequence numbers are assigned to the RESIDUES so as to maintain the same order as was in the UNIT before it was copied into the UNIT being constructed. This command builds reasonable starting coordinates for all ATOMs within the UNIT; it does this by assigning internal coordinates to the linkages between the RESIDUES and building the external coordinates from the internal coordinates from the linkages and the internal coordinates that were defined for the individual UNITS in the sequence.

```
> tripeptide = sequence { ALA GLY PRO }
```

3.4.38. set

```
set default variable value
or set container parameter object
```

This command sets the values of some global parameters (when the first argument is “default”) or sets various parameters associated with *container*. The following parameters can be set within LEaP:

For “default” parameters:

OldPrmtopFormat If set to “on”, the `saveAmberParm` command will write a prmtop file in the format used in Amber 6 and earlier versions; if set to “off” (the default), it will use the new format.

Dielectric If set to “distance” (the default), electrostatic calculations in LEaP will use a distance-dependent dielectric; if set to “constant”, a constant dielectric will be used.

PdbWriteCharges If set to “on”, atomic charges will be placed in the “B-factor” field of PDB files saved with the `savePdb` command; if set to “off” (the default), no such charges will be written.

PBRadii Used to choose various sets of atomic radii for generalized Born or Poisson-Boltzmann calculations. Options are: “bondi”, which gives values from Ref. [64], which may be used with *igb* = 2, 5 or 7; “mbondi”, which is the default, and the recommended parameter set for *igb* = 1 [65]; “mbondi2”, which is a second modification of the Bondi radii set [66], and can also be used with *igb* = 2 or 5; and “amber6”, which is only to be used for reproducing very early calculations that used *igb* = 1 [67].

For ATOMs:

name A unique STRING descriptor used to identify ATOMs.

3. LEaP

type This is a STRING property that defines the AMBER force field atom type.

charge The charge property is a NUMBER that represents the ATOM's electrostatic point charge to be used in a molecular mechanics force field.

position This property is a LIST of NUMBERS containing three values: the (X, Y, Z) Cartesian coordinates of the ATOM.

pertName This STRING is a unique identifier for an ATOM in its final state during a Free Energy Perturbation calculation.

pertType This STRING is the AMBER force field atom type of a perturbed ATOM.

pertCharge This NUMBER represents the final electrostatic point charge on an ATOM during a Free Energy Perturbation.

For RESIDUES:

connect0 This identifies the first of up to three ATOMs that will be used to make links to other RESIDUES. In a UNIT containing a single RESIDUE, the RESIDUE's connect0 ATOM is usually defined as the UNIT's head ATOM.

connect1 This identifies the second of up to three ATOMs that will be used to make links to other RESIDUES. In a UNIT containing a single RESIDUE, the RESIDUE's connect1 ATOM is usually defined as the UNIT's tail ATOM.

connect2 This identifies the third of up to three ATOMs that will be used to make links to other RESIDUES. In amino acids, the convention is that this is the ATOM to which disulfide bridges are made.

restype This property is a STRING that represents the type of the RESIDUE. Currently, it can have one of the following values: "undefined", "solvent", "protein", "nucleic", or "saccharide".

name This STRING property is the RESIDUE name.

For UNITS:

head Defines the ATOM within the UNIT that is connected when UNITS are joined together: the tail ATOM of one UNIT is connected to the head ATOM of the subsequent UNIT in any sequence.

tail Defines the ATOM within the UNIT that is connected when UNITS are joined together: the tail ATOM of one UNIT is connected to the head ATOM of the subsequent UNIT in any sequence.

box This property defines the bounding box of the UNIT. If it is set to null then no bounding box is defined. If it is a single NUMBER, the bounding box will be defined to be a cube with each side being box Å across. If it is a LIST, it must contain three NUMBERS, the lengths (in Å) of the three sides of the bounding box.

cap This property defines the solvent cap of the UNIT. If it is set to null then no solvent cap is defined. Otherwise, it should be a LIST of four NUMBERS; the first three NUMBERS define the Cartesian coordinates (X, Y, Z) of the origin of the solvent cap in Å, while the fourth defines the radius of the solvent cap, also in Å.

3.4.39. solvateBox and solvateOct

```
solvateBox solute solvent distance [ closeness ]
solvateOct solute solvent distance [ closeness ]
```

These two commands create periodic solvent boxes around *solute*, which should be a UNIT. *solvateBox* creates a cuboid box, while *solvateOct* creates a truncated octahedron. *solute* is modified by the addition of copies of the RESIDUES found within *solvent*, which should also be a UNIT, such that the closest distance between any atom originally present in *solute* and the edge of the periodic box is given by the *distance* parameter. The resulting solvent box will be repeated in all three spatial directions.

The optional *closeness* parameter can be used to control how close, in Å, solvent ATOMS may come to solute ATOMS. The default value of *closeness* is 1.0. Smaller values allow solvent ATOMS to come closer to solute ATOMS. The criterion for rejection of overlapping solvent RESIDUES is if the distance between any solvent ATOM and its nearest solute ATOM is less than the sum of the two ATOMS' van der Waals radii multiplied by *closeness*.

```
> mol = loadpdb my.pdb
> solvateOct mol TIP3PBOX 12.0 0.75
```

3.4.40. solvateCap

```
solvateCap solute solvent position radius [ closeness ]
```

The *solvateCap* command creates a solvent cap around *solute*, which is a UNIT. *solute* is modified by the addition of copies of the RESIDUES found within *solvent*, which should also be a UNIT. The solvent box will be repeated in all three spatial directions to create a large solvent sphere with a radius of *radius* Å.

The *position* argument defines where the center of the solvent cap is to be placed. If *position* is a UNIT, a RESIDUE, an ATOM, or a LIST of UNITS, RESIDUES, or ATOMS, then the geometric center of the ATOM or ATOMS within the object will be used as the center of the solvent cap sphere. If *position* is a LIST containing three NUMBERS, then it will be treated as a vector describing the position of the solvent cap sphere center.

The optional *closeness* parameter can be used to control how close, in Å, solvent ATOMS may come to solute ATOMS. The default value of *closeness* is 1.0. Smaller values allow solvent ATOMS to come closer to solute ATOMS. The criterion for rejection of overlapping solvent RESIDUES is if the distance between any solvent ATOM and its nearest solute ATOM is less than the sum of the two ATOMS' van der Waals radii multiplied by *closeness*.

This command modifies *solute* in several ways. First, the UNIT is modified by the addition of solvent RESIDUES copied from *solvent*. Secondly, the “cap” parameter of *solute* is modified to reflect the fact that a solvent cap has been created around the solute.

3. LEaP

```
> mol = loadpdb my.pdb
> solvateCap mol WATBOX216 mol.2.CA 12.0 0.75
```

3.4.41. solvateShell

```
solvateShell solute solvent thickness [ closeness ]
```

The `solvateShell` command adds a solvent shell to *solute*, which should be a UNIT. *solute* is modified by the addition of copies of the RESIDUES found within *solvent*, which should also be a UNIT. The resulting solute/solvent UNIT will be irregular in shape since it will reflect the contours of the original solute molecule. The solvent box will be repeated in three directions to create a large solvent box that can contain the entire solute and a shell *thickness* Å thick. Solvent RESIDUES are then added to *solute* if they lie within the shell defined by *thickness* and do not overlap with any ATOM originally present in *solute*. The optional *closeness* parameter can be used to control how close solvent ATOMS can come to solute ATOMS. The default value of the *closeness* argument is 1.0. Please see the `solvateBox` command for more details on the closeness parameter.

```
> mol = loadpdb my.pdb
> solvateShell mol WATBOX216 12.0 0.8
```

3.4.42. source

```
source filename
```

This command executes the contents of the file given by *filename*, treating them as LEaP commands. To display the commands as they are read, see the `verbosity` command.

3.4.43. transform

```
transform atoms, matrix
```

Transform all of the ATOMS within *atoms* by a symmetry operation. The symmetry operation is represented as a (3 × 3) or (4 × 4) matrix, and given as nine or sixteen NUMBERS in *matrix*, a LIST of LISTS. The general matrix looks like:

```
r11 r12 r13 -tx r21 r22 r23 -ty r31 r32 r33 -tz 0 0 0 1
```

The matrix elements represent the intended symmetry operation. For example, a reflection in the (x,y) plane would be produced by the matrix:

```
1 0 0 0 1 0 0 0 -1
```

This reflection could be combined with a 6 Å translation along the x-axis by using the following matrix:

```
1 0 0 6 0 1 0 0 0 0 -1 0 0 0 0 1
```

In the following example, wrB is transformed by an inversion operation:

```
transform wrpB { { -1 0 0 } { 0 -1 0 } { 0 0 -1 } }
```

3.4.44. translate

```
translate atoms direction
```

Translate all of the ATOMs within *atoms* by the vector given by *direction*, a LIST of three NUMBERS.

Example:

```
translate wrpB { 0 0 -24.53333 }
```

3.4.45. verbosity

```
verbosity level
```

This command sets the level of output that LEaP provides the user. A value of 0 is the default, providing the minimum of messages. A value of 1 will produce more output, and a value of 2 will produce all of the output of level 1 and display the text of the script lines executed with the source command. The following line is an example of this command:

```
> verbosity 2
Verbosity level: 2
```

3.4.46. zMatrix

```
zMatrix object zmatrix
```

The `zMatrix` command is quite complicated. It is used to define the external coordinates of ATOMs within *object* using internal coordinates. The second parameter of the `zMatrix` command is a LIST of LISTS; each sub-list has several arguments:

```
{ a1 a2 bond12 }
```

This entry defines the coordinate of *a1*, an ATOM, by placing it *bond12* Å along the x-axis from ATOM *a2*. *a2* is placed at the origin if its coordinates are not defined.

```
{ a1 a2 a3 bond12 angle123 }
```

This entry defines the coordinate of *a1* by placing it *bond12* Å away from *a2* making an angle of *angle123* degrees between *a1*, *a2* and *a3*. The angle is measured in a right-hand sense and in the xy plane. ATOMs *a2* and *a3* must have coordinates defined.

```
{ a1 a2 a3 a4 bond12 angle123 torsion1234 }
```

This entry defines the coordinate of *a1* by placing it *bond12* Å away from *a2*, creating an angle of *angle123* degrees between *a1*, *a2*, and *a3*, and making a torsion angle of *torsion1234* degrees between *a1*, *a2*, *a3*, and *a4*.

```
{ a1 a2 a3 a4 bond12 angle123 angle124 orientation }
```

3. LEaP

This entry defines the coordinate of *a1* by placing it *bond12* Å away from *a2*, and making angles *angle123* degrees between *a1*, *a2*, and *a3*, and *angle124* degrees between *a1*, *a2*, and *a4*. The argument *orientation* defines whether *a1* is above or below a plane defined by *a2*, *a3* and *a4*. If *orientation* is positive, *a1* will be placed so that the triple product $((a3-a2) \times (a4-a2)) \cdot (a1-a2)$ is positive. Otherwise, *a1* will be placed on the other side of the plane. This allows the coordinates of a molecule like fluoro-chloro-bromo-methane to be defined without having to resort to dummy atoms.

The first arguments within the `zMatrix` entries (*a1*, *a2*, *a3* and *a4*) are either ATOMs, or STRINGs containing names of ATOMs that already exist within *object*. The subsequent arguments (*bond12*, *angle123*, *torsion1234* or *angle124*, and *orientation*) are all NUMBERs. Any ATOM can be placed at the *a1* position, even one that has coordinates defined. This feature can be used to provide an endless supply of dummy atoms, if they are required. A predefined dummy atom with the name "*" (a single asterisk, no quotes) can also be used.

There is no order imposed in the sub-lists. The user can place sub-lists in arbitrary order, as long as they maintain the requirement that all ATOMs *a2*, *a3*, and *a4* must have external coordinates defined, except for entries that define the coordinate of an ATOM using only a bond length. (See the `add` command for an example of the `zMatrix` command.)

3.5. Building oligosaccharides and lipids

The approach presented below has been automated, with many additional options available, at the GLYCAM-Web site: www.glycam.org.

Before continuing in this section, you should review the GLYCAM naming conventions covered in Section 2.9. After that, there are two important things to keep in mind. The first is that GLYCAM is designed to build oligosaccharides, not just monosaccharides. In order to link the monosaccharides together, each residue in GLYCAM will have at least one open valence position. That is, each GLYCAM residue lacks either a hydroxyl group or a hydroxyl proton, and may be lacking more than one proton depending on the number of branching locations. Thus, none of the residues is a complete molecule unto itself. For example, if you wish to build α -D-glucopyranose, you must explicitly specify the anomeric -OH group (see Figure 3.1 for two examples).

The second thing to keep in mind is that when the `sequence` command is used in LEaP to link monosaccharides together to form a linear oligosaccharide (analogous to peptide generation), the residue ordering is opposite to the standard convention for writing the sequence. For example, to build the disaccharides illustrated in Figure 3.1, using the `sequence` command in LEaP, the format would be:

```
upperdisacc = sequence { ROH 3GB 0GB }
lowerdisacc = sequence { OME 4GB 0GA }
```

While the `sequence` command is the most direct method to build a linear glycan, it is not the only method. Alternatives that facilitate building more complex glycans and glycoproteins are presented below. For those who need to build structures (and generate topology and coordinate files) that are more complex, a convenient interface that uses GLYCAM is available on the internet (<http://glycam.ccrcc.uga.edu> or <http://www.glycam.org>).

3.5. Building oligosaccharides and lipids

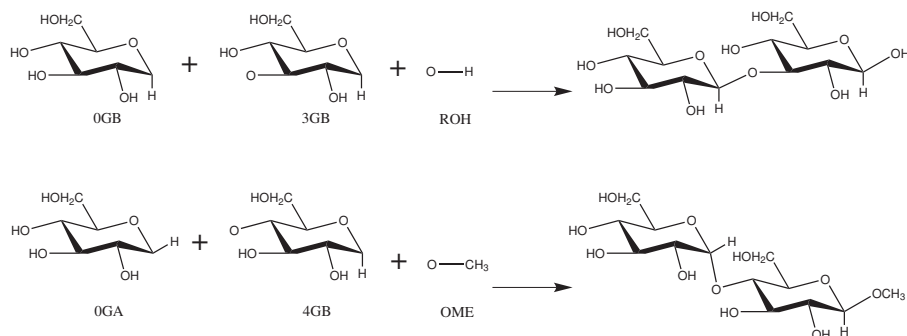


Figure 3.1.: Schematic representation of disaccharide formation, indicating the need for open valences on carbon and oxygen atoms at linkage positions.

Throughout this section, sequences of LEaP commands will be entered in the following format:

```
command argument(s) # descriptive comment
```

This format was chosen so that the lines can be copied directly into a file to be read into LEaP. The number sign (#) signifies a comment. Comments following commands may be left in place for future reference and will be ignored by LEaP. Files may be read into LEaP either by sourcing the file or by specifying it on the command line at the time that LEaP is invoked, e.g.:

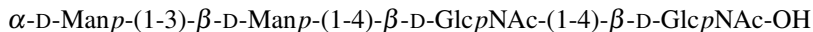
```
tleap -f leap_input_file
```

Note that any GLYCAM parameter set shipped with Amber is likely to be updated in the future. The current version is *GLYCAM_06h.dat*. This file and *GLYCAM_06h.prep* are automatically loaded with the default *leaprc.GLYCAM_06h*. The user is encouraged to check www.glycam.org for updated versions of these files.

3.5.1. Procedures for building oligosaccharides using the GLYCAM-06 parameters

3.5.1.1. Example: Linear oligosaccharides

This section contains instructions for building a simple, straight-chain tetrasaccharide:



First, it is necessary to determine the GLYCAM residues that will be used to build it. Since the initial $\alpha\text{-D-Manp}$ residue links only at its anomeric site, the first character in its name is 0 (zero), indicating that it has no branches or other connections, i.e., it is terminal. Since it is a D-mannose, the second character, the one-letter code, is M (capital). Since it is an α -pyranose, the third character is A. Therefore, the first residue in the sequence above is 0MA. Since the second

3. LEaP

residue links at its 3-position as well as at the anomeric position, the first character in its name is 3, and, being a β -pyranose, it is 3MB. Similarly, residues three and four are both 4YB. It will also be necessary to add an OH residue at the end to generate a complete molecule. Note that in Section 3.5.3, below, the terminal OH *must* be omitted in order to allow subsequent linking to a protein or lipid. Note also that when present, a terminal OH (or OME etc) is assigned its own residue number.

Converting the order for use with the sequence command in LEaP, gives:

```
Residue name sequence: ROH 4YB 4YB 3MB OMA
Residue number:      1   2   3   4   5
```

Here is a set of LEaP instructions that will build the sequence (there are, of course, other ways to do this):

```
source leaprc.GLYCAM_06h # load leaprc
glycan = sequence { ROH 4YB 4YB 3MB OMA } # build oligosaccharide
```

Using the `sequence` command, the ϕ angles are automatically set to the orientation that is expected on the basis of the exo-anomeric effect ($\pm 60^\circ$). If you wish to change the torsion angle between two residues, the `impose` command may be used. In the following example, the ψ angles between the two 4YB residues and between the 4YB and the 3MB are being set to the standard value of zero.

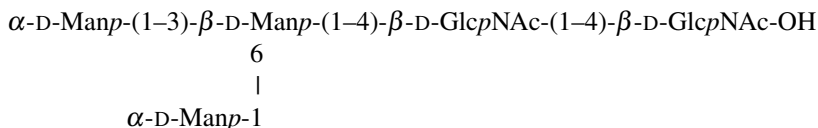
```
impose glycan {3} { {C1 O4 C4 H4 0.0} } # set psi between 4YB (3) & 4YB (2)
impose glycan {4} { {C1 O4 C4 H4 0.0} } # set psi between 3MB (4) & 4YB (3)
```

You may now generate coordinate, topology and PDB files, for example:

```
saveamberparm glycan glycan.top glycan.crd # save top & crd
savepdb glycan glycan.pdb # save pdb file
```

3.5.1.2. Example: Branched oligosaccharides

This section contains instructions for building a simple branched oligosaccharide. The example used here builds on the previous one. Again, it will be assumed that the carbohydrate is not destined to be linked to a protein or a lipid. If it were, one should omit the ROH residue from the structure. The branched oligosaccharide is



Note that the β -D-mannopyranose is now branched at the 3- and 6-positions. Consulting Tables 2.4 to 2.7 informs us that the first character assigned to a carbohydrate linked at the 3- and 6-positions is V. Thus, the name of the residue called 3MB in the previous section must change to VMB.

Thus, when rewritten for LEaP this glycan becomes:


```
Residue name sequence: ROH 4YB 4YB VMB OMA OMA
Residue number:      1   2   3   4   5   6
```

To ensure that the correct residues are linked at the 3- and 6-positions in VMB, it is safest to specify these linkages explicitly in LEaP. In the current example, the two terminal residues are the same (OMA), but that need not be the case.

```
source leaprc.GLYCAM_06h # load leaprc
glycan = sequence { ROH 4YB 4YB VMB } # linear sequence to branch
```

The longest linear sequence is built first, ending at the branch point “VMB” in order to explicitly specify subsequent linkages. The following commands will place a terminal, OMA residue at the number three position:

```
set glycan tail glycan.4.O3 # set attachment point to the O3 in VMB
glycan = sequence { glycan OMA } # add one of the OMA's
```

The following commands will link the other OMA to the 6-position. Note that the name of the molecule changes from “glycan” to “branch”. This change is not necessary, but makes such command sequences easier to read, particularly with complex structures.

```
set glycan tail glycan.4.O6 # set attachment point to the O6 in VMB
branch = sequence { glycan OMA } # add the other OMA
```

It can be especially important to reset torsion angles when building branched oligosaccharides. The following set of commands cleans up the geometry considerably and then generates a set of output files:

```
impose branch {4} { {H1 C1 O6 C6 -60.0} } # set phi torsion and
impose branch {4} { {C1 O6 C6 H6 0.0} } # set psi OMA(6) & VMB
impose branch {4} { {H1 C1 O4 C4 60.0} } # set phi torsion and
impose branch {4} { {C1 O4 C4 H4 0.0} } # set psi 3MB & 4YB
impose branch {3} { {H1 C1 O4 C4 60.0} } # set phi torsion and
impose branch {3} { {C1 O4 C4 H4 0.0} } # set psi 4YB & 4YB
impose branch {5} { {H1 C1 O3 C3 -60.0} } # set phi torsion and
impose branch {5} { {C1 O3 C3 H3 0.0} } # set psi OMA(3) & VMB
saveamberparm branch branch.top branch.crd # save top & crd
savepdb branch branch.pdb # save pdb
```

3.5.1.3. Example: Complex branched oligosaccharides

The following example builds a highly branched, high-mannose structure shown in Figure 3.2. In this example, it is especially important to note that when the branching is ambiguous, LEaP might not choose the attachment point one wants or expects. For this reason, connectivity should be specified explicitly whenever the structure branches. That is, one cannot specify the longest linear sequence and add branches later. The sequence command must be interrupted at

3. LEaP

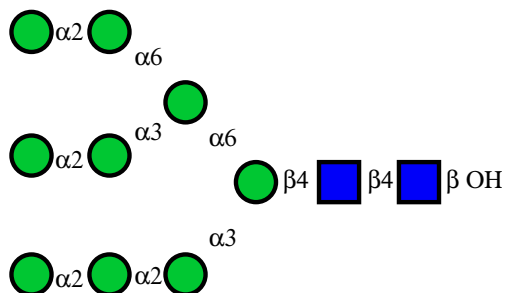
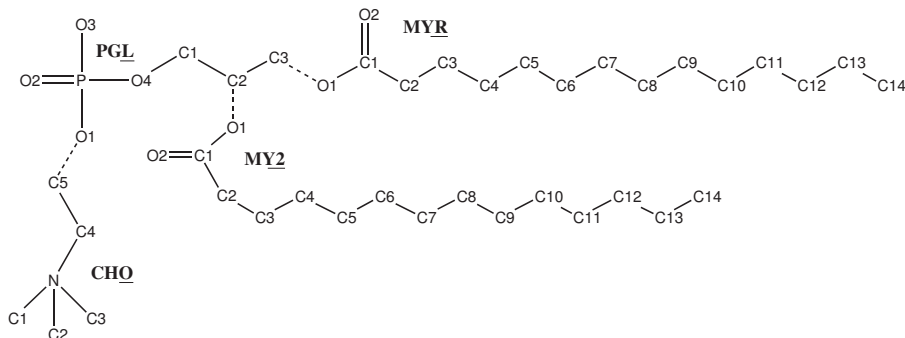


Figure 3.2.: Structure of Man-9, represented in the symbolic notation used by the Consortium for Functional Glycomics. Here, ● =D-Manp and ■ =D-GlcNAc

each branch point. Otherwise, the connectivity is not assured. In this example, a branch occurs at each VMA (-3,6-D-Manp) residue.

The following set of commands, given to tleap, will safely produce the structure represented in Figure 3.2 .

```
source leaprc.GLYCAM_06h
glycan = sequence { ROH 4YB 4YB VMB }
set glycan tail glycan.4.O6
glycan=sequence { glycan VMA }
set glycan tail glycan.5.O6
glycan=sequence { glycan 2MA OMA }
set glycan tail glycan.5.O3
glycan=sequence { glycan 2MA OMA }
set glycan tail glycan.4.O3
glycan=sequence { glycan 2MA 2MA OMA }
impose glycan {3} { {H1 C1 O4 C4 60.0} }
impose glycan {3} { {C1 O4 C4 H4 0.0} }
impose glycan {4} { {H1 C1 O4 C4 60.0} }
impose glycan {4} { {C1 O4 C4 H4 0.0} }
impose glycan {5} { {H1 C1 O6 C6 -60.0} } # 1-6 Link from (5) to (4), Phi
impose glycan {5} { {C1 O6 C6 C5 180.0} } # 1-6 Link from (5) to (4), Psi
impose glycan {4} { {O6 C6 C5 O5 60.0} } # 1-6 Link from (5) to (4), Chi
impose glycan {10} { {H1 C1 O3 C3 -60.0} }
impose glycan {10} { {C1 O3 C3 H3 0.0} }
impose glycan {6} { {H1 C1 O6 C6 -60.0} }
impose glycan {6} { {C1 O6 C6 C5 180.0} }
impose glycan {5} { {O6 C6 C5 O5 -60.0} }
impose glycan {8} { {H1 C1 O3 C3 -60.0} }
impose glycan {8} { {C1 O3 C3 H3 0.0} }
impose glycan {7} { {H1 C1 O2 C2 -60.0} }
impose glycan {7} { {C1 O2 C2 H2 0.0} }
impose glycan {9} { {H1 C1 O2 C2 -60.0} }
```

Figure 3.3.: *DMPC*

```
impose glycan {9} { {C1 O2 C2 H2 0.0} }
impose glycan {11} { {H1 C1 O2 C2 -60.0} }
impose glycan {11} { {C1 O2 C2 H2 0.0} }
impose glycan {12} { {H1 C1 O2 C2 -60.0} }
impose glycan {12} { {C1 O2 C2 H2 0.0} }
saveamberparm glycan glycan.prmtop glycan.restrt
```

3.5.2. Procedures for building a lipid using GLYCAM-06 parameters

The procedure described here allows a user to produce a single lipid molecule without consideration for axial alignment. Lipid bilayers are typically built in the (x,y) plane of a Cartesian coordinate system, which requires the individual lipids to be aligned hydrophilic “head” to hydrophobic “tail” along the z-axis. This can be done relatively easily by loading a template PDB file that has been appropriately aligned on the z-axis.

The lipid described in this example is 1,2-dimyristoyl-*sn*-glycero-3-phosphocholine or DMPC. For this example, DMPC will be composed of four fragments: CHO, the choline “head” group; PGL, the phospho-glycerol “head” group; MYR, the *sn*-1 chain myristic acid “tail” group; and MY2, the *sn*-2 chain myristic acid “tail” group. See the molecular diagram in 3.3 for atom labels (hydrogens and atomic charges are removed for clarity) and bonding points between each residue (dashed lines). This tutorial will use only prep files for each of the four fragments. These prep files were initially built as PDB files and formatted as prep files using *antechamber*. GLYCAM-compatible charges were added to the prep files and a prep file database (GLYCAM_lipids_06h.prep) was created containing all four files.

3.5.2.1. Example: Building a lipid with LEaP.

One need not load the main GLYCAM prep files in order to build a lipid using the GLYCAM-06 parameter set, but it is automatically loaded with the default *leaprc.GLYCAM_06h*. Note that the lipid generated by this set of commands is not necessarily aligned appropriately to create a bilayer along an axis. The commands to use are:

```
source leaprc.GLYCAM_06h # source the leaprc for GLYCAM-06
```

3. LEaP

```
loadamberprep GLYCAM_06_lipids.prep # load the lipid prep file
set CHO tail CHO.1.C5 # set the tail atom of CHO as C5.
set PGL head PGL.1.O1 # set the head atom of PGL to O1
set PGL tail PGL.1.C3 # set the tail atom of PGL to C3
lipid = sequence { CHO PGL MYR } # generate the straight-chain
# portion of the lipid
set lipid tail lipid.2.C2 # set the tail atom of PGL to C2
lipid = sequence { lipid MY2 } # add MY2 to the "lipid" unit
impose lipid {2} { {C1 C2 C3 O1 163} } # set torsions for
impose lipid {2} { {C2 C3 O1 C1 -180} } # PGL & MYR
impose lipid {2} { {C3 O1 C1 C2 180} }
impose lipid {2} { {O4 C1 C2 O1 -60} } # set torsions for
impose lipid {2} { {C1 C2 O1 C1 -180} } # PGL & MY2
impose lipid {2} { {C2 O1 C1 C2 180} }
# Note that the values here may not necessarily
# reflect the best choice of torsions.
savepdb lipid DMPC.pdb # save pdb file
saveamberparm lipid DMPC.top DMPC.crd # save top and crd files
```

3.5.3. Procedures for building a glycoprotein in LEaP.

The LEaP commands given in this section assume that you already have a PDB file containing a glycan and a protein in an appropriate relative configuration. Thorough knowledge of the commands in LEaP is required in order to successfully link any but the simplest glycans to the simplest proteins, and is beyond the scope of this discussion. Several options for generating the relevant PDB file are given below (see Items 5a-5c).

The protein employed in this example is bovine ribonuclease A (PDBID: 3RN3). Here the branched oligosaccharide assembled in the second example will be attached (*N*-linked) to ASN 34 to generate ribonuclease B.

3.5.3.1. Setting up protein pdb files for glycosylation in LEaP.

1. Delete any atoms with the “HETATM” card from the PDB file. These would typically include bound ligands, non-crystallographic water molecules and non-coordinating metal ions. Delete any hydrogen atoms if present.
2. In general, check the protein to make sure there are no duplicate atoms in the file. This can be quickly done by loading the protein in LEaP and checking for such warnings. In this particular example, residue 119 (HIS) contained duplicate side chain atoms. Delete all but one set of duplicate atoms.
3. Check for the presence of disulfide bonds (SSBOND) by looking at the header section of the PDB file. 3RN3 has four disulfide bonds, between the following pairs of cysteine residues: 26—84, 40—95, 58—110, and 65—72. Change the names of these eight cysteine residues from CYS to CYX.

4. At present, it is possible to link glycans to serine, threonine, hydroxyproline and asparagine. You must rename the amino acid in the protein PDB file manually prior to loading it into LEaP. The modified residue names are OLS (for *O*-linkages to SER), OLT (for *O*-linkages to THR), OLP (for *O*-linkages to hydroxyproline, HYP) and NLN (for *N*-linkages to ASN). Libraries containing amino acid residues that have been modified for the purpose are automatically loaded when *leaprc.GLYCAM_06h* is sourced. See the lists of library files in 2.9 for more information.
5. Prepare a PDB file containing the protein and the glycan, with the glycan correctly aligned relative to the protein surface. There are several approaches to performing this including:
 - a) It is often the case that one or more glycan residues are present in the experimental PDB file. In this case, a reasonable method is to superimpose the linking sugar residue in the GLYCAM-generated glycan upon that present in the experimental PDB file, and to then save the altered coordinates. If you use this method, remember to delete the experimental glycan from the PDB file! It is also essential to ensure that each carbohydrate residue is separated from other residues by a TER card in the PDB file. Also remember to delete the terminal OH or OMe from the glycan. Alternately, the experimental glycan may be retained in the PDB file, provided that it is renamed according to the GLYCAM 3-letter code, and that the atom names and order in the PDB file match the GLYCAM standard. This is tedious, but will work. Again, be sure to insert TER cards if they are missing between the protein and the carbohydrate and between the carbohydrate residues themselves.
 - b) Use a molecular modeling package to align the GLYCAM-generated glycan with the protein and save the coordinates in a single file. Remember to delete the terminal OH or OMe from the glycan.
 - c) Use the Glycoprotein Builder tool at <http://www.glycam.org>. This tool allows the user to upload protein coordinates, build a glycan (or select it from a library), and attach it to the protein. All necessary AMBER files may then be downloaded. This site is also convenient for preprocessing protein-only files for subsequent uploading to the glycoprotein builder.

3.5.3.2. Example: Adding a branched glycan to 3RN3 (*N*-linked glycosylation).

In this example we will assume that the glycan generated above (“branch.pdb”) has been aligned relative to the ASN 34 in the protein file and that the complex has been saved as a new PDB file (e.g., as “3rn3_nlink.pdb”). The last amino acid residue should be VAL 124, and the glycan should be present as 4YB 125, 4YB 126, VMB 127, OMA 128 and OMA 129.

Remember to change the name of ASN 34 from ASN to NLN. For the glycan structure, ensure that each residue in the PDB file is separated by a “TER” card. *The sequence command is not to be used here, and all linkages (within the glycan and to the protein) will be specified individually.*

Enter the following commands into *xleap* (or *tLeap* if a graphical representation is not desired). Alternately, copy the commands into a file to be sourced.

3. LEaP

```
source leaprc.GLYCAM_06h # load the GLYCAM-06 leaprc
source leaprc.ff12SB # load the (modified) ff12 force field
glyprot = loadpdb 3nr3_nlink.pdb # load protein and glycan pdb file
bond glyprot.125.O4 glyprot.126.C1 # make inter glycan bonds
bond glyprot.126.O4 glyprot.127.C1
bond glyprot.127.O6 glyprot.128.C1
bond glyprot.127.O3 glyprot.129.C1
bond glyprot.34.SG glyprot.125.C1 # make glycan -- protein bond
bond glyprot.26.SG glyprot.84.SG # make disulfide bonds
bond glyprot.40.SG glyprot.95.SG
bond glyprot.58.SG glyprot.110.SG
bond glyprot.65.SG glyprot.72.SG
addions glyprot Cl- 0 # neutralize appropriately
solvateBox glyprot TIP3P BOX 8 # solvate the solute
savepdb glyprot 3nr3_glycan.pdb # save pdb file
saveamberparm glyprot 3nr3_glycan.top 3nr3_glycan.crd # save top, crd
quit # exit leap
```

4. Antechamber and MCPB

These are a set of tools to generate files for organic molecules and for some metal centers in proteins, which can then be read into LEaP. The Antechamber suite was written by Junmei Wang, and is designed to be used in conjunction with the general AMBER force field (GAFF) (gaff.dat).[68] See Ref. [69] for an explanation of the algorithms used to classify atom and bond types, to assign charges, and to estimate force field parameters that may be missing in gaff.dat. The Metal Center Parameter Builder (MCPB) program was developed by Martin Peters [70], and is described in Section 4.6.

Like the traditional AMBER force fields, GAFF uses a simple harmonic function form for bonds and angles. Unlike the traditional AMBER force fields, atom types in GAFF are more general and cover most of the organic chemical space. In total there are 33 basic atom types and 22 special atom types. The charge methods used in GAFF can be HF/6-31G* RESP or AM1-BCC.[71, 72] All of the force field parametrization were carried out with HF/6-31G* RESP charges. However, in most cases, AM1-BCC, which was parametrized to reproduce HF/6-31G* RESP charges, is recommended in large-scale calculations because of its efficiency.

The van der Waals parameters are the same as those used by the traditional AMBER force fields. The equilibrium bond lengths and bond angles came from *ab initio* calculations at the MP2/6-31G* level and statistics derived from the Cambridge Structural Database. The force constants for bonds and angles were estimated using empirical models, and the parameters in these models were trained using the force field parameters in the traditional AMBER force fields. General torsional angle parameters were extensively applied in order to reduce the huge number of torsional angle parameters to be derived. The force constants and phase angles in the torsional angle parameters were optimized using our PARMSCAN package,[73] with an aim to reproduce the rotational profiles depicted by high-level *ab initio* calculations (geometry optimizations at the MP2/6-31G* level, followed by single point calculations at MP4/6-311G(d,p)).

By design, GAFF is a complete force field (so that missing parameters rarely occur); it covers almost all the organic chemical space that is made up of C, N, O, S, P, H, F, Cl, Br and I. Moreover, GAFF is totally compatible with the AMBER macromolecular force fields. It should be noted that GAFF atom types, except metal types, are in lower case, while AMBER atom types are always in upper case. This feature makes it possible to load both AMBER protein/nucleic acid force fields and GAFF without any conflict. One can even merge the two kinds of force fields into one file. The combined force fields are capable of studying complicated systems that include both proteins/nucleic acids and organic molecules. We believe that the combination of GAFF with AMBER macromolecular force fields will provide a useful molecular mechanical tool for rational drug design, especially in binding free energy calculations and molecular docking studies. Since its introduction, GAFF has been used for a wide range of applications, including ligand docking,[74] bilayer simulations,[75, 76] and the study of pure organic liquids [77].

4.1. Principal programs

The *antechamber* program itself is the main program of Antechamber. If your molecule falls into any of several fairly broad categories, *antechamber* should be able to process your PDB file directly, generating output files suitable for LEaP. Otherwise, you may provide an input file with connectivity information, i.e., in a format such as Mol2 or SDF. If there are missing parameters after *antechamber* is finished, you may want to run *parmchk* to generate a frcmod template that will assist you in generating the needed parameters.

4.1.1. antechamber

This is the most important program in the package. It can perform many file conversions, and can also assign atomic charges and atom types. As required by the input, *antechamber* executes the following programs: *sqm* (or, alternatively, *mopac* or *divcon*), *atomtype*, *am1bcc*, *bondtype*, *espgen*, *respden* and *prepgen*. It typically produces many intermediate files; these may be recognized by their names, in which all letters are upper-case. If you experience problems while running *antechamber*, you may want to run the individual programs that are described below.

Antechamber options:

```
-help print these instructions
-i input file name
-fi input file format
-o output file name
-fo output file format
-c charge method
-cf charge file name
-nc net molecular charge (int)
-a additional file name
-fa additional file format
-ao additional file operation
    crd : only read in coordinate
    crg: only read in charge
    name : only read in atom name
    type : only read in atom type
    bond : only read in bond type
-m multiplicity (2S+1), default is 1
-rn residue name, if not available in the input file
-rf residue topology file name in prep input file, default is molecule.res
-ch check file name in gaussian input file, default is molecule
-ek empirical calculation (mopac or sqm) keyword (in quotes)
-gk gaussian keyword in a pair of quotation marks
-gm gaussian assign memory, inside a pair of quotes, such as "%mem=1000MB"
-gn gaussian assign number of processor, inside a pair of quotes, such as "%nproc=8"
```



```

-df use divcon flag, 0 - use mopac; 2 - use sqm (the default)
-at atom type, can be gaff, amber, bcc and sybyl, default is gaff
-du check atom name duplications, can be yes(y) or no(n), default is yes
-j atom type and bond type prediction index, default is 4
    0 : no assignment
    1 : atom type
    2 : full bond types
    3 : part bond types
    4 : atom and full bond type
    5 : atom and part bond type
-eq equalize atomic charge, default is 1 for '-c resp' and '-c bcc'
    0 : no equalization
    1 : by atomic paths
    2 : by atomic paths and geometry, such as E/Z configurations
-s status information, can be 0 (brief), 1 (the default) and 2 (verbose)
-pf remove the intermediate files: can be yes (y) and no (n, default)
-i -o -fi and -fo must appear in command lines and the others are optional
Use 'antechamber -L' to list the supported file formats and charge methods

```

List of the File Formats:

file format type	abbre.	index	file format type	abbre.	index
Antechamber	ac	1	Sybyl Mol2	mol2	2
PDB	pdb	3	Modified PDB	mpdb	4
AMBER PREP (int)	prepi	5	AMBER PREP (car)	prepc	6
Gaussian Z-Matrix	gzmat	7	Gaussian Cartesian	gcrt	8
Mopac Internal	mopint	9	Mopac Cartesian	mopcrt	10
Gaussian Output	gout	11	Mopac Output	mopout	12
Alchemy	alc	13	CSD	csd	14
MDL	mdl	15	Hyper	hin	16
AMBER Restart	rst	17	Jaguar Cartesian	jcrt	18
Jaguar Z-Matrix	jzmat	19	Jaguar Output	jout	20
Divcon Input	divcrt	21	Divcon Output	divout	22
SQM Input	sqmcrt	23	SQM Output	sqmout	24
Charmm	charmm	25	Gaussian ESP	gesp	26

AMBER restart file can only be read in as additional file

List of the Charge Methods:

charge method	abbre.	index	charge method	abbre.	index
RESP	resp	1	AM1-BCC	bcc	2
CM1	cm1	3	CM2	cm2	4

4. Antechamber and MCPB

ESP (Kollman)	esp	5		Mulliken	mul	6
Gasteiger	gas	7		Read in charge	rc	8
Write out charge	wc	9		Delete Charge	dc	10

Examples:

```
(1) antechamber -i g98.out -fi gout -o sustiva_resp.mol2 -fo mol2 -c resp
(2) antechamber -i g98.out -fi gout -o sustiva_bcc.mol2 -fo mol2 -c bcc -j 5
(3) antechamber -i g98.out -fi gout -o sustiva_gas.mol2 -fo mol2 -c gas
(4) antechamber -i g98.out -fi gout -o sustiva_cm2.mol2 -fo mol2 -c cm2
(5) antechamber -i g98.out -fi gout -o sustiva.ac -fo ac
(6) antechamber -i sustiva.ac -fi ac -o sustiva.mpdb -fo mpdb
(7) antechamber -i sustiva.ac -fi ac -o sustiva.mol2 -fo mol2
(8) antechamber -i sustiva.mol2 -fi mol2 -o sustiva.gzmat -fo gzmat
(9) antechamber -i sustiva.ac -fi ac -o sustiva_gas.ac -fo ac -c gas
(10) antechamber -i mtz.pdb -fi pdb -o mtz.mol2 -fo mol2 -c rc -cf mtz.charge
(11) antechamber -i g03.out -fi gout -o mtz.mol2 -fo mol2 -c resp
    -a mtz.pdb -fa pdb -ao name
(12) antechamber -i ch3I.mol2 -fi mol2 -o gcrt -fo gcrt -gv 1 -ge ch3I.gesp
(13) antechamber -i acetamide.out -fi gout -o acetamide_eq0.mol2 -fo mol2
    -c resp -eq 0
(14) antechamber -i acetamide.out -fi gout -o acetamide_eq0.mol2 -fo mol2
    -c resp -eq 1 (15) antechamber -i acetamide.out -fi gout
    -o acetamide_eq0.mol2 -fo mol2 -c resp -eq 2
```

The following is the detailed explanations of some flags

-nc This flag specifies the net charge of the input molecule, otherwise, the net charge is read in from the input directly (such as gout, mopout, sqmout, sqmcrt, gcrt, etc.) or calculated by summing the partial charges (such as mol2, prepri, etc).

-a,-fa,-ao Sometimes, one wants to read additional information from another file other than the input, the '-ao' flag informs the program to read in which information from the additional file specified with '-a' flag. In Example (11), a mol2 file is generated from a Gaussian output file with atom names read in from a pdb file.

-ch,-gk,-gm,-gn Those flags specify the keywords and resource usage in Gaussian calculations

-ge,-gv The '-ge' flag specifies the file name of gesp file generated using iop(6/50=1) with Gaussian 09; the -gv flag specifies the Gaussian version and the default is '1' for Gaussian 09. If one wants to generate Gaussian input files (gcrt and gzmat) for older Gaussian versions, '-gv' must be set to '0'.

-rn The '-rn' line specifies the residue name to be used; thus, it must be one to three characters long.

- at** This flag is used to specify whether atom types are to be created for the GAFF force field or for atom types consistent with parm94.dat and parm99.dat (i.e., the AMBER force fields). If you are using *antechamber* to create a modified residue for use with the standard AMBER parm94/parm99 force fields, you should set this flag to “amber”; if you are looking at a more arbitrary molecule, set it to “gaff”, even if the molecule is intended for use as a ligand bound to a macromolecule described by the AMBER force fields.
- j** This flag instructs the program how to run ‘bondtype’ and ‘atom type’. ‘-j 1’ assumes the bond types already exists; ‘-j 4’ first predicts the connectivity table, then assigns bond and atom types sequentially; ‘-j 5’ reads in connectivity table from the input and then run ‘bondtype’ and ‘atomtype’ sequentially. In most situations, ‘-j 4’, the default option, is recommended. However, ‘-j 5’ should be used if the input structure is not good enough and it includes the bond connectivity information (such as mol2, mdl, gzmat, etc.)
- eq** This flag specifies how to do charge equilibration. With ‘-eq 1’, atomic charge equilibration is predicted only by atom paths, in another word, if two or more atoms have exactly same sets of atom paths, they are equivalent and their charges are forced to be same. While ‘-eq 2’ predicts charge equilibration using both atom paths and some geometrical information (E/Z configuration). With the ‘-eq 2’ option, the charges of two hydrogen atoms bonded to the No 2 carbon of chloroethene are different as they adopt different configurations to chlorine (one is cis and the other is trans). Similarly, the two amide hydrogen atoms of acetamide do not share the same partial charge as the amide bond cannot rotate freely. To back-compatible to the older versions, the default is set to ‘1’

In Example (12), a gcrt file of iodine methane is generated and a gesp file named ch3I.gesp is produced when running Gaussian 09 with the default keyword. In Examples (13-15), RESP charges are generated for acetamide using different charge equilibration options. In the following table, the charges are listed for comparison purposes.

atom names	eq = 0 no equalization	eq = 1 atomic paths	eq = 2 + geometry
-----	-----	-----	-----
methyl carbon	-0.5190	-0.5516	-0.5193
methyl hydrogen	0.1412/0.1380/0.1396	0.1470	0.1397
carbonyl carbon	0.9673	0.9786	0.9673
oxygen	-0.6468	-0.6463	-0.6468
nitrogen	-1.1189	-1.1219	-1.1189
amide hydrogen	0.4556/0.4429	0.4501	0.4556/0.4429
-----	-----	-----	-----

4.1.2. parmchk

parmchk reads in an ac file as well as a force field file (the default is \$AMBERHOME/dat/leap/parm/gaff.dat). It writes out a force field modification (frcmod) file containing any force field parameters that are needed for the molecule but not supplied by the

4. Antechamber and MCPB

force field (*.dat) file. Problematic parameters, if any, are indicated in the frcmod file with the note, “ATTN, need revision”, and are typically given values of zero. This can cause fatal terminations of programs that later use a resulting prmtop file; for example, a zero value for the periodicity of the torsional barrier of a dihedral parameter will be fatal in many cases. For each atom type, an atom type corresponding file (ATCOR.DAT) lists its replaceable general atom types. By default, only the missing parameters are written to the frcmod file. When the “-a” switch is given the value “Y”, *parmchk* prints out all force field parameters used by the input molecule, whether they are already in the parm file or not. This file can be used to prepare the frcmod file used by thermodynamic integration calculations using sander.

```
parmchk -i input file name
        -o frcmod file name
        -f input file format (prepi, ac ,mol2)
        -p ff parmfile
        -c atom type corresponding file, default is ATCOR.DAT
        -a print out all force field parameters including those in the parmfile
           can be 'Y' (yes) or 'N' (no), default is 'N'
        -w print out parameters that matching improper dihedral parameters
           that contain 'X' in the force field parameter file, can be 'Y' (yes)
           or 'N' (no), default is 'Y'
```

Example:

```
parmchk -i sustiva.prep -f prepi -o frcmod
```

This command reads in *sustiva.prep* and finds the missing force field parameters listed in *frcmod*.

4.2. A simple example for antechamber

The most common use of the antechamber program suite is to prepare input files for LEaP, starting from a three-dimensional structure, as found in a PDB file. The antechamber suite automates the process of developing a charge model and assigning atom types, and partially automates the process of developing parameters for the various combinations of atom types found in the molecule.

As with any automated procedure, the output should be carefully examined, and users should be on the lookout for any unusual or incorrect program behavior.

Suppose you have a PDB-format file for your ligand, say thiophenol, which looks like this:

ATOM	1	CG	TP	1	-1.959	0.102	0.795
ATOM	2	CD1	TP	1	-1.249	0.602	-0.303
ATOM	3	CD2	TP	1	-2.071	0.865	1.963
ATOM	4	CE1	TP	1	-0.646	1.863	-0.234
ATOM	5	C6	TP	1	-1.472	2.129	2.031
ATOM	6	CZ	TP	1	-0.759	2.627	0.934

4.2. A simple example for antechamber

```

ATOM      7  HE2  TP      1      -1.558   2.719   2.931
ATOM      8  S15  TP      1      -2.782   0.365   3.060
ATOM      9  H19  TP      1      -3.541   0.979   3.274
ATOM     10  H29  TP      1      -0.787  -0.043  -0.938
ATOM     11  H30  TP      1       0.373   2.045  -0.784
ATOM     12  H31  TP      1      -0.092   3.578   0.781
ATOM     13  H32  TP      1      -2.379  -0.916   0.901

```

(This file may be found at `$AMBERHOME/AmberTools/test/antechamber/tp/tp.pdb`). The basic command to create a mol2 file for LEaP is just:

```
antechamber -i tp.pdb -fi pdb -o tp.mol2 -fo mol2 -c bcc
```

The output file will look like this:

```

@<TRIPOS>MOLECULE
TP
    13    13    1    0    0
SMALL
bcc
@<TRIPOS>ATOM
    1 CG      -1.9590   0.1020   0.7950 ca    1 TP   -0.132000
    2 CD1      -1.2490   0.6020  -0.3030 ca    1 TP   -0.113000
    3 CD2      -2.0710   0.8650   1.9630 ca    1 TP    0.015900
    4 CE1      -0.6460   1.8630  -0.2340 ca    1 TP   -0.137000
    5 C6       -1.4720   2.1290   2.0310 ca    1 TP   -0.132000
    6 CZ       -0.7590   2.6270   0.9340 ca    1 TP   -0.113000
    7 HE2      -1.5580   2.7190   2.9310 ha    1 TP    0.136500
    8 S15      -2.7820   0.3650   3.0600 sh    1 TP   -0.254700
    9 H19      -3.5410   0.9790   3.2740 hs    1 TP    0.190800
   10 H29      -0.7870  -0.0430  -0.9380 ha    1 TP    0.133500
   11 H30       0.3730   2.0450  -0.7840 ha    1 TP    0.134000
   12 H31      -0.0920   3.5780   0.7810 ha    1 TP    0.133500
   13 H32      -2.3790  -0.9160   0.9010 ha    1 TP    0.136500

@<TRIPOS>BOND
    1    1    2 ar
    2    1    3 ar
    3    1   13 1
    4    2    4 ar
    5    2   10 1
    6    3    5 ar
    7    3    8 1
    8    4    6 ar
    9    4   11 1
   10    5    6 ar
   11    5    7 1

```

4. Antechamber and MCPB

```
12      6      12 1
13      8      9 1
@<TRIPOS>SUBSTRUCTURE
1 TP          1 TEMP          0 ****  ****  0 ROOT
```

This command says that the input format is pdb, output format is Sybyl mol2, and the BCC charge model is to be used. The output file is shown in the box titled .mol2. The format of this file is a common one understood by many programs. However, to display molecules properly in software packages other than LEaP and gleap, one needs to assign atom types using the '-at sybyl' flag rather than using the default gaff atom types.

You can now run parmchk to see if all of the needed force field parameters are available:

```
parmchk -i tp.mol2 -f mol2 -o frcmod
```

This yields the frcmod file:

```
remark goes here
MASS
BOND
ANGLE
DIHE
IMPROPER
ca-ca-ca-ha          1.1          180.0          2.0          General improper \
                                torsional angle (2 general atom types)
ca-ca-ca-sh          1.1          180.0          2.0          Using default value
NONBON
```

In this case, there were two missing dihedral parameters from the gaff.dat file, which were assigned a default value. (As gaff.dat continues to be developed, there should be fewer and fewer missing parameters to be estimated by parmchk.) In rare cases, parmchk may be unable to make a good estimate; it will then insert a placeholder (with zeros everywhere) into the frcmod file, with the comment "ATTN: needs revision". After manually editing this to take care of the elements that "need revision", you are ready to read this residue into LEaP, either as a residue on its own, or as part of a larger system. The following LEaP input file (leap.in) will just create a system with thiophenol in it:

```
source leaprc.gaff
mods = loadAmberParams frcmod
TP = loadMol2 tp.mol2
saveAmberParm TP prmtop inpcrd
quit
```

You can read this into LEaP as follows:

```
tleap -s -f leap.in
```

This will yield a prmtop and inpcrd file. If you want to use this residue in the context of a larger system, you can insert commands after the loadAmberPrep step to construct the system you want, using standard LEaP commands.

In this respect, it is worth noting that the atom types in gaff.dat are all lower-case, whereas the atom types in the standard AMBER force fields are all upper-case. This means that you can load both gaff.dat and (say) parm99.dat into LEaP at the same time, and there won't be any conflicts. Hence, it is generally expected that you will use one of the AMBER force fields to describe your protein or nucleic acid, and the gaff.dat parameters to describe your ligand; as mentioned above, gaff.dat has been designed with this in mind, i.e., to produce molecular mechanics descriptions that are generally compatible with the AMBER macromolecular force fields.

The procedure above only works as it stands for neutral molecules. If your molecule is charged, you need to set the -nc flag in the initial antechamber run. Also note that this procedure depends heavily upon the initial 3D structure: it must have all hydrogens present, and the charges computed are those for the conformation you provide, after minimization in the AM1 Hamiltonian. In fact, this means that you must have an reasonable all-atom initial model of your molecule (so that it can be minimized with the AM1 Hamiltonian), and you may need to specify what its net charge is, especially for those molecular formats that have no net charge information, and no partial charges or the partial charges in the input are not correct. The system should really be a closed-shell molecule, since all of the atom-typing rules assume this implicitly.

Further examples of using antechamber to create force field parameters can be found in the `$AMBERHOME/test/antechamber` directory. Here are some practical tips from Junmei Wang:

1. For the input molecules, make sure there are no open valences and the structures are reasonable.
2. The Antechamber package produces two kinds of messages, error messages and informative messages. You may safely ignore those message starting with "Info". For example: "Info: Bond types are assigned for valence state 1 with penalty of 1".
3. Failures are most often produced when antechamber infers an incorrect connectivity. In such cases, you can revise by hand the connectivity information in "ac" or "mol2" files. Systematic errors could be corrected by revising the parameters in `$AMBERHOME/-dat/antechamber/CONNECT.TPL`.
4. It is a good idea to check the intermediate files in case of a program failure, and you can run separate programs one by one. Use the "-s 2" flag to antechamber to see details of what it is doing.
5. Beginning with Amber 10, a new program called *acdoctor* is provided to diagnose possible problem of an input molecule. If you encounter failure when running antechamber programs, it is highly recommended to let *acdoctor* perform a diagnosis.
6. By default, the AM1 Mulliken charges that are required for the AM1-BCC procedure are computed using the *sqm* program, with the following keyword (which is placed inside the *&qmmm* namelist):

```
qm_theory='AM1', grms_tol=0.0002, tight_p_conv=1, scfconv=1.d-10,
```

4. Antechamber and MCPB

For some molecules, especially if they have bad starting geometries, convergence to these tight criteria may not be obtained. If you have trouble, examine the *sqm.out* file, and try changing *scfconv* to 1.d-8 and/or *tight_p_conv* to 0. You may also need to increase the value of *grms_tol*. You can use the *-ek* flag to antechamber to change these, or just manually edit the *sqm.in* file. But be aware that there may be something “wrong” with your molecule if these problems arise; the *acdoctor* program may help.

4.3. Programs called by antechamber

The following programs are automatically called by antechamber when needed. Generally, you should not need to run them yourself, unless problems arise and/or you want to fine-tune what antechamber does.

4.3.1. atomtype

Atomtype reads in an ac file and assigns the atom types. You may find the default definition files in \$AMBERHOME/dat/antechamber: ATOMTYPE_AMBER.DEF (AMBER), ATOMTYPE_GFF.DEF (general AMBER force field). ATOMTYPE_GFF.DEF is the default definition file. It is pointed out that the usage of atomtype is not limited to assign force field atom types, it can also be used to assign atom types in other applications, such as QSAR and QSPR studies. The users can define their own atom type definition files according to certain rules described in the above mentioned files.

```
atomtype -i input file name
         -o output file name (ac)
         -f input file format(ac (the default) or mol2)
         -p amber or gaff or bcc or gas, it is suppressed by "-d" option
         -d atom type definition file, optional
         -a do post atom type adjustment when '-d' is used
           1: yes, 0: no (the default)
```

Example:

```
atomtype -i sustiva_resp.ac -o sustiva_resp_at.ac -f ac -p amber
```

This command assigns atom types for sustiva_resp.ac with amber atom type definitions. The output file name is sustiva_resp_at.ac

4.3.2. am1bcc

Am1bcc first reads in an ac or mol2 file with or without assigned AM1-BCC atom types and bond types. Then the bcc parameter file (the default, BCCPARAM.DAT is in \$AMBERHOME/dat/antechamber) is read in. An ac file with AM1-BCC charges [71, 72] is written out. Be sure the charges in the input ac file are AM1-Mulliken charges.


```

amlbcc -i input file name in ac format
       -o output file name
       -f output file format(pdb or ac, optional, default is ac)
       -p bcc parm file name (optional))
       -j atom and bond type judge option, default is 0)
           0: No judgement
           1: Atom type
           2: Full bond type
           3: Partial bond type
           4: Atom and full bond type
           5: Atom and partial bond type

```

Example:

```
amlbcc -i comp1.ac -o comp1_bcc.ac -f ac -j 4
```

This command reads in comp1.ac, assigns both atom types and bond types and finally performs bond charge correction to get AM1-BCC charges. The '-j' option of 4, which is the default, means that both the atom and bond type information in the input file is ignored and a full atom and bond type assignments are performed. The '-j' option of 3 and 5 implies that bond type information (single bond, double bond, triple bond and aromatic bond) is read in and only a bond type adjustment is performed. If the input file is in mol2 format that contains the basic bond type information, option of 5 is highly recommended. comp1_bcc.ac is an ac file with the final AM1-BCC charges.

4.3.3. bondtype

bondtype is a program to assign six bond types based upon the read in simple bond types from an ac or mol2 format with a flag of "-j part" or purely connectivity table using a flag of "-j full". The six bond types as defined in AM1-BCC [71, 72] are single bond, double bond, triple bond, aromatic single, aromatic double bonds and delocalized bond. This program takes an ac file or mol2 file as input and write out an ac file with the predicted bond types. After the continually improved algorithm and code, the current version of bondtype can correctly assign bond types for most organic molecules (>99% overall and >95% for charged molecules) in our tests.

Starting with Amber 10, bond type assignment is proceeded based upon residues. The bonds that link two residues are assumed to be single bonded. This feature allows antechamber to handle residue-based molecules, even proteins are possible. It also provides a remedy for some molecules that would otherwise fail: it can be helpful to dissect the whole molecule into residues. Some molecules have more than one way to assign bond types; for example, there are two ways to alternate single and double bonds for benzene. The assignment adopted by bondtype is purely affected by the atom sequence order. To get assignments for other resonant structures, one may freeze some bond types in an *ac* or *mol2* input file (appending 'F' or 'f' to the corresponding bond types). Those frozen bond types are ignored in the bond type assignment procedure. If the input molecules contain some unusual elements, such as metals, the involved bonds are automatically frozen. This frozen bond feature enables bondtype to handle unusual molecules in a practical way without simply producing an error message.

4. Antechamber and MCPB

```
bondtype -i input file name
         -o output file name
         -f input file format (ac or mol2)
         -j judge bond type level option, default is part
           full full judgment
           part partial judgment, only do reassignment according
             to known bond type information in the input file
```

Example:

```
#!/bin/csh -fv
set mols = `ls *.ac`
foreach mol ($mols)
  set mol_dir = $mol:r
  antechamber -i $mol_dir.ac -fi ac -fo ac -o $mol_dir.ac -c mul
  bondtype -i $mol_dir.ac -f ac -o $mol_dir.dat -j full
  amlbcc -i $mol_dir.dat -o $mol_dir\_bcc.ac -f ac -j 0
end
exit(0)
```

The above script finds all the files with the extension of "ac", calculates the Mulliken charges using antechamber, and predicts the atom and bond types with bondtype. Finally, AM1-BCC charges are generated by running amlbcc to do the bond charge correction. More examples are provided in *\$AMBERHOME/test/antechamber/bondtype* and *\$AMBERHOME/test/antechamber/chemokine*.

4.3.4. prepgen

Prepgen generates the prep input file from an ac file. By default, the program generates a mainchain itself. However, you may also specify the main-chain atoms in the main chain file. From this file, you can also specify which atoms will be deleted, and whether to do charge correction or not. In order to generate the amino-acid-like residue (this kind of residue has one head atom and one tail atom to be connected to other residues), you need a main chain file. Sample main chain files are in *\$AMBERHOME/dat/antechamber*.

```
prepgen -i input file name(ac)
        -o output file name
        -f output file format (car or int, default: int)
        -m mainchain file name
        -rn residue name (default: MOL)
        -rf residue file name (default: molecule.res)
        -f -m -rn -rf are optional
```

Examples:

```
prepgen -i sustiva.ac -o sustiva_int.prep -f int -rn SUS -rf SUS.res
```

```

prepgen -i sustiva.ac -o sustiva_car.prep -f car -rn SUS -rf SUS.res
prepgen -i sustiva.ac -o sustiva_int_main.prep -f int -rn SUS
        -rf SUS.res -m mainchain_sus.dat
prepgen -i ala_cm2_at.ac -o ala_cm2_int_main.prep -f int -rn ALA
        -rf ala.res -m mainchain_ala.dat

```

The above commands generate different kinds of prep input files with and without specifying a main chain file.

4.3.5. espngen

Espngen reads in a gaussian (92,94,98,03) output file and extracts the ESP information. An esp file for the resp program is generated.

```

espngen -i  input file name
        -o  output file name

```

Example:

```

(1) espngen -i sustiva_g98.out -o sustiva.esp
(2) espngen -i ch3I.gesp -o ch3I.esp

```

Command (1) reads in sustiva_g98.out and writes out sustiva.esp, which can be used by the resp program. Command (2) reads in a gesp file generated by Gaussian 09 and outputs the esp file. Note that this program replaces shell scripts formerly found on the AMBER web site that perform equivalent tasks.

4.3.6. respgen

Respgen generates the input files for two-stage resp fitting. Starting with Amber 10, the program supports a single molecule with one or multiple conformations RESP fittings. Atom equivalence is recognized automatically. Frozen charges and charge groups are read in with '-a' flag. If there are some frozen charges in the additional input data file, a RESP charge file, QIN is generated as well. Here are flags to *respgen*:

```

-i input file name(ac)
-o output file name
-l maximum path length (default is -1, i.e. the path can be any long)
-f output file format
    resp1 - first stage resp fitting
    resp2 - second stage resp fitting
    iresp1 - first stage i_resp fitting
    iresp2 - second stage i_resp fitting
-e equalizing atomic charge (default is 1)
    0 not use
    1 by atomic paths
    2 by atomic paths and geometry (such as E/Z configuration)

```

4. Antechamber and MCPB

```
-a additional input data (predefined charges, atom groups etc)
-n number of conformations (default is 1)
-w weight of charge constraint
    the default values are 0.0005 for resp1/iresp1 and 0.001 for
    resp2/iresp2
```

The following is a sample of additional respgen input file

```
//predefined charges in a format of (CHARGE partial_charge atom_ID atom_name)
CHARGE -0.417500 7 N1
CHARGE 0.271900 8 H4
CHARGE 0.597300 15 C5
CHARGE -0.567900 16 O2
//charge groups in a format of (GROUP num_atom net_charge),
//more than one group may be defined.
GROUP 10 0.00000
//atoms in the group in a format of (ATOM atom_ID atom_name)
ATOM 7 N1
ATOM 8 H4
ATOM 9 C3
ATOM 10 H5
ATOM 11 C4
ATOM 12 H6
ATOM 13 H7
ATOM 14 H8
ATOM 15 C5
ATOM 16 O2
```

Example:

```
respfn -i sustiva.ac -o sustiva.respin1 -f resp1
respfn -i sustiva.ac -o sustiva.respin2 -f resp2
resp -O -i sustiva.respin1 -o sustiva.respout1 -e sustiva.esp -t qout_stage1
resp -O -i sustiva.respin2 -o sustiva.respout2 -e sustiva.esp
    -q qout_stage1 -t qout_stage2
antechamber -i sustiva.ac -fi ac -o sustiva_resp.ac -fo ac -c rc -cf qout_stage2
respfn -i acetamide.ac -o acetamide.respin1 -f resp1 -e 2
respfn -i acetamide.ac -o acetamide.respin2 -f resp2 -e 2
```

The above commands first generate the input files (sustiva.respin1 and sustiva.respin2) for resp fitting, then do two-stage resp fitting and finally use antechamber to read in the resp charges and write out an ac file, *sustiva_resp.ac*. A more complicated example has been provided in *\$AMBERHOME/test/antechamber/residuegen*. The last two 'respfn' commands generate resp input files for acetamide discriminating the two amide hydrogen atoms.

4.4. Miscellaneous programs

The Antechamber suite also contains some utility programs that perform various tasks in molecular mechanical calculations. They are listed in alphabetical order.

4.4.1. acdoctor

Acdoctor reads in all kinds of file formats applied in the *antechamber* program and 'diagnose' possible reasons that cause antechamber failure. Molecular format is first checked for some commonly-used molecular formats, such as pdb, mol2, mdl (sdf), etc. Then unusual elements (elements other than C, O, N, S, P, H, F, Cl, Br and I) are checked for all the formats. Unfilled valence is checked when atom types and/or bond types are read in. Those file formats include ac, mol2, sdf, prepi, prepc, mdl, alc and hin. *Acdoctor* also applies a more stringent criterion than that utilized by *antechamber* to determine whether a bond is formed or not. A warning message is printed out for those bonds that fail to meet the standard. Then *acdoctor* diagnoses if all atoms are linked together through atomic paths. If not, an error message is printed out. This kind of errors typically imply that the input molecule has one or several bonds missing. Finally, *acdoctor* tries to assign bond types and atom types for the input molecule. If no error occurs during running *bondtype* and *atomtype*, presumably the input molecule should be free from problems when running the other Antechamber programs. It is recommended to diagnose your molecules with *acdoctor* when you encounter Antechamber failures.

```
Usage: acdoctor -i input file name
          -f input file format
```

Example:

```
acdoctor -i test.mol2 -f mol2
```

The program reads in test.mol2 and checks the potential problem when running the Antechamber programs. Errors and warning message are printed out. (Possible file formats are listed above in Section 4.1.1.

4.4.2. parmcal

parmcal is an interactive program to calculate the bond length and bond angle parameters, according to the rules outlined in Ref. [68].

```
Please select:
1. calculate the bond length parameter: A-B
2. calculate the bond angle parameter: A-B-C
3. exit
```

4.4.3. residuegen

It can be painful to prepare an amino-acid-like residues. In Amber 10 and later versions, the program *residuegen* has been included. It facilitates residue topology generation. *residuegen*

4. Antechamber and MCPB

reads in an input file and applies a set of antechamber programs to generate residue topologies in prep format. The program can be applied to generate amino-acid-like topologies for amino acids, nucleic acids and other polymers as well. An example is provided below and the file format of the input file is also explained.

Usage: `residuegen input_file`

Example:

`residuegen ala.input`

This command reads in ala.input and generate residue topology for alanine. The file format of ala.input is explained below.

```
#INPUT_FILE:      structure file in ac format, generated from a Gaussian output
INPUT_FILE        ala.ac
#CONF_NUM:        Number of conformations utilized
CONF_NUM          2
#ESP_FILE:        esp file generated from gaussian output with 'espgen'
#                 for multiple conformations, cat all CONF_NUM esp files onto ESP_FILE
ESP_FILE          ala.esp
#SEP_BOND:        bonds that separate residue and caps, input in a format of
#                 (Atom_Name1 Atom_Name2), where Atom_Name1 belongs to residue and
#                 Atom_Name2 belongs to a cap; must show up two times
SEP_BOND          N1 C2
SEP_BOND          C5 N2
#NET_CHARGE:      net charge of the residue
NET_CHARGE        0
#ATOM_CHARGE:     predefined atom charge, input in a format of
#                 (Atom_Name Partial_Charge); can show up multiple times.
ATOM_CHARGE       N1 -0.4175
ATOM_CHARGE       H4 0.2719
ATOM_CHARGE       C5 0.5973
ATOM_CHARGE       O2 -0.5679
#PREP_FILE:       prep file name
PREP_FILE         ala.prep
#RESIDUE_FILE_NAME: residue file name in PREP_FILE
RESIDUE_FILE_NAME ala.res
#RESIDUE_SYMBOL:  residue symbol in PREP_FILE
RESIDUE_SYMBOL    ALA
```

4.5. New Development of Antechamber And GAFF

One important of functions of Antechamber is to assign AM1-BCC charges for organic molecules. Openeye's Quacpak module can also assign AM1-BCC charges. The careful users may find that the charges assigned by the two programs are only marginally different (the largest charge difference is smaller than 0.05) in most cases. The difference is probably rooted from

the difference of AM1 Mulliken charges. In unusual cases, large discrepancy occurs (the largest charge difference is larger than 0.1). Recently, we have systematically studied 585 marketed drugs using the both packages and the result is presented below. As the general AMBER force field is tightly related to the antechamber package, the new development of the GAFF is also summarized here.

4.5.1. Extensive Test of AM1-BCC Charges

Three methods, namely Antechamber/Mopac (Mulliken charges are calculated by Mopac), Antechamber/Sqm (Mulliken charges are calculated by sqm) and Openeye's Quacpak have been applied to assign the AM1-BCC charges for the 585 drug molecules. The first two methods give essentially similar charges for all the cases and the average charge difference is 0.005. The Quacpak, on the other hand, has an average charge difference of 0.015 to Antechamber/Mopac. When compared to RESP charges, the average charge differences are 0.102 and 0.105 for Antechamber/Mopac and Quacpak, respectively. In AM1-BCC, five BCC parameters were adjusted in order to improve agreement with the experimental free energies of solvation. Adjustments were made to bonds of amine nitrogen-H and amine nitrogen-tetravalent carbon.[71, 72] As a consequence, the average largest charge differences between AM1-BCC and RESP charges are very big: 0.441 for Antechamber/Mopac and 0.452 for Quacpak.

There are 71 molecules (12%) having the largest charge difference larger than 0.1 between Antechamber/Mopac and Quacpak. In comparison with the RESP charges, the average charge differences of the 71 molecules are 0.107 and 0.129 for Antechamber/Mopac and Quacpak, respectively. As to the average largest charge differences, the corresponding values are 0.444 and 0.522. It is clearly that Antechamber/Mopac-bcc has a similar average charge differences to RESP for the whole data set and the 71-molecule subset (0.102 vs 0.107), in contrast, Quacpak has a much larger average charge difference for the 71 molecules (0.129) than that of the whole data set (0.105). The similar trend is observed for the average largest charge difference as well (0.441 vs 0.444 for Antechamber/Mopac and 0.452 vs 0.522 for Quacpak).

4.5.2. New Development of GAFF

We have modified some parameters according to users' feedback. We would like to thank users who provide us nice feedback/suggestion, especially David Mobley and Gabriel Rocklin. This version (GAFF1.4) is a meta-version between gaff1.0 and gaff2.0 and the following is the major changes:

1. All the sp² carbon in a AR2 ring (such as pyrrole, furan, pyrazole) are either 'cc' or 'cd' atom types (not 'c2' any more). This is suggested by Gabriel Rocklin from UCSF. This modification improves the planarity of multiple-ring systems
2. New van der Waals parameters have been developed for 'br' and 'i' atom types. The current parameters can well reproduce the experimental density data of CH₃Br (1.6755, 20 degree) and CH₃I (2.2789, 20 degree): 1.642 for CH₃Br and 2.25 for CH₃I, in contrast, the old parameters give 1.31 and 1.84, respectively.[77]
3. New van der Waals parameters have been suggested by David Mobley for 'c1', 'cg' and 'ch' atom types.[78]

4. Antechamber and MCPB

4. We have performed B3LYP/6-31G* optimization for 15 thousands marketed or experimental drugs/bio-actives. Reliable bond length and bond angle equilibrium parameters were obtained by statistics: each bond length parameter must show up at least five times and has a rmsd smaller than 0.02 Å; each bond angle parameter must show up at least five times and has a rmsd smaller than 2.5 degrees. Those new parameters not showing up in old gaff were directly added into gaff 1.4; and some low-quality gaff parameters which show up less than five times or have large rmsd values (>0.02 Å for bond length and >5 degrees for bond angles) were replaced with those newly generated. In summary, 59 low quality bond stretching parameters were replaced and 56 new parameters were introduced; 437 low quality bond bending parameters were replaced and 618 new parameters were introduced.

4.6. Metal Center Parameter Builder (MCPB)

4.6.1. Introduction

The Metal Center Parameter Builder (MCPB) program provides a means to rapidly build, prototype, and validate MM models of metalloproteins. It uses the bonded plus electrostatics model to expand existing pairwise additive force fields. It was developed by Martin Peters at the University of Florida in the lab of Kenneth Merz Jr. MCPB is described fully Ref. [70].

Why is it desirable to model metalloprotein systems using MM models and more precisely within the bonded plus electrostatic model? Structure/function and dynamics questions that are not currently attainable using QM or QM/MM based methods due to unavailability of parameters or system size can be answered. Force fields have been developed for zinc, copper, nickel, iron and platinum containing systems using the bonded plus electrostatics model.

Incorporating metals into protein force fields can seem a daunting task due to the plethora of QM Hamiltonians, basis sets and charge models to choose from which the parameters are created. It was also generally carried out by hand without extensive validation for specific metalloproteins. MCPB was developed to remove the latter and create a framework in which to test various methods, basis sets and charges models in the creation of metalloprotein force fields.

The MCPB program was built using the MTK++ Application Program Interface (API). For more information regarding MTK++ and MCPB please see the MTK++ manual: *\$AMBER-HOME/doc/MTKpp.pdf* A more extensive description of metalloproteins and the theory within MCPB can be found in sections 10 to 12 of the MTK++ manual.

4.6.2. Running MCPB

MCPB takes two command-line arguments. One is the control file, which is required and chosen with the -i flag. The other is the log file, which is optional and chosen with the -l flag. A full listing of all the commands used by MCPB can be obtained with the -f flag.

```
MCPB: Semi-automated tool for metalloprotein parametrization
usage: MCPB [flags] [options]
options:
```


4.6. Metal Center Parameter Builder (MCPB)

```
-i script file
-l log file
flags:
-h help
-f function list
```

Full details of a metalloprotein parametrization procedure using MCPB can be found in section 15.10 of the user manual. This example describes the active site parametrization of a di-zinc system (PDB ID: 1AMP). The parametrization is broken down into stages since several MCPB operations rely on the output of external packages such as Gaussian and RESP. Most of the steps are carried out using MCPB but some require user input and instruction.

5. amberlite: Some AMBER-Tools-Based Utilities

Romain M. Wolf

Novartis Institutes for Biomedical Research, NIBR, Basle, Switzerland

AMBER "Lite" is a small set of utilities making use of the free AMBER Tools package (currently for version 1.4 or higher). The main focus is on the preparation of files for MM(GB)(PB)/SA-type simulations. The utilities can be used as delivered or they can serve as a starting point for further development. Examples are included to illustrate the concepts or to test the correct functioning of the installation. The text also contains a (very) condensed introduction to some AMBER file preparation concepts.

The AMBER Lite package (© Novartis Institutes for Biomedical Research, Basel, Switzerland) is free software under the GNU General Public License (GPL), as are the parts on which the package builds, namely the Amber tools *ptraj*, *leap*, *antechamber*, *sqm*, *pbsa* and the *NAB* package.

Users are free to modify the tools according to their needs. Strange or obviously wrong behavior should be communicated to the author (at romain.wolf@novartis.com or romain.wolf@gmail.com). Feedback (positive or negative) is welcome although I cannot guarantee continuous support. I will do my best to answer questions, correct bugs, or add features if they seem useful and if my time allows it.

5.1. Introduction

For many standard simulation tasks, only a limited number of tools within the AMBER package are required. Furthermore, the full set of routines can be confusing for new or casual users. The constantly enhanced and updated AMBER tutorials certainly offer an excellent entry point. The set of tools described hereafter should present another initiation, based entirely on the freely available portions of AMBER code. The emphasis in the AMBER Lite tools is on the MM(GB)(PB)/SA approach to compute (relative) free energies of interaction between ligands and receptors, a major task in structure-based drug discovery. The tools are simple enough to be understood, modified, and enhanced.

One section (Appendix 5.8.1) is dedicated to the preparation of PDB files prior to use them with AMBER. In my own experience, this is a critical part in setting up simulations. Scanning through the AMBER Mail Reflector, I find many reported problems and questions which originate from "bad" or badly prepared PDB starting files.

Another section (Appendix 5.9) gives a brief introduction on AMBER "masks" and NAB "atom expressions", used to select parts of molecular structures. Users should also read in detail

5. *amberlite*: Some AMBER-Tools-Based Utilities

the corresponding information in original AMBER documentations. Wrong partial selections are tricky because they may often go unnoticed, i.e., everything seems to run OK but the results are totally flawed.

5.1.1. Installation

Python (version 2.4 or newer) and a **C-compiler** for generating the binary executables of NAB-based applications must be available.

The **AMBER Tools package (version 1.3 or better 1.4)** must be installed and the environment variable pointing to its main directory (*\$AMBERHOME*) must be set correctly. The *\$AMBERHOME/bin* subdirectory must be in the executables path (*\$PATH*). If the *AMBER Tools* installation passes the tests that are delivered with that package, the utilities described in this document should also work.

The Python scripts do not require special packages or modules other than those included in (most?) standard Python distributions. They have only been tested on UNIX-like systems like Linux and Mac OSX, but not under MS-Windows.

The *AMBER Lite* distribution has the following file structure:

amberlite/ is the root folder;

- ../python** contains the Python scripts;
- ../src** contains the NAB source files (extension ".nab");
- ../bin** should eventually contain all binary NAB applications and also soft links to the Python scripts in the **../python** subfolder so that only this single folder has to be added to the global *\$PATH* variable;
- ../doc** is for documentation and contains this manual and the GPL license text;
- ../examples** and its subfolders contain the files used as examples in Appendix C of this manual.

The simplest first-time installation procedure is to expand the file *amberlite.tgz*, go to the generated **amberlite** directory, and execute the **install.py** script. The script will check that the *AMBERHOME* environment variable is set and that all required AMBER Tools executables are found in the path. It will then create a 'bin' subdirectory, compile the NAB sources, put the resulting binaries into the **bin** subdirectory and also make symbolic links to the python scripts in the same directory.

You must finally add the resulting **bin** directory to your *PATH* environment variable.

5.1.2. Python Scripts

The following Python scripts are currently included:

pytleap prepares AMBER parameter-topology (PRM) files, AMBER coordinates (CRD) files and corresponding PDB files for proteins, organic ligands (or peptides), and receptor/ligand complexes, using as input PDB files (for proteins and peptides) or SDF files for organic molecules. It is basically a wrapper around *tleap* and *antechamber*.

pymdpbsa is a full analysis tool for MD(GB,PB)/SA computations, given an MD trajectory (or a single PDB file) of a receptor-ligand complex and the individual PRM files for the complex, the receptor, and the ligand.

The Python scripts take **command line options** many of which assume default values. If the default values apply, these options can be omitted. Most options are of the form `--option value` where *value* can be a filename, an integer, a float, or a special string (to be included in quotes). Typing just the executable name or followed by `--help` lists the options and exits.

Common errors, like e.g. missing files, are captured by the scripts which also always check that the AMBERHOME environment variable is set and that all required binary executables are available and in the execution path.

The *pymdpbsa* script creates a **temporary subdirectory** of the current working directory. Computations are executed in this temporary folder and all output is stored there also. When finished, the resulting data are copied back to the starting directory. By default, the temporary directory is **not** removed. The user can explicitly request its automatic removal via the `--clean` option. Alternatively, it can be removed manually later. Temporary directories have names which make them easy to identify and all have the extension `.tmpdir` (see details later).

5.1.3. NAB Applications

The NAB applications are written in NAB language, which is "C" with numerous additional functions specific to computational chemistry problems. NAB works as a pre-compiler, generating C-code from the NAB source which is then processed through the default C-compiler. NAB functionality has much in common with the "big" AMBER modules, but there are also some notable differences:

The NAB applications cannot handle explicit solvent and periodic boundaries but work only with implicit solvation models. The possibilities to use restraints on atoms are also more limited and use a notation different from the AMBER 'mask' scheme (explained later). Otherwise, they deliver results which are fully compatible with original AMBER simulations under identical conditions.

The NAB applications presented here use the same parameter-topology files as AMBER modules like, e.g., *sander*, but they read coordinates (initial atom positions) from PDB files and not from AMBER-specific coordinate (CRD) files. The only output format for MD trajectories is the "binpos" format which can be read by various other packages or can also be converted to other formats via the *ptraj* utility included in AMBER Tools.

The following NAB-based tools are currently included:

ffgbsa returns the AMBER energy (MM + GB polar solvation + "non-polar" solvent-accessible surface term) of a system, given its PRM and PDB file.

minab is a crude conjugate-gradient minimizer using PRM and PDB files as input and generating a PDB file with the refined coordinates.

mdnab is a molecular dynamics routine with a minimum of user-specified options which takes PRM and PDB files as input and writes out the MD trajectory in the "binpos" format.

These NAB applications are single-line commands taking a number of arguments (which makes it easy to incorporate them into other scripts). In contrast to the Python scripts, they do not use the (more) convenient `--option` scheme, but **require the command line arguments in the correct order**. Entering just the name of the application without arguments lists a help which shows and explains the arguments to be used.

There is no extensive exception handling in the NAB applications. User errors are punished by simple crashes of the applications!

Users who want to modify NAB applications must edit the source, re-compile it into a NAB binary (using the command `nab source.nab -o binary_name`), and then copy the binary into a directory of their executable path.

5.2. Coordinates and Parameter-Topology Files

Simulations with AMBER modules require defined data and control files. The error-free generation of these files is often a discouraging hurdle for beginners or users who do not use AMBER regularly.

At least two data file types are required: a **coordinates (CRD)** file for AMBER modules (or **PDB** files for NAB applications) with atom positions and a **parameter-topology (PRM)** file containing all force field data required for the system. The two file types **must** have the **same number** of atoms and all atoms in the **same sequential order**. Not respecting this fundamental rule leads to severe flaws. The separation of coordinates and topology has the advantage that the same topology file can be used for various different starting coordinates. However, any change in the coordinate file that alters also the number of atoms or even their sequential order is not allowed. This is a frequent source of error and re-using PRM files created some time in the past under not well documented conditions is strongly discouraged.

The current tool delivered with AMBER to prepare coordinate (CRD or PDB) and parameter-topology (PRM) files is called **leap** (*tleap* for the terminal variant and *xleap* for the graphics variant).

Since *leap* is not particularly user-friendly, a Python script *pytleap* (see section 5.3) has been prepared which runs the terminal version of *leap* in the background and does not require a direct interaction with *leap* itself, at least for simple tasks like preparing a protein or a receptor-ligand complex for simulations with implicit solvent.

For small organic molecules, *pytleap* first invokes **antechamber** [79] before passing them through *leap*, allowing the usage of the *gaff* force field [68] for organics without directly interfering with *antechamber* itself.

The Appendix 5.8 (page 132) gives a short outline of the most important preparation steps required on the raw data (mostly PDB files) before using any AMBER-related tools. Those recipes may not be the most elegant ones but they work in most cases and help avoid common problems.

5.3. *pytleap*: Creating Coordinates and Parameter-Topology Files

pytleap calls the *tleap* and/or *antechamber* utilities in the background. It is invoked by a single command line with a set of options and eventually creates the files required for an AMBER simulation, starting from a PDB file (protein) and/or an SDF file (ligand). The script is especially useful to set up receptor-ligand complexes for simulations using MM(GB)(PB)/SA and related techniques, but can also be used for isolated proteins or ligands.

Proteins (or peptides) are read as PDB files in *pytleap*. Other formats are not supported. Be sure to have a "clean" PDB file as described in Appendix 5.8.

The SDF format for small organic ligands is chosen for reasons of compatibility. SDF files can be written by most standard molecular modeling packages and contain the information required by the *antechamber* package to generate the files for AMBER simulations. The format is simple and includes the connectivity with bond orders. Note that the SDF file of the ligand **must** have **all hydrogens** included. Also, the formal charge on the ligand (if any) is **not** read from the SDF file but **must** be explicitly specified (see later). For charge calculations, we use the *sqm* semi-empirical QM routine from AMBER Tools instead of MOPAC. After some tests, we have opted for less severe gradient requests than those used by default in *antechamber* to speed up the partial charges generation for ligands: `grms_tol` is set to 0.05. We include the peptide bond correction by setting `peptide_corr=1`.

To generate AMBER files for a **protein-ligand complex**, prepare the protein in PDB and the low-molecular-weight ligand in SDF, i.e., save both components in distinct files (and make sure that the protein PDB file does not contain the ligand anymore). In the case of protein-peptide (or protein-protein) complexes, you must also separate the two entities, in this case into distinct PDB files, since individual parameter-topology files have to be generated for the complex and for each component separately if MM(GB)(PB)SA computations are envisaged later.

Obviously, the geometry of the entire complex must be reflected in the coordinates of the respective files. *pytleap* will only combine the protein and the ligand into a single structure, assuming that the ligand fits the target in a desired way. It will of course not "dock"!

5.3.1. Running *pytleap*

Note: Since *pytleap* and the modules called by it read or write temporary files with defined names, it is wise to **run one single instance of *pytleap* in a directory**. Not respecting this rule will lead to confusion and errors!

Typing *pytleap* without any arguments (or followed by `--help`) results in the following output:

```
-----  
pytleap version 1.2 (December 2010)  
-----
```

```
Usage: pytleap [options]
```

```
Options:
```

5. amberlite: Some AMBER-Tools-Based Utilities

```
-h, --help          show this help message and exit
--prot=FILE         protein PDB file                      (no default)
--pep=FILE          peptide PDB file                      (no default)
--lig=FILE          ligand MDL (SDF) file                 (no default)
--cplx=FILE         name for complex files                (no default)
--ppi=FILE          name for protein-peptide complex files (no default)
--chrg=INTEGER      formal charge on ligand               (default: 0)
--rad=STRING        radius type for GB                   (default: mbondi)
--disul=FILE        file with S-S definitions in protein  (no default)
--sspep=FILE        file with S-S definitions in peptide  (no default)
--pffrc=STRING      protein (peptide) force field         (default: ff03.r1)
--lffrc=STRING      ligand force field                   (default: gaff)
--ctrl=FILE         leap command file name               (default: leap.cmd)
```

The command line options are presented here below:

--prot *filename* uses the PDB file *filename* as the protein structure. The PDB file must be "clean", according to the rules outlined in the Appendix 5.8.1. The *leap* module adds hydrogens with correct names (and also missing heavy atoms, if any), attributes the correct partial charges and AMBER atom types,¹ and eventually writes out the files for the protein as mentioned in section 5.3.2.

--pep *filename* reads a (clean) PDB file *filename* as the peptide structure. There is no difference to the **--prot** option except that a second (separate) peptide (or protein) can be read in and combined later with the structure read via **--prot** to a protein-peptide (or protein-protein) complex (see **--ppi** below).

--lig *filename* uses the SDF file *filename* as the ligand structure. The ligand file **must include all hydrogens**. The structure is processed through *antechamber* that generates various files required by *tleap* to build the PRM file for the ligand. Inside *antechamber*, the ligand becomes a molecule (residue) with the name "LIG". This name is then taken over by *leap* and appears as such in the resulting PRM and PDB files. The name "LIG" is the default name for a ligand in the *pymdpbsa* (section 5.7). See also option **--chrg** when using the **--lig** option. **Note:** We assume here that a ligand is a **single-residue** low-molecular-weight organic molecule.

--cplx *filename* (**no extension!**) will generate the AMBER files PRM, CRD and a PDB file of the complex of the protein and the ligand read in with the **--prot** and **--lig** options. When generating AMBER files for the complex, the files for the individual protein and ligand are always generated also. They are useful when running MM(GB)(PB)/SA computations later (section 5.7). **This option only makes sense when both the --prot and --lig options are also chosen.**

--ppi *filename* works the same as **--cplx**, except that it generates a complex between two units supposed to be clean proteins (peptides), not requiring any intervention of *antechamber*. Furthermore, **--cplx** and **--ppi** cannot be used in the same run, i.e., we

¹Charges and atom types will correspond the chosen force field parameter set and the libraries going with them.

can only deal with either a protein/organic-ligand complex or a protein/protein (or protein/peptide) complex.

- chrg** *integer* must be used if an organic ligand read from an SDF file is formally charged (even if the charge is also given in the SDF file). For neutral ligands, this option can be omitted. For charged ligands however, it is required! Enter it as an integer reflecting the correct total charge of the ligand. The computation of partial charges via the AM1-BCC method[71, 72] will fail if the formal charge on the ligand does not make sense with the chemical structure including all hydrogens and *pytleap* will quit.
- rad** *radius_type* is used to choose the atomic radii for Generalized-Born. The default radius type is the "modified Bondi" option to be used with the GB option *gb* set to 1. For *gb* = 2 or 5, the original AMBER documentation suggests the radius type *mbondi2*.
- disul** and **--sspep** *filename* are used to generate disulfide bonds. Disulfide bridges must be prepared in the original PDB file by renaming the involved cysteine residues from CYS to CYX (see 5.8.2.2). The *filename* in this option must relate to a file that contains pairwise integer numbers of cysteine residue names to be connected (one pair per line!). These numbers must correspond to the ones in the original PDB file!² See the example in section 5.10.1. We consider that this explicit formation of disulfide bonds is to be preferred over "automatic" S-S bond formation, be it by using *Sy* distances or by relying on *CONNECT* records in PDB files. **NOTE: --disul applies to the file read in via --prot while --sspep is applied to the molecule read in via --pep.** If both proteins (peptides) have disulfide bonds, you must use separate definition files for the respective S-S links!
- sspep:** see above.
- pfrc** *filename* specifies the force field parameter set for the protein. Since AMBER can use different force fields, this option allows to choose among them. The selection actually does not call the parameter file itself but a *leap* command file that initializes it. These special *leap* files all have a name *leaprc.xxxx* and are retrieved when the `AMBERHOME` environment variable is set correctly. **You must only specify the xxxx part of the name!** Thus, *ff99* selects the *parm99* parameter set, while the **default** *ff03.r1* selects the latest *parm03* force field with the correct charges for N- and C-terminal residues also. Make sure to have this file (with the full name *leaprc.ff03.r1* included in the directory where all default *leap* command files are kept.³
- lfrc** *filename* selects the force field for the ligand. At this point, the default *gaff* force field is the only reasonable choice in most cases and you can omit this (default) option.
- ctrl** *filename* can be used to change the default name of the leap command file generated by *pytleap* (default *leap.cmd*). In general, this is not necessary, except if you would like to keep this particular file and protect it from being overwritten by the default name the next time you use *pytleap* in the same directory.

²In 'weird' PDB files where insertions and deletions get special names, trying to keep a 'standard' numbering of residues for the main protein of a family, much can go wrong. In these cases, it is best to renumber the residues sequentially in the PDB file before referring to residue numbers.

³The full path to this place is `$AMBERHOME/dat/leap/cmd`.

NOTE: This version of *pytleap* does not offer the possibility to add solvent and counter-ions. It would be straightforward to add these options to the script if you are familiar with *leap*. Alternatively, you could use the *leap.cmd* (or alike) created by *pytleap*, edit it with a standard editor to add specific *leap* commands, and then resource it through *tleap* (e.g., with `tleap -f leap.cmd`).

5.3.2. Output from *pytleap*

Output from *pytleap* varies with the chosen command line options (see 5.3.1). Coordinate (CRD) files, parameter-topology (PRM) files and a corresponding PDB file are always generated. Hydrogen names in the output PDB files are "wrapped", making these files readable also by elder software packages which require this format. Note that the actual atom names in the PRM file are unwrapped. This has no consequence on computations. However, special residue names like HIE, HID, HIP, CYX, etc., are kept and may lead to flawed representations of the PDB files in software packages which do not recognize these residue names. The *ambpdb* routine included with AMBER Tools can be used to regenerate "standard" residue names if you need them.

Files generated by *pytleap* have a `'.leap.'` string in their name to identify them as "created by *leap*". **You should always use the corresponding *.leap.* files (or copies of them) for simulations!** This guarantees that the CRD, PDB, and PRM files are compatible, having the same number and sequence of atoms.

In addition, a file *leap.cmd* is left over. This is the file that was generated by *pytleap* and run through *leap*. The file *leap.out* is the output from *leap*, with the messages that would have been generated by running *leap* interactively. Finally, the *leap.log* file is the standard log from *leap*.

A special SYBYL MOL2 file is created when running *pytleap* on a ligand (i.e., a low-molecular-weight organic compound which is processed through *antechamber*). This file has the format of a generic MOL2 file, apart from the fact that atom types are not SYBYL but *gaff* atom types. The name of this file is `filename.ac.mol2`, with `.ac.` marking it as a file generated by *antechamber*.⁴ The partial charges are those from the AM1-BCC method.[71, 72]

Some additional files may be left over when *antechamber* is invoked. One **important file** is the `*.leap.frcmod` file containing additional parameters which are not in the original *gaff* parameter file. They are generated based on equivalences, "guessing", or empirical rules described the *gaff* paper.[68] The *frcmod* file can also be used as a quality check for the ligand parameters. Large *frcmod* files with many "guessed" parameters (especially for torsion angles) should be considered carefully.

Finally, the input and output files of the semi-empirical tool *sqm* are left. The output file (*sqm.out*) might be useful for debugging if the partial charges seem totally inadequate despite the correct usage of the `--chrg` option (if required).

⁴Opening this MOL2 file in a standard software that can read MOL2 files may lead to strange results because the *gaff* atom types do not reflect chemical elements as standard SYBYL MOL2 files with TRIPOS force field atom types.

5.3.3. Error Checking

If you have experience with the *leap* application, look at the *leap.cmd* file that was created via *pytleap*. All the options that you have chosen should be represented as correct *leap* command lines. Furthermore, the *leap.output* and *leap.log* files should not show any errors, at best some warnings. If in doubt that the parameter-topology files have been correctly generated, look at these warnings and decide if they are benign. Eventually, the NAB application *ffgbsa* described below (section 5.4) can be used to run a single AMBER energy evaluation on the system. If the results returned by *ffgbsa* look very strange for a supposedly reasonable structure, you probably have a serious issue with your set of CRD, PDB, and PRM files.

5.4. Energy Checking Tool: *ffgbsa*

The NAB routine *ffgbsa* is an energy function called by the *pymdpbsa* application presented later (section 5.7). It can also be used as a standalone routine to check the AMBER energy of a molecular system and to test the correct working of a PDB/PRM file combination. It is invoked as:

```
ffgbsa pdb prm gbflag saflag
```

The order in the command line input is compulsory! *pdb* is the PDB file of the system and *prm* the related PRM file. *gbflag* is a flag to switch on one of the Generalized-Born (GB) options in AMBER and can be 1, 2, or 5.⁵ Other values switch off GB and a simple distance-dielectric function $\epsilon = r_{ij}$ is used.

When *saflag* = 1, the solvent-accessible surface area (SASA) is also computed (via the *molsurf* routine included with NAB) and returned in Å², together with a SASA energy term which is simply *SASA* * 0.0072 in this case. The default cutoff for non-bonded interactions is 100 Å, i.e., virtually no cutoff for most systems. An example for the usage of *ffgbsa* is given in section 5.10.2.

Remarks regarding the usage of *molsurf*:

The correct way to evaluate the SASA is to augment the radii of all atoms by the probe radius (usually 1.4 Å) and then run *molsurf* with a probe of radius of zero. This is also the implementation in *ffgbsa* here. The atom radii values are given in the following table:

Note: In some rare cases *molsurf* fails to give back a valid surface area. Scripts calling *ffgbsa* must be prepared to capture this. The *pymdpbsa* procedure described later catches such instances and excludes value sets in which the error occurs from the statistical analysis (cf. end of section 5.7.5.1).

5.5. Energy Minimizer: *minab*

The main purpose of this (very) simple minimizer is to refine a system prior to MD runs, mainly to remove potential hotspots which might destabilize the MD initiation. Using it for other purposes is at the discretion of the user.

⁵These values correspond to the "igb=" options in AMBER commands and stand for different implementations of the GB scheme.

Table 5.1.: *Atom Radii Used in molsurf*

atom	radius (Å)	atom	radius (Å)
C	1.70	H	1.20
N	1.55	O	1.50
S	1.80	P	1.80
F	1.47	Cl	1.75
Br	1.85	I	1.98
any other	1.50		

The NAB routine *minab* uses the conjugate gradient minimizer of NAB to refine the energy of a system. To circumvent cutoff problems,⁶ the cutoff for non-bonded interactions (vdW and Coulomb) is set to 100 Å and that for GB is fixed to 15 Å. The non-bonded list is not updated at all. The default for the gradient rms is set to 0.1.

For large systems, this is far from efficient. However, as stated above, the main purpose of this routine is to get rid of hotspots prior to running MD and in general, a few hundred iterations are sufficient to guarantee a decent structure for MD, especially when the MD starts with a heat-up phase as used in the *mdnab* application described in section 5.6.

The *minab* routine is invoked by:

```
minab pdb prm pdbout gbflag niter ['restraints' resforce]
```

Just typing *minab* without arguments gives a help screen. The explanation for the arguments follows:

- *pdb* and *prm* are the PDB and **corresponding** PRM file of the system;
- *pdbout* becomes the PDB file of the refined system;
- *gbflag* is the GB flag which can be 1, 2, or 5 while any other value switches to distance-dependent dielectrics (as in section 5.4);
- *niter* is the maximum number of iterations;

and for the optional arguments:

- *restraints* specifies residues or atoms to be tethered in their motion (NAB atom expression **between quotes**);
- *resforce* is a float specifying the restraint potential in $\text{kcal}\cdot\text{mol}^{-1}\cdot\text{\AA}^{-2}$.

⁶The current cutoff scheme for non-bonded interactions in AMBER modules and NAB does not use a switching function to smooth the cutoff. This can lead to problems every time the non-bonded list is updated. Thus a fairly short cutoff distance with frequent list updates usually ends in line search problems before the required number of iterations or the requested rms of the components of the gradient is reached.

The `restraints` entry must be an atom expression according to the NAB rules outlined in 5.9.2. If for example all $C\alpha$ atoms should be restrained, this entry would be `::CA`. **If the restraint mask is given, the restraint potential `resforce` must also be specified.**

Since *minab* is a simple command-line tool, it can be called by other routines or scripts where a rough energy refinement is desired. The output (by default to the screen) can be captured for later analysis into a file via a simple redirect ("`>`").

5.6. Molecular Dynamics "Lite": *mdnab*

The NAB application *mdnab* has been written for simple molecular dynamics with a minimum number of settings required by the user. Its main purpose is to run moderately short trajectories to be used e.g. for MM(GB)(PB)/SA applications.

Most settings are hardcoded and can only be changed by editing and re-compiling the source *mdnab.nab*.

The following (non-mutable) defaults are used:

The cutoff for non-bonded interactions and GB is always 12 Å. An update of the nonbonded list occurs every 25 steps. The integration step is 2 femtoseconds (using "rattle" to allow this fairly large step). The temperature is controlled via Langevin dynamics with a friction factor ("`gamma_ln`") of 2 for the production phase. The production temperature is fixed at 300 K. And *mdnab* **always saves one frame per picosecond**, independent of the length of the trajectory.

A heating and equilibration phase is automatically invoked prior to the actual production trajectory recording: 100 steps from 50 to 100 K, 300 steps from 100 to 150 K, 600 steps from 150 to 200 K, 1000 steps from 200 to 250 K, 3000 steps from 250 to 300 K, and an additional 10000 steps at 300 K.⁷

mdnab is started by

```
mdnab pdb prm traj gbflag picosecs ['restraints' resforce]
```

The command *mdnab* without arguments lists the possible arguments, the **sequence** of which is **compulsory**. The command line arguments are similar to those in *minab*:

- `pdb` and `prm` are the PDB and corresponding PRM file of the system;
- `traj` is the name for the production phase trajectory which will be saved in the binary "binpos" format (the extension `.binpos` is automatically attached);⁸
- `gbflag` is the GB flag which can be 1, 2, or 5 (as in section 5.4), or anything else to switch off GB and use a distance-dependent dielectric function $\epsilon = r_{ij}$;
- `picosecs` is the total number of picoseconds to run the production phase;

with the optional arguments:

⁷Since these last 10000 steps at 300 K are run under identical conditions as subsequent the production phase, the user can simply extend the "equilibration" by discarding all frames from the production phase up to the point where the trajectory can be considered "stable" (noting that "stable" or "steady-state" are not well-defined terms anyway).

⁸This format can be read by various software packages like *VMD*, but can also be translated into other formats using the AMBER utility *ptraj*.

5. amberlite: Some AMBER-Tools-Based Utilities

- `restraints` specifies atoms to be tethered in their motion (given as a NAB atom expression **between quotes**, see section 5.9.2);
- `resforce` is the restraint potential in $\text{kcal}\cdot\text{mol}^{-1}\cdot\text{\AA}^{-2}$ **which has to be given** if a restraint expression is specified.

While the trajectory is saved to the specified file name (the `traj` command line argument), the full output goes to the screen. To capture the output for later inspection, use the UNIX "redirect" (`>`) to a file and end the command line with a `&` (making `mdnab` a background job).

Note that only the production phase of the trajectory is recorded into the `traj` file. The heat-up phases are only documented in the general output (to the screen or to a text file, if redirected).

5.7. MM(GB)(PB)/SA Analysis Tool: *pymdpbsa*

5.7.1. Brief Overview on MM(GB)(PB)/SA Concepts

The original MM(GB)(PB)/SA procedure was developed in the late 1990's and the user should refer to some original papers on this subject.[80–83] The goal was to develop a relatively fast molecular-mechanics (-dynamics) based method to evaluate free energies of interactions. MM stands for Molecular Mechanics, PB for Poisson-Boltzmann, and SA for Surface Area. You may also wish to refer to reviews summarizing many of the applications of this model,[82, 84] as well as to papers describing some of its applications.[85–89]

The free energy for each species (ligand, receptor, or complex) is decomposed into a gas-phase energy ("enthalpy"), a solvation free energy and an entropy term, as shown in equation 5.1.

$$G = E_{\text{gas}} + G_{\text{solv}} - T \cdot S \quad (5.1)$$

$$= E_{\text{bat}} + E_{\text{vdW}} + E_{\text{coul}} + G_{\text{solv,polar}} + G_{\text{solv,nonpolar}} - T \cdot S \quad (5.2)$$

where E_{bat} is the sum of bond, angle, and torsion terms in the force field, E_{vdW} and E_{coul} are the van der Waals and Coulomb energy terms, $G_{\text{solv,polar}}$ is the polar contribution to the solvation free energy and $G_{\text{solv,nonpolar}}$ is the nonpolar solvation free energy.

The sum $E_{\text{bat}} + E_{\text{vdW}} + E_{\text{coul}}$ is the complete gas phase force field energy, the molecular mechanics ("MM") part.

The polar solvation free energy $G_{\text{solv,polar}}$ can be evaluated via implicit solvation models like Poisson-Boltzmann (PB) or Generalized-Born (GB). The nonpolar contribution $G_{\text{solv,nonpolar}}$ is usually computed by a simple linear relation for a "cavity" term

$$G_{\text{solv,nonpolar}} = \gamma \cdot \text{SASA} + \text{const.} \quad (5.3)$$

where SASA is the solvent-accessible surface and γ has the dimension of surface-tension. Similarly, one could also use the volume enclosed by the SASA (SAV)

$$G_{\text{solv,nonpolar}} = p \cdot \text{SAV} + \text{const.} \quad (5.4)$$

with p having the dimensions of pressure.

In a more sophisticated approach, $G_{solv,nonpolar}$ can be split into a repulsive ("cavity") and an attractive ("dispersion") term, as described in detail in the 2007 paper of Ray Luo and coworkers.^[90]

The vibrational entropy can be evaluated, for example, via normal mode analysis. It has become common practice in recent work to exclude the entropy terms from MM(GB)(PB)/SA computations. This is acceptable when only relative free energies are computed to compare similar ligands in similar receptors. Furthermore, the entropy computation is the fuzziest part of the procedure and contributes to the largest fluctuations in the overall free energy when evaluating it over a number of MD frames.

The free energy of interaction in the complex can then be evaluated as:

$$\Delta G_{int} = G_{complex} - G_{receptor} - G_{ligand} \quad (5.5)$$

In the early work, separate dynamics trajectories were recorded for all three species in explicit solvent. The solvent was then discarded, the free energy was evaluated according to the procedure above for a number of frames for each species. Eventually, ΔG was calculated by

$$\Delta G_{int} = \langle G_{complex} \rangle_{traj} - \langle G_{receptor} \rangle_{traj} - \langle G_{ligand} \rangle_{traj} \quad (5.6)$$

where $\langle G_i \rangle_{traj}$ is the average value for species i over all selected frames recorded during the production phase of the MD trajectory.

In the meantime, the method has been implemented and used in many variants, all of which have their advantages and disadvantages. The method presented hereafter is among the simplest and cheapest in terms of CPU power. It is based on a single trajectory of the complex alone. Each recorded frame is then split into receptor and ligand and equation 5.5 is applied to compute the interaction energy of the frame. The final interaction energy is then the average over the ΔG values of the selected frames. Also, the entropy is not evaluated at all.

5.7.2. Pitfalls and Error Sources

While the basic concepts are simple, there are many pitfalls. The initial idea was to compute values for the free energy of binding close to experimentally observed ones, without further tuning of parameters. However, since the computations of energy terms are based on force field parameters (internal energy, van der Waals interactions, and vibrational entropy via normal-mode analysis) and on concepts like atomic radii and partial charges (electrostatics and polar solvation terms), discussions on the quality of parameters are inevitable.

An issue not discussed in enough detail in many papers reporting MM(GB)(PB)/SA (and variants) is the quality of the MD trajectory. Unstable trajectories with unreasonably strong fluctuations or important transitions (conformational changes, ligand pose variations, etc.) will always yield questionable results. If such transitions happen, they must be checked carefully before the results are used for MM(GB)(PB)/SA.

In the "one-trajectory" approach implemented here, there is an additional pitfall. Since both the receptor and the ligand are only considered in the bound state, strain energy from distortions in the complex is not evaluated. This may not be an issue for the receptor if there are no strong induced-fit effects. For the ligand however, this can amount to a perceivable difference if the

bound state adopts a conformation which is definitely higher than for the unbound ligand in solution. Such "errors" may partially cancel when series of similar ligands are compared in the same receptor. But it obviously adds to the fuzziness of the results. When in doubt, a trajectory of the ligand alone (under identical conditions as for the complex) should be recorded to assess the average energy of the ligand in the unbound state.

5.7.3. Some Technical Remarks on *pymdpbsa*

pymdpbsa uses *ffgbsa* (see section 5.4) or the stand-alone Poisson-Boltzmann solver *pbsa* to evaluate energies. The tool *ptraj* is called to decompose the MD trajectory into individual frames for the complex, the ligand, and the receptor.

Because various temporary files are generated during execution, *pymdpbsa* automatically creates a subdirectory in which all calculations are run. This subdirectory (extension `.tmpdir`) contains all temporary files and also the final results, copies of which are transferred to the starting working directory upon completion. By default, the temporary directory is **not** removed automatically.

The following files are necessary to run *pymdpbsa* on a receptor-ligand complex:

- a molecular dynamics trajectory file of the complex (any format that can be read by *ptraj*, including Z-compressed ones and binary binpos files like those created by *mdnab*, see section 5.6);
- three AMBER parameter-topology PRM files, one for the complex, one for the ligand alone, and one for the receptor alone (as created by *pytleap*, see section 5.3);

5.7.4. Running *pymdpbsa*

Invoking *pymdpbsa* without any arguments (or with `--help`) will list all possible options.

```
-----
pymdpbsa version 0.6 (December 2010)
-----

Usage: pymdpbsa [options]

Options:
  -h, --help            show this help message and exit
  --proj=NAME           global project name
  --traj=FILE           MD trajectory file                (default: traj.binpos)
  --cprm=FILE           complex prmtop file              (default: com.prm)
  --lprm=FILE           ligand only prmtop file          (default: lig.prm)
  --rprm=FILE           receptor only prmtop file        (default: rec.prm)
  --lig=STRING          residue name of ligand          (default: LIG)
  --start=INT           first MD frame to be used       (default: 1)
  --stop=INT            last MD frame to be used        (default: 1)
  --step=INT            use every [step] MD frame       (default: 1)
  --solv=INT            0 for no solvation term (eps=r)
                      1, 2, or 5 for GBSA
                      3 for PBSA
```


	4 for PBSA/dispersion	(default: 1)
--clean	clean up temporary files	(default: no clean)

You only need to specify options that are different from the default. Thus, you can avoid entering a lot of options by simply selecting file names like *com.prm*, *rec.prm*, and *lig.prm* for the PRM files, calling the trajectory file *traj.binpos*, and by giving the ligand the residue name *LIG* in your original structure file (the default if *pytleap* in section 5.3 was used).

--proj has to be followed by a the global name of the project and all output files will incorporate this string. The name of the temporary directory created will also start with the project name (followed by sequence of random characters and the extension '.tmpdir'). When this options is omitted, the project name becomes *None* (not really useful for later identification).

--traj is followed by the filename of the trajectory. As already mentioned, the trajectory file can be any format which can be processed by the AMBER tool *ptraj*. If the trajectory file name is *traj.binpos*, this option can be omitted.

--cprm, --lprm, --rprm are used to feed in the names for the PRM files of the complex, the ligand, and the empty receptor. None of these PRM files is generated by *pymdpbsa*. They must be specified by the user. If the *pytleap* utility (see section 5.3) has been used on a complex, these three files should have been created. If you want to use default names, rename these files to *com.prm*, *rec.prm*, and *lig.prm*.

--lig is used to specify the name of the ligand. This is the (up to 4 characters long) "residue" name the ligand would have in a PDB file. If the complex has been prepared via *pytleap*, the ligand name will probably be *LIG* (i.e., the default). Note that the ligand is supposed to be one single residue in that case. Alternatively, the ligand can also be specified by its residue number. Thus if the ligand is residue 281 in the PDB file of the complex, you may specify --lig 281. This also allows to have multi-residue ligands like in protein-peptide (protein-protein) complexes. If e.g. the ligand covers residues 134 to 156 in the **overall** PDB file of the complex, you can specify --lig '134-156'.⁹

--start, --stop, and --step set the first and last frame of the MD trajectory to be used for evaluating the energy, and the step size (e.g., --step 5 means every fifth frame). **By default, these values are all 1**, i.e., only the first frame is used. Thus, the free energy of interaction for a single PDB file can be computed by specifying as 'trajectory' (with --traj) the name of the PDB file and neglecting the start/stop/step options.

--solv followed by an integer chooses the solvation option. The **default** is '--solv 1'. For values other than 1 to 5, the returned electrostatic energy term is evaluated with a distant-dependent dielectric function $\epsilon = r_{ij}$ with no additional polar solvation correction. For values 1, 2, or 5, the corresponding GB variant (*igb* in AMBER) is used with a nonpolar contribution of $0.0072 * \text{SASA}$ (where the solvent-accessible surface SASA is computed via *molsurf*); for *solv* = 3, GB is replaced by PB and the non-polar solvation energy term is $0.005 * \text{SASA} + 0.86$; for *solv* = 4, the polar solvation free energy part is

⁹Using quotes to include more complex atom masks is a safe way to circumvent problems with the shell interpretation.

5. amberlite: Some AMBER-Tools-Based Utilities

computed with PB, the nonpolar portion is evaluated by a "cavity" term and a "dispersion term";^[90] the detailed settings for this approach are identical to those suggested in the original *pbsa* documentation; **note that the '`--solv 4`' option is experimental at this stage and not widely tested, ...use with care.**

`--clean` removes the temporary directory, including all PDB or CRD files for the various MD frames. By default, these files are kept. You might choose to keep the files for debugging purposes in initial runs or for some graphics of overlays (since proteins are automatically RMS-fitted to the C α during the *ptraj* extraction). In any case, the relevant data are saved to the working directory, even when the `--clean` option is used.

5.7.5. Details on Internal Workings and Output of *pymdpbsa*

The internal workings and the output of *pymdpbsa* vary depending on the `--solv` options. In all cases, the *ptraj* tool is called to split the trajectory into individual frames. Since each interaction energy evaluation requires three files (complex, receptor, ligand), the splitting of a trajectory with N frames results in 3·N files.¹⁰

5.7.5.1. Distance-Dependent Dielectrics or Generalized Born

For `--solv = 0, 1, 2, or 5`, the *ffgbsa* routine is called to evaluate energy terms. Since *ffgbsa* required PDB files as coordinate input, the trajectory is split into individual PDB files. These files are named according to the project, the part of the structure (C for whole complex, R for receptor alone, L for ligand alone), and the frame number.

Thus a file `TEST.R.pdb.45` would be the PDB file of the empty receptor corresponding to frame 45 of the trajectory of the project named TEST.

Each run creates **four tables** with energy values returned by *ffgbsa*: one for the ligand, one for the receptor, one for the complex, and one for the interaction energies. The tables inherit the name of the project, followed by L, or R, or C, or D, (ligand, receptor, complex, and energy difference) and the extension ".nrg". These tables are simple text files and can be used as input for plotting routines, e.g., to check possible drifts or strong fluctuations. An excerpt of a *.D.nrg output is shown next:

10	-56.84	0.00	-55.30	-61.67	67.98	-7.85
20	-58.67	-0.00	-52.84	-68.51	70.51	-7.83
...						
...						
90	-57.21	0.00	-56.83	-52.23	59.57	-7.72
100	-59.20	0.00	-57.10	-41.51	47.27	-7.86

The first column is the frame number, followed by the total energy, the internal force field term (stretch, bend, and torsion terms), the van der Waals term, the Coulomb term, the Generalized-Born term, and the solvent-accessible surface term. **Note** that the internal force field term **must**

¹⁰The splitting into ligand and receptor is performed by separate *ptraj* calls. Depending on the part to be written out, the *ptraj* command "strip" followed by an AMBER mask is used to remove the rest of the structure. Thus for example, if the ligand is a residue called LIG, the ligand alone is obtained with the strip mask "' :*&!:LIG'" meaning "strip off all residues but not the residue named LIG".

be zero (within the limits of precision) in the *.D.nrg tables because we use a single trajectory and do not account for distortions in the receptor or the ligand. The corresponding columns in the respective C, R, and L tables will not be zero. In the special case `--solv 0`, the GB column has also zero values only.

The final evaluation summary is stored in a file with the project name and the extension ".sum". The summary shows averages and corresponding standard deviations and mean errors for all energy terms. All values are given in kcal·mol⁻¹. The header lines show additional information useful for later documentation. An example is shown below:

```
=====
Summary Statistics for Project SOLV5
Frames                : 10 to 100 (every 10)
Solvation              : GB (--solv=5)
Trajectory File        : traj.binpos
Complex parmtop File   : com.prm
Receptor parmtop File  : rec.prm
Ligand parmtop File    : lig.prm
=====

----Ligand Energies-----
Etot =   -169.82 (  3.62,   1.14) Ebat =    64.78 (  4.69,   1.48)
Evdw =    20.51 (  2.25,   0.71) Ecoul =   -192.35 (  1.61,   0.51)
EGB  =   -68.33 (  1.51,   0.48) Esasa =    5.56 (  0.06,   0.02)

----Receptor Energies-----
Etot =  -4045.29 ( 31.63,  10.00) Ebat =   4157.74 ( 37.50,  11.86)
Evdw =   -756.47 ( 15.16,   4.79) Ecoul =  -4863.38 ( 94.96,  30.03)
EGB  =  -2681.64 ( 91.98,  29.09) Esasa =    98.45 (  0.54,   0.17)

----Complex Energies-----
Etot =  -4276.73 ( 34.61,  10.94) Ebat =   4222.53 ( 39.39,  12.46)
Evdw =   -791.58 ( 14.82,   4.69) Ecoul =  -5110.62 (102.32,  32.36)
EGB  =  -2693.26 ( 97.49,  30.83) Esasa =    96.20 (  0.56,   0.18)

----Interaction Energy Components-----
Etot =   -61.62 (  2.90,   0.92) Ebat =    -0.00 (  0.01,   0.00)
Evdw =   -55.62 (  1.43,   0.45) Ecoul =   -54.89 ( 12.81,   4.05)
EGB  =    56.70 ( 11.87,   3.75) Esasa =   -7.81 (  0.09,   0.03)
=====
```

For `--solv = 0, 1, 2, or 5`, the solvent-accessible surface is computed via the NAB subroutine *molsurf* in *ffgbsa*. The surface returned by *ffgbsa* is multiplied by a surface tension of **0.0072** to yield the "nonpolar" free energy component in kcal/mol. For details about the calls to *molsurf*, see section 5.4.

As mentioned before, the *molsurf* routine is generally robust, but has shown problems in some rare cases. Since the *pymdpbsa* script requires the output from *molsurf* (called via *ffgbsa*), we have built in a catch for these rare cases. If *molsurf* should fail, the returned surface value is set to zero for that frame and *pymdpbsa* emits a warning. In later statistical evaluations, frames with this problem are excluded from the evaluation, i.e., average values and standard deviations relate to "healthy" frames only.

5.7.5.2. Poisson-Boltzmann

For `--solv = 3` or `4`, the *pbsa* routine is called. This is done by generating a temporary input (control) file for *pbsa* called `pbsasfe.in`. The output of *pbsa* goes to `pbsasfe.out`. Both files are left over after the run and can be used to verify that everything went correctly.

Since *pbsa* requires CRD files, the trajectory is split into AMBER restart files rather than PDB files. The name giving is the same as for the PDB files (see 5.7.5.1) except that the "pdb" part in filenames is changed to "crd".

The script eventually calls *pbsa* by:

```
pbsa -O -i pbsasfe.in -o pbsasfe.out -p prmfile -c crdfile
```

The generated output tables are named as for the non-PB settings in section 5.7.5.1. However, the content of the tables varies slightly:

10	5.85	-55.31	-61.69	92.17	-40.36	71.02
20	2.61	-52.83	-68.54	92.79	-38.97	70.16
...						
...						
90	-1.51	-56.83	-52.21	78.21	-40.25	69.57
100	0.35	-57.10	-41.55	67.08	-39.25	71.16

The first column is the frame number, followed by the total energy, the van der Waals term, the Coulomb term, the Poisson-Boltzmann term, the solvent-accessible surface ("cavity") term, and the "dispersion term" (which is zero if the option `--solv=3` was used).

The final evaluation summary is stored in a file with the project name and the extension ".sum". This file is similar to that shown in section 5.7.5.1 except that some specific terms vary. An example is shown here:

```
=====
Summary MDPBSA Statistics for Project SOLV4
Solvation           : PB+SAV+DISP (--solv=4)
Frames              : 10 to 100 (every 10)
Trajectory File     : traj.binpos
Complex parmtop File : com.prm
Receptor parmtop File : rec.prm
Ligand parmtop File  : lig.prm
=====
-----Ligand Energies-----
Etot =      32.16 (  2.27,   0.72) Evdw =     -4.64 (  1.92,   0.61)
Ecoul =    106.04 (  1.70,   0.54) Epb  =    -72.46 (  1.37,   0.43)
Ecav =     53.22 (  0.36,   0.11) Edisp =    -49.99 (  0.43,   0.14)
-----Receptor Energies-----
Etot = -17754.52 ( 38.42, 12.15) Evdw = -1662.84 (  8.07,   2.55)
Ecoul = -14139.19 (122.39, 38.70) Epb  = -2662.75 ( 90.70,  28.68)
Ecav =   1848.33 (  6.20,   1.96) Edisp = -1138.08 (  5.39,   1.70)
-----Complex Energies-----
Etot = -17719.32 ( 40.29, 12.74) Evdw = -1723.10 (  8.18,   2.59)
Ecoul = -14088.04 (126.86, 40.12) Epb  = -2652.15 ( 94.66,  29.93)
Ecav =   1862.08 (  6.73,   2.13) Edisp = -1118.11 (  5.69,   1.80)
-----Interaction Energy Components-----
```

```

Etot  =      3.04 (  4.12,   1.30) Evdw  =     -55.62 (  1.43,   0.45)
Ecout =    -54.90 ( 12.81,   4.05) Epb   =      83.06 ( 12.49,   3.95)
Ecav  =    -39.47 (  0.88,   0.28) Edisp =      69.96 (  0.84,   0.27)
=====

```

5.7.6. Using *pymdpbsa* for Single-Point Interaction Energy

Since *ptraj* can read a single coordinate set (frame) as a "trajectory", *pymdpbsa* can also be used to generate the free energy of interaction for an isolated PDB or CRD file of a receptor-ligand complex. Just specify for the "trajectory" (`--traj`) the name of the single PDB or CRD file and leave the `--start`, `--step` and `--stop` options to their default of 1. Any of the `--solv` options can be used.

Obviously, the PRM files for the complex, the receptor, and the ligand, must be available also and must be specified if their names are different from the default.

In the single-point case, the output looks the same as for the multiple-frame evaluations. The tables have only one line (record) and the statistical data like standard deviation or standard error in the `.sum` file are all zero of course.

5.7.7. Remark Concerning Poisson-Boltzmann Options `--solv 3` and

`--solv 4`

The PB option has more adjustable parameters than the GB variants. The `--solv 3` option uses the following settings for calling *pbsa*:

```

&cntrl
ntx=1, imin=1, igb=10, inp=1,
/
&pb
epsout=80.0, epsin=1.0, space=0.5, bcopt=6, dprob=1.4,
cutnb=0, eneopt=2,
accept=0.001, sprob=1.6, radiopt=0, fillratio=4,
maxitn=1000, arcres=0.0625,
cavity_surften=0.005, cavity_offset=0.86
/

```

This command sequence is generated in *pymdpbsa* in the function `pbsacontrol_solv3`. Users who want to try different settings can change this code section of *pymdpbsa* to their gusto. Read the original *pbsa* documentation before doing so, however.

The `--solv 4` option to compute interaction energies is highly experimental. Users should read the original paper of Ray Luo et coworkers[90] before using this option. The command sequence generated by *pymdpbsa* is found in the source code in the function `pbsacontrol_solv4`:

```

&cntrl
ntx=1, imin=1, igb=10, inp=2
/
&pb
npbverb=0, istrng=0.0, epsout=80.0, epsin=1.0,

```

```
radiopt=1, dprob=1.6,  
space=0.5, nbuffer=0, fillratio=4.0,  
accept=0.001, arcres=0.25,  
cutnb=0, eneopt=2,  
decompopt=2, use_rmin=1, sprob=0.557, vprob=1.300,  
rhow_effect=1.129, use_sav=1,  
cavity_surften=0.0378, cavity_offset=-0.5692  
/  

```

5.8. Appendix A: Preparing PDB Files

The only required or useful data in a PDB file to set up AMBER simulations are: atom names, residue names, and maybe chain identifiers (if more than one chain is present), and the coordinates of heavy atoms. Non-protein structures (especially low-molecular-weight ligands) will cause problems, with the exception of water and some ions which are automatically recognized if their names in the PDB file correspond to the internal names in the AMBER libraries.

NOTE: Recent changes in *leap* are supposed to handle some of the hurdles (like generation of disulfide bonds) described below "automatically". I have not tested these options intensively. I suppose that they can be relied on in most cases but I still recommend to follow the recipes given below to be on the safe side.

5.8.1. Cleaning up Protein PDB Files for AMBER

This is a crucial step in the preparation and many potential problems and subsequent errors depend on this step!

Analyze the PDB file visually in any viewer that can represent (and maybe modify) the file. Alternatively, use a text editor. Delete all parts which are judged irrelevant for the simulation. Be aware that anything not protein or water can be expected to cause trouble later.

If the x-ray unit cell in the PDB file contains more than one image, choose the entity you want to use and delete the other(s).

If there is a **ligand**, save it as an MDL standard data file (SDF). Many software packages are able to do this directly. You may also save the ligand in PDB format and then use some other tools later to convert it into a decent SDF file (**including correct bond order and all hydrogens**). It is crucial to **keep the coordinates of its heavy atoms at their original location**. Then delete it from the PDB file. The ligand must be treated separately later.

Delete all water molecules that are not considered relevant. Some waters might be essential for ligand binding. If those waters are kept, they should be made part of the receptor (as distinct "residues"), not of the ligand. *leap* recognizes water if the residue name is WAT or HOH. In later simulations, they may have to be tethered (more or less strongly) to their original positions to prevent them from "evaporating".

Apply the same delete procedure to ions, co-factors, and other stuff that has no special relevance for the planned simulation.

Get rid off all protein (or peptide) hydrogens that are explicitly expressed in the PDB file. The AMBER *leap* utility adds hydrogens automatically with predefined names. Having

hydrogens in PDB files with names that *leap* does not recognize within its residue libraries leads to a total mess.

Eventually, **remove also all connectivity records**. These are mostly referring to ligands, or, in some cases, to disulfide links. The latter should be explicitly re-connected (see later) without relying on connectivity records in the PDB file.

The final PDB file of the protein should only contain unique locations¹¹ for heavy atoms of amino acids (and maybe oxygens of specific water molecules). Missing atoms in amino acids are mostly allowed since *leap* can rebuild them if the **residue names** are **correct** and if the **atoms** already present have **correct names** also.

Make use of "TER" records to separate parts in the PDB file which are not connected covalently. This is especially important in protein structures in which parts are missing (gaps). Not separating the loose ends by a "TER" record may lead to strange (and wrong) behavior in *leap* or later in the simulations. Apply the same rule to individual water molecules which you want to keep and separate each water by a "TER" record.

5.8.2. Special Residues, Name Conventions, Chain Terminations

Tautomeric and protonation states are not rendered in PDB files. If a defined state for a residue is required, its **name** in the PDB file must reflect the choice. The following subsections deal with these cases. **Important:** if you change a residue name in a PDB file, make sure to change it for **all** atoms of that residue!

Note also that PDB files written out by *leap* will keep the "special" names, which sometimes leads to annoying effects in software packages which are not prepared for amino acids called HIE, HIP, CYX, and alike. You might consider to change these names back to the standard prior to using these PDB files in other software packages. You can also use the *ambpdb* AMBER utility to do that (see the original AMBER documentation for details on this tool).

5.8.2.1. Histidine: HID, HIE, HIP

Histidine can exist in three forms (δ , ϵ , and protonated). The PDB file must reflect the choice of the user. In the current versions of *leap* command files included with AMBER, ϵ -histidine is the default, i.e., a "HIS" residue in a PDB file will be translated automatically to HIE (for ϵ -histidine). If the residue is called "HID" in the PDB file, the resulting residue for AMBER will become δ -histidine, while "HIP" will yield the protonated form.

5.8.2.2. Cysteine: CYS, CYX

Cysteine can exist in free form or as part of a disulfide bridge. PDB residues named "CYS" are automatically converted into a free cysteine with a SH side chain end. If the cysteine is known to be in a **S-S bridge**, the residue name in the PDB file **must** be "CYX". In that case, no hydrogen is automatically added to the side chain which ends in a bare sulfur. However, S-S bonds to pairing cysteines are not automatically made but must be specified by the user. The

¹¹In some PDB files, the same amino acid may be represented by different states (conformations). You must decide which unique location you want to use later in the simulations.

pytleap Python script described in section 5.3 takes care of this through a special command line option and a file specifying which residues are to be connected (page 119).

5.8.2.3. Protonation: ASH, GLH, LYN

Sometimes the usually charged residues aspartate "ASP", glutamate "GLU", and lysine "LYS" might have to be used in their uncharged form. The residue names must then be changed to "ASH", "GLH", and "LYN", respectively. A neutral form of **arginine** is not foreseen in AMBER (as the pKa of arginine is around 12, it is always considered protonated).

5.8.2.4. Terminals: ACE, NHE, NME

There are special **N- and C-terminal cap residues** which can be used to neutralize the N- and C-terminal in peptide chains when the defaults (NH_3^+ for the N-terminal and COO^- for the C-terminal) are not appropriate.

The "ACE" residue [$-C(=O)-CH_3$] can be used to cap the N-terminal. The PDB entry of the capping residue ACE (this **name** is **compulsory**) must be:

ATOM	1	CH3	ACE	resnumber	x	y	z
ATOM	2	C	ACE	resnumber	x	y	z
ATOM	3	O	ACE	resnumber	x	y	z

Note the **atom name "CH3"** for this special carbon! Another name is not allowed! Hydrogens should be omitted. They are automatically added if the residue name and the heavy atom names are correct.

For capping the C-terminal, two possibilities are given. The first one is a simple NH_2 termination giving [$C(=O)-NH_2$]. This residue **must** be called "NHE" in the PDB file and consists of a single N to be named N:

ATOM	1	N	NHE	resnumber	x	y	z
------	---	---	-----	-----------	---	---	---

The second possible C-terminal cap is $NH-CH_3$, resulting in [$C(=O)-NH-CH_3$] at the C-terminal. Its entry in the PDB file **must** have the residue name "NME" and has the following PDB entry:

ATOM	1	N	NME	resnumber	x	y	z
ATOM	2	CH3	NME	resnumber	x	y	z

As above for "ACE", the atom name for the carbon must be "CH3"! "NHE" and "NME" residues are automatically completed with hydrogens. Do not enter them explicitly.

Important: The "ACE" residue should be the first residue in a chain (strand) while "NHE" or "NME" should be the last. If cap residues are used to terminate gaps in incomplete protein chains, they must appear at the exact gap location, respecting N-terminal and C-terminal order. Gaps must be separated by a "TER" record in the PDB file. See section 5.8.3.

5.8.3. Chains, Residue Numbering, Missing Residues

AMBER preparation modules assume that residues in a PDB file are connected and appear sequentially in the file. If not covalently connected (i.e., linked by an amide bond), the residues must be separated by "TER" records in the PDB file. Thus for example, a protein consisting of two chains should have a "TER" record after the final residue of the first chain. Similarly, if residues are missing (e.g., not detected in x-ray, or cut by the user), the gap should also be separated by a "TER" record. Terminal residues will be charged by default. If the user wants to avoid this (especially for gaps), these residues should be capped (by ACE and NHE or NME).

In general, *leap* and tools calling it refer to the original **input residue numbers**. Thus, residues are numbered (rather "named") according to the original PDB file for special commands like the disulfide connections.

Important: In some PDB files, residue numbers are not following a simple sequential scheme. There may be added 'numbers' if the residue numbering should globally reflect that of a 'mother' protein of a whole family. In such cases, you may encounter residue numberings like e.g. 11.. 12.. 12A.. 12B.. 13.. etc, where 12A and 12B are insertions. This may lead to serious trouble when trying to refer to residue 'numbers' or 'names'. The safest way to avoid trouble is then to renumber the residues sequentially (without insertion or deletion letters) before using them in any tool that requires a precise reference to a residue name/number.

In **output files** from *leap* and related tools, **residues will always be numbered starting from 1**, irrespective of the original numbering. Gaps are not considered either. Thus if a protein chain runs from 21 to 80, with residues 31 to 40 (i.e., 10 residues) missing, the final numbering of residues will run from 1 to 50.

Important: The final residue numbers are the ones that must be used in later simulations to refer to individual residues via **AMBER masks** or **NAB atom expressions**. For example, if a protein chain with residues from 30 to 110 is prepared for AMBER simulations, the final numbering will go from 1 to 81. If the original residues 35 to 40 should be fixed or tethered, the actual residues to be specified are 6 to 11. This can lead to serious errors. So be careful about residue numbers. The script *pytleap* described later will always generate a new PDB file with exact AMBER residue numbering and atom names. This PDB file should be used as reference throughout all subsequent AMBER simulations. Above all, when using atom masks or atom expressions (see Appendix 5.9), always check that they really refer to the desired atoms before running lengthy simulations. **Fixing or tethering wrong atoms are a common error which may easily go unnoticed.**

5.9. Appendix B: Atom and Residue Selections

There are two standards to select atoms and residues in AMBER-related routines: the **AMBER "mask"** notation, used by all original AMBER modules, and the **NAB "atom expressions"**, which work only with NAB-compiled applications.

Users who only use the NAB routines presented in this document may skip to section 5.9.2. Those who intend to use original AMBER routines should also become familiar with the AMBER masks notations.

5.9.1. Amber Masks

A "mask" is a notation which selects atoms or residues for special treatment. A frequent usage is fixing or tethering selected atoms or residues during minimization or molecular dynamics.

The following lines are partially copied from the original AMBER documentation. For more details, refer to the entire section of that documentation describing the *ambmask* utility.¹²

The "mask" selection expression is composed of "elementary selections". These start with ":" to select by residues, or "@" to select by atoms. Residues can be selected by numbers (given as numbers separated by commas, or as ranges separated by a dash) or by names (given as a list of residue names separated by commas). The same holds true for atom selections by atom numbers or atom names. In addition, atoms can be selected by AMBER atom type, in which case "@" must be immediately followed by "%". The notation ":*" means all residues and "@*" means all atoms. The following examples show the usage of this syntax.

5.9.1.1. Residue Number List Examples

```
:1-10      = "residues 1 to 10"  
:1,3,5     = "residues 1, 3, and 5"  
:1-3,5,7-9 = "residues 1 to 3 and residue 5 and residues 7 to 9"
```

5.9.1.2. Residue Name List Examples

```
:LYS      = "all lysine residues"  
:ARG,ALA,GLY = "all arginine and alanine and glycine residues"
```

5.9.1.3. Atom Number List Examples

Note that these masks use the **actual sequential numbers of atoms** in the file. This is tricky and a serious source of error. You must know these numbers correctly. Using the atom numbers of a PDB file written out by an AMBER tool is an appropriate way to avoid pitfalls. **Do not use the original atom numbers from the raw PDB file you started with.**

```
@12,17      = "atoms 12 and 17"  
@54-85      = "all atoms from 54 to 85"  
@12,54-85,90 = "atom 12 and all atoms from 54 to 85 and atom 90"
```

5.9.1.4. Atom Name List Examples

Atom names follow the standard names in PDB files for heavy atoms. For hydrogen atom names with more than 3 characters, the choice may be critical since some AMBER tools¹³ wrap hydrogen atom names in the PDB files they write out, but internally use the "unwrapped" name

¹²The utility *ambmask* is not part of the free Amber Tools but is available only together with the full AMBER package.

¹³Even that is not consistent because NAB-compiled routines use the unwrapped notation.

version. For example, the second hydrogen atom at the first C γ (e.g., in isoleucine) would be called HG12, but in the official PDB notation, it would be 2HG1. Since it very rarely (actually never) makes sense to fix individual hydrogen atoms in side chains, we do not worry about this. Even in ligand names, hydrogens are generally not the first choice of selection when fixing or tethering parts of the ligand.

```
@CA          = all atoms with the name CA (i.e., all C-alpha atoms)
@CA,C,O,N,H = all atoms with names CA or C or O or N or H
              (i.e., the entire protein backbone)
```

5.9.1.5. Atom Type List Examples

This last mask type is only used by specialists and mentioned here for completeness. It allows the selection of AMBER atom types and requires detailed knowledge of AMBER force fields.

```
@%CT          = all atoms with the force field type CT
                (the standard sp3 aliphatic carbon)
@%N*,N3       = all atoms with the force field type N* or N3
                (N* is a special sp2 nitrogen, N3 is an sp3 nitrogen)
```

Note that in the above example, N* is actually an atom type. The * is **not** a wild card meaning "all N-something types"!

5.9.1.6. Logical Combinations

The selections above can be combined by various logical operators, including selections like "all atoms within a certain distance from...". The use of such combinations goes beyond this introductory script. Interested users should refer to the original AMBER documentation.

5.9.2. "Atom Expressions" in NAB Applications

NAB applications do not use the AMBER mask scheme outlined in the previous sections. They use simpler (but less powerful) selection criteria. The scheme is:

```
chains(or "strands"):residues:atoms
```

For example, A:GLU:CA would select all C α carbons of all glutamate residues in chain A. A plain :: would select all atoms in all residues and all chains (not very useful). ::H* would select all hydrogen atoms in any chain and any residue, the * being a wild card for any sequence of characters. Similarly, ::*C* would select all atoms which contain at least one "C" character, i.e., the wild card can be used in any position. The ? can be used as a wild card for a single character. Thus, ::H? would select any atom starting with H plus one additional character (e.g., HC, H1, HN, but **not** HG11).

The wild card can also be used in residue names. :A*: would select all alanines, asparagines, and arginines.

Selections can be combined separated by a vertical bar "|". :1-3,ALA:C*|:2-5:N* would select all carbon atoms in residues 1 to 3, in all alanines **and** all nitrogen atoms in all residues

5. amberlite: Some AMBER-Tools-Based Utilities

from 2-5. If you would like to tether all C α atoms of a protein and the oxygen atom of explicit water molecules (with residue names 'WAT'), you would use `::CA|:WAT:O*`.

Output from NAB applications always tells how many atoms have been selected for a special treatment. If you are not sure that your selection is correct, this number might at least be a hint. If you run a simulation with a protein having 200 residues and want to tether all C α carbons, `::CA` should result in 200 selected atoms (provided that all residues have a well-defined CA atom, which they should).

5.10. Appendix C: Examples and Test Cases

5.10.1. Example 1: Generating AMBER Files for Crambin with Disulfide Bonds

In *crambin* (`1CRN.pdb`, ...`amberlite/examples/CRN`), there are 3 disulfide bonds connecting CYS3 to CYS40, CYS2 to CYS32, and CYS16 to CYS26. In the PDB file, these residues must all be changed from CYS to CYX. Then a text file (e.g. `sslinks`) should be created that looks like this:

```
3    40
2    32
16   26
```

In the `CRN` `examples` subfolder, the file `1crnx.pdb` is the modified `1CRN.pdb` file with the six cysteines above changed to CYX in their residue name. Also, everything has been removed except the `ATOM` records. Since we create explicitly the disulfide bonds via the `bond` command in *leap*, the connectivity records have been discarded also.

The **correct** command should be (assuming defaults for most settings):

```
pytleap --prot 1crnx.pdb --disul sslinks
```

where `sslinks` specifies the text file containing the numbers of the residues to be S-S linked (one pair per line). Now the disulfide bonds are recognized and registered in the PRM file, i.e., all bonded interactions for $-CH_2-S-S-CH_2-$ are correctly computed.

The file `leap.cmd` generated by *pytleap* shows the bonding between the corresponding SG atoms in the three disulfide linkages on lines 2 to 5:

```
set default pbradii mbondi
prot = loadpdb 1crnx.pdb
bond prot.3.SG prot.40.SG
bond prot.4.SG prot.32.SG
bond prot.16.SG prot.26.SG
saveamberparm prot 1crnx.leap.prm 1crnx.leap.crd
savepdb prot 1crnx.leap.pdb
quit
```

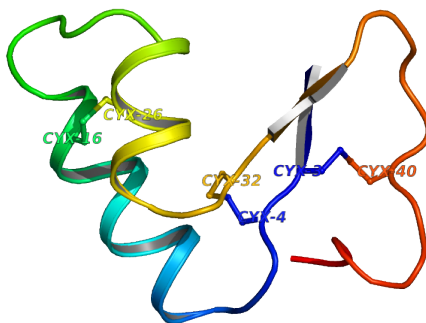


Figure 5.1.: The 3 S-S links in 1crn.pdb

5.10.2. Example 2: Energy Minimization of the Crambin Structure

5.10.2.1. Starting Energy

We can use the files *1crnx.leap.prm* and *1crnx.leap.pdb* which were created in section 5.10.1 to evaluate the AMBER energy terms in the unrefined crambin structure with the Generalized-Born option 1 and the SASA (nonpolar) energy:

```
ffgbsa 1crnx.leap.pdb 1crnx.leap.prm 1 1
```

The result is:

```
Reading parm file (1crnx.leap.prm)
title:

mm_options: cut=100
mm_options: rgbmax=100
mm_options: diel=C
mm_options: gb=1
  iter   Total      bad    vdW    elect   nonpolar   genBorn    frms
ff:      0   -813.56   611.05   -92.09   -980.60     0.00    -351.91   1.52e+01
sasa:    3079.71
Esasa = 0.0072 * sasa =      22.17
```

In this output, the line starting with "ff:" lists the total energy of the system and the components (bad = bond-angle-dihedral combined energy, i.e., the sum of the bonded terms). The line starting with "sasa:" gives the solvent-accessible surface in Å². The final line is the result from SASA multiplied by a surface tension of 0.0072. All energies are in kcal·mol⁻¹.¹⁴

This procedure is a good (although rough) health check of the PRM/PDB (and corresponding PRM/CRD) file pairs prior to using them in longer simulations. If the starting structure file is considered of good quality (no major steric bumps) but some of the values reported by

¹⁴Note that the "nonpolar" term in the main energy components line is 0.00 because we compute this term separately. The "nonpolar" term in NAB applications can also include other terms (e.g., restraints) and is sometimes misleading.

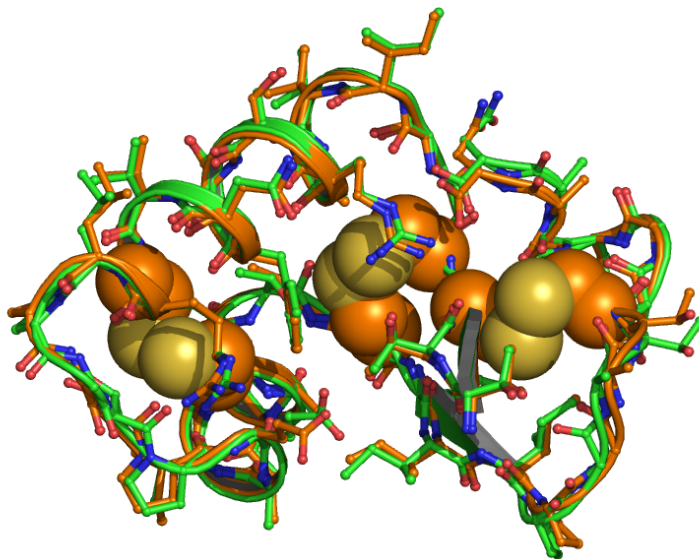


Figure 5.2.: Starting (green) and refined (orange) coordinates of 1CRN. Disulfide bonds in the refined structure are shown in CPK mode.

ffgbsa look weird, there might be a serious error in the PRM file. If the coordinate file and the parameter-topology file are incompatible, e.g., different number of atoms or different order of atoms, *ffgbsa* will give very strange results in most cases (or fail completely).

5.10.2.2. Energy Minimization with *minab*

The structure refinement via conjugate gradient minimization can be carried out by the command (all on one line):

```
minab 1crnx.leap.pdb 1crnx.leap.prm crambin.min.pdb 1 1000  
> crambin.min.out &
```

We use the GB option 1 and request a maximum of 1000 steps. No restraints are applied. The refined coordinates go to the PDB file *crambin.min.pdb*. The output of *minab* is redirected to the text file *crambin.min.out*. The final '&' puts the process into the background. The minimization can be followed interactively by the command `tail -f crambin.min.out`

The last lines of the output are:

```
-----  
initial energy: -814.840 kcal/mol  
final   energy: -1093.158 kcal/mol
```

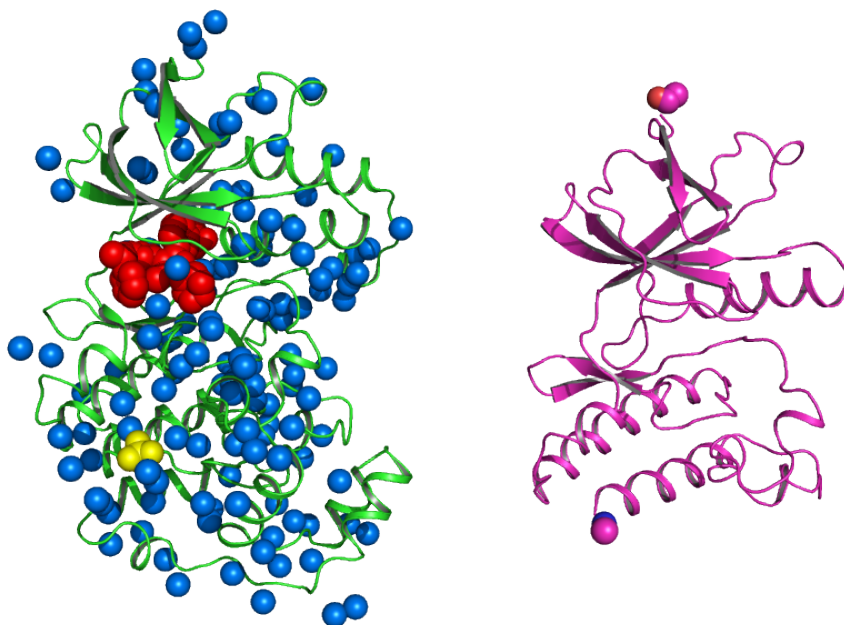


Figure 5.3.: *Left side: original structure 10UK.pdb with ligand (red), sulfate (yellow) and water molecules (blue); right side: final structure p38.pdb with the N- and C-terminal caps.*

```
minimizer finished after 619 iterations
refined coordinates written to crambin.min.pdb
-----
```

Figure 5.2 shows the initial (green) and refined (orange) structure of crambin. The disulfide bonds in the refined structure are shown in CPK mode to emphasize that the S-S links have been correctly assigned. If this were not the case, the respective sulfur atoms would drift apart because of non-bonded interactions.

5.10.3. Example 3: Preparation of a Complex between P38 MAP Kinase and Ligand

5.10.3.1. Cleaning Up PDB Entry 10UK.pdb for Usage with AMBER

In the `...amberlite/examples/P38` directory, the PDB file `10UK.pdb` (P38 MAP kinase with inhibitor) is included in its original version. The structure (see Figure 5.3) contains a ligand (red), a sulfate ion (yellow), and a number of water molecules (blue). The file `p38.pdb` (also included in `...amberlite/examples/P38`) was created from this PDB file by cutting off a large part of the protein and deleting everything except the heavy atoms. The resulting "nonnatural"

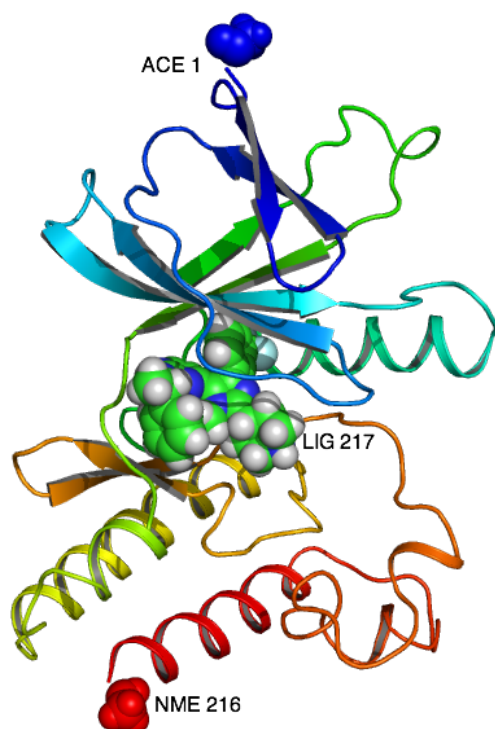


Figure 5.4.: The final complex structure *com.leap.pdb*: The ACE cap becomes residue 1, the NME cap is residue 216, and the ligand is residue LIG 217

N- and C-terminal were then completed by ACE and NME caps, respectively.¹⁵ The resulting PDB file is "clean" for *leap* and passes without errors. The ligand was processed separately into SDF format (file *lig.sdf* in ...amberlite/examples/P38) including all hydrogens and bond orders. This file is ready to be processed via *antechamber* before re-complexing it with the protein (see later).

5.10.3.2. Generating AMBER Files for a P38/Ligand Complex

We re-use as a receptor the reduced and corrected PDB file from section 5.10.3.1, *p38.pdb*. For the ligand, we use the *lig.sdf* file, containing the ligand with its heavy-atoms coordinates from the original pdb file *IOUK.pdb* and hydrogen atoms added via any other software that can handle this kind of problem. Note that the ligand has a formal charge of +1.

The following command line will create the PRM, CRD, and PDB files for the empty receptor, the ligand, and the complex; partial charges on the ligand will be computed via the AM1-BCC method;[71, 72] the complex will be named *com*:

¹⁵This can be done by any modeling software which allows building, but make sure that the final atom and residue names in the cap residues are those described in section 5.8.2.4.


```
pytleap --prot p38.pdb --lig lig.sdf --chrg 1 --cplx com
```

The longest part in the execution time is the processing of the ligand via the *sqm* tool to get the AMB1-BCC charges. During execution, various temporary files appear in the working directory. They result from the different modules called in *antechamber*. Most are removed when *antechamber* has finished.

The resulting AMBER files are called *.leap.prm, *.leap.crd, and *.leap.pdb, respectively. One set of files is generated for the ligand (*lig*), the receptor (*p38*), and the complex (*com*).¹⁶

Among the various other files left over, *lig.leap.frcmod* might be the most relevant to inspect since it contains parameters which were used in addition to those explicitly present in the *gaff* parameter set.

The file *lig.ac.mol2* is the MOL2 for the ligand containing the *gaff* atom types and the AM1-BCC partial charges. This file can be read by a variety of software packages but the atom elements will not be recognized because atom types are not original TRIPOS atom types, indicating the chemical element.

In the resulting complex, the ligand has the residue name LIG and the residue number 217. The N- and C-terminal caps ACE and NME get residue numbers 1 and 216.

5.10.4. Example 4: Interaction Energy between P38 and Ligand in the Unrefined (Original) Complex

We use the files generated in example 3 (section 5.10.3.2). In order to make use of the default settings in the command line options, we copy the respective files to the default names proposed by *pymdpbsa*: Make copies (or symbolic links) of *com.leap.prm*, *p38.leap.prm*, and *lig.leap.prm* to *com.prm*, *rec.prm*, and *lig.prm*. Also, make a copy (or symbolic link) of *com.leap.pdb* to *com.pdb*.

Now the command

```
pymdpbsa --proj RAWPDB --traj com.pdb
```

computes the interaction energy. As "trajectory" (*--traj*) we specify the single complex PDB file *com.pdb*. We call the project "RAWPDB" and leave all other input options at their default, i.e., we also use the default GB option 1.

A subdirectory *RAWPDB_XXXXXX.tmpdir* is generated, where *XXXXXX* is a random sequence of characters. This temporary directory can be removed since the relevant output files are copied to the directory in which *pymdpbsa* was started. We could also have used the additional command line option *--clean* to remove the temporary directory automatically.

The output of interest is the summary file *RAWPDB.sum* (see also 5.7.5.1 and 5.7.5.2):

```
=====
Summary Statistics for Project RAWPDB
Frames           : 1 to 1 (every 1)
Solvation        : GB (--solv=1)
```

¹⁶Note that we use the *.leap.* name giving to underline that these files have been generated via leap. This is useful to avoid confusion, especially for the PDB or CRD files which must correspond to the respective PRM files.

```

Trajectory File      : com.pdb
Complex parmtop File : com.prm
Receptor parmtop File : rec.prm
Ligand parmtop File : lig.prm
=====
----Ligand Energies-----
Etot =   -127.64 (  0.00,  0.00) Ebat =    117.53 (  0.00,  0.00)
Evdw =     3.79 (  0.00,  0.00) Ecoul =   -183.01 (  0.00,  0.00)
EGB  =   -71.71 (  0.00,  0.00) Esasa =     5.76 (  0.00,  0.00)
----Receptor Energies-----
Etot =  -4072.40 (  0.00,  0.00) Ebat =   2653.90 (  0.00,  0.00)
Evdw =    657.95 (  0.00,  0.00) Ecoul =  -4409.58 (  0.00,  0.00)
EGB  =  -3068.98 (  0.00,  0.00) Esasa =    94.31 (  0.00,  0.00)
----Complex Energies-----
Etot =  -4250.68 (  0.00,  0.00) Ebat =   2771.43 (  0.00,  0.00)
Evdw =    608.82 (  0.00,  0.00) Ecoul =  -4636.90 (  0.00,  0.00)
EGB  =  -3086.27 (  0.00,  0.00) Esasa =    92.24 (  0.00,  0.00)
----Interaction Energy Components-----
Etot =    -50.64 (  0.00,  0.00) Ebat =         0.00 (  0.00,  0.00)
Evdw =   -52.92 (  0.00,  0.00) Ecoul =   -44.31 (  0.00,  0.00)
EGB  =    54.42 (  0.00,  0.00) Esasa =    -7.83 (  0.00,  0.00)
=====

```

5.10.5. Example 5: Minimization of P38 Complex with minab and Resulting Interaction Energy

We minimize the P38/ligand complex prepared in section 5.10.3. We use the (renamed) PDB file *com.pdb*, the corresponding PRM file *com.prm*, *gb* = 1 and a maximum of 500 iterations. We tether *Cα* atoms with a force constant of $1.0 \text{ kcal}\cdot\text{mol}^{-1}\cdot\text{\AA}^{-2}$. The refined coordinates are written to *com.min.pdb*. We redirect the output to a file *minab.out*.

For the command line

```
minab com.pdb com.prm com.min.pdb 1 500 '::CA' 1.0 > minab.out &
```

the output file *minab.out* would be:

```

Reading parm file (com.prm)
title:

mm_options:  cut=100.000000
mm_options:  nsnb=501
mm_options:  diel=C
mm_options:  gb=1
mm_options:  rgbmax=15.000000
mm_options:  wcons=1.000000
mm_options:  ntp=10
constrained 214 atoms using expression ::CA

```

```

constrained 214 atoms from input array
      iter      Total      bad      vdW      elect      nonpolar      genBorn      frms
ff:      0 -4378.52  2771.43   608.82 -4636.90      0.00 -3121.87  3.31e+01
ff:     10 -5630.74  2669.08 -446.65 -4721.49      0.11 -3131.79  4.86e+00

{...more like this cut from this demo output...}

ff:    490 -6861.80  2521.23 -1205.14 -5170.41     16.59 -3024.07  1.67e-01
ff:    500 -6862.46  2521.87 -1205.86 -5170.56     16.23 -3024.15  1.41e-01
-----
initial energy: -4378.522 kcal/mol
final   energy: -6862.463 kcal/mol
minimizer stopped because number of iterations was exceeded
refined coordinates written to com.min.pdb
-----

```

The minimization did not reach the requested default rms of the components of the gradient of 0.1, but stopped after the required 500 iterations.

The final line reminds that the refined structure has been saved into the PDB file `com.min.pdb`. Note that the energy term listed here under "nonpolar" is actually the energy stemming from the restraints, in this example tethering all C α atoms.

We can now repeat the interaction energy computation on the refined complex, using the same settings as for the raw PDB file in section 5.10.4:

```
pymdpbsa --proj REFINEDPDB --traj com.min.pdb
```

with `--traj` now specifying the refined PDB file `com.min.pdb`. The resulting summary `REFINEDPDB.sum` is:

```

=====
Summary Statistics for Project MINPDB
Frames           : 1 to 1 (every 1)
Solvation        : GB (--solv=1)
Trajectory File  : com.min.pdb
Complex parmtop File : com.prm
Receptor parmtop File : rec.prm
Ligand  parmtop File : lig.prm
=====
----Ligand Energies-----
Etot =   -210.13 ( 0.00, 0.00) Ebat =    34.74 ( 0.00, 0.00)
Evdw =    15.06 ( 0.00, 0.00) Ecoul =   -195.36 ( 0.00, 0.00)
EGB  =   -70.16 ( 0.00, 0.00) Esasa =    5.61 ( 0.00, 0.00)
----Receptor Energies-----
Etot =  -6470.57 ( 0.00, 0.00) Ebat =   2487.56 ( 0.00, 0.00)
Evdw =  -1160.97 ( 0.00, 0.00) Ecoul =  -4904.64 ( 0.00, 0.00)
EGB  =  -2988.22 ( 0.00, 0.00) Esasa =    95.70 ( 0.00, 0.00)
----Complex Energies-----
Etot =  -6750.93 ( 0.00, 0.00) Ebat =   2522.30 ( 0.00, 0.00)
Evdw =  -1205.93 ( 0.00, 0.00) Ecoul =  -5170.51 ( 0.00, 0.00)

```

```

EGB   =   -2990.31 (  0.00,   0.00) Esasa =    93.52 (  0.00,   0.00)
-----Interaction Energy Components-----
Etot   =   -70.25 (  0.00,   0.00) Ebat  =    0.00 (  0.00,   0.00)
Evdw   =   -60.02 (  0.00,   0.00) Ecoul =   -70.51 (  0.00,   0.00)
EGB    =    68.07 (  0.00,   0.00) Esasa =   -7.79 (  0.00,   0.00)
=====

```

5.10.6. Example 6: Generate MD Trajectory for the P38-Ligand Complex with *mdnab*

We use *mdnab* to run a 100 picoseconds MD trajectory of P38 complex, using as starting geometry the refined complex `com.min.pdb` from the previous section:

```
mdnab com.min.pdb com.prm com 1 100 '::

```

The trajectory will go to the file `com.binpos`, specified as the third command line argument (the extension ".binpos" is appended automatically). We tether the C α atoms with the same force as for the minimization in 5.10.5 through the last two arguments '::mdnab command).

The file `md.out` will start with:

```

Reading parm file (com.prm)
title:

mm_options:  cut=12.000000
mm_options:  nsnb=25
mm_options:  diel=C
mm_options:  gb=1
mm_options:  rgbmax=12.000000
mm_options:  rattle=1
mm_options:  dt=0.002000
mm_options:  ntpr=101
mm_options:  ntpr_md=10
mm_options:  ntwx=0
mm_options:  zerov=0
mm_options:  tempi=50.000000
mm_options:  temp0=100.000000
mm_options:  gamma_ln=20.000000
mm_options:  wcons=1.000000
constrained 214 atoms using expression ::CA

```

The last two lines shown above indicate that all C α atoms (214 in this case) have indeed been tethered with a force constant `wcons=1.000000`. It is important to verify this line to make sure that the atom selection on the command line (in this case '::

The output file `md.out` then continues through the heat-up and equilibration stages. Then the time is reset to zero and the production phase begins. The final lines in the example above are:

```
...
...
md:      49500      99.000      2137.72    -4563.14    -2425.42      302.35
md:      50000     100.000      2084.67    -4519.03    -2434.36      294.84

trajectory with 100 picoseconds was written to com.binpos...
```

confirming that 50000 steps (i.e. 100 picoseconds with a stepsize of 2 femtoseconds) have been recorded and written to the trajectory file `com.binpos`.

5.10.7. Example 7: Running *pymdpbsa* on the P38/Ligand Complex Trajectory

If we have previously renamed all PRM files to the expected defaults, since the ligand is called "LIG" by default in *pytleap*, and since we want the default GB 1 option, we only enter the project name P38. We use every tenth frame from the total 100-frames production phase of the trajectory, so that `--start 10`, `--stop 100`, and `--step 10` are used.

The command line is:

```
pymdpbsa --proj P38 --traj com.binpos --start 10 --stop 100 --step 10
```

The summary of the results goes into the file `P38.sum` and is shown below.

```
=====
Summary Statistics for Project P38
Frames           : 10 to 100 (every 10)
Solvation        : GB (--solv=1)
Trajectory File   : com.binpos
Complex parmtop File : com.prm
Receptor parmtop File : rec.prm
Ligand parmtop File : lig.prm
=====
-----Ligand Energies-----
Etot =   -171.86 (  3.80,  1.20) Ebat =    65.11 (  3.96,  1.25)
Evdw =    19.74 (  2.06,  0.65) Ecoul =   -191.30 (  2.13,  0.67)
EGB  =   -71.04 (  0.68,  0.22) Esasa =    5.62 (  0.05,  0.01)
-----Receptor Energies-----
Etot =  -4246.59 ( 39.66, 12.54) Ebat =   4156.11 ( 34.62, 10.95)
Evdw =   -772.67 ( 20.92,  6.61) Ecoul =  -4921.45 ( 38.73, 12.25)
EGB  =  -2804.11 ( 27.70,  8.76) Esasa =    95.53 (  0.57,  0.18)
-----Complex Energies-----
Etot =  -4478.81 ( 40.64, 12.85) Ebat =   4221.22 ( 36.59, 11.57)
Evdw =   -806.44 ( 22.34,  7.07) Ecoul =  -5161.88 ( 42.37, 13.40)
EGB  =  -2825.07 ( 30.74,  9.72) Esasa =    93.36 (  0.60,  0.19)
-----Interaction Energy Components-----
Etot =   -60.36 (  3.30,  1.04) Ebat =   -0.00 (  0.01,  0.00)
Evdw =   -53.51 (  3.65,  1.15) Ecoul =   -49.14 ( 12.02,  3.80)
```

5. *amberlite*: Some AMBER-Tools-Based Utilities

```
EGB      =      50.08 ( 10.21,   3.23) Esasa =      -7.80 (   0.12,   0.04)
=====
```

The numbers in parentheses after the actual energy values are the standard deviation and the standard error of the mean (SEM). Note that the energy term E_{bat} (the sum of the bond, angle, and torsion terms) for the interaction energy is zero (or almost so, because of rounding errors). This is the obvious consequence of the single-trajectory approach because we neglect any strain in the ligand or the receptor. The strain would have to be evaluated by running three distinct trajectories (i.e., also for the free ligand and the empty receptor).

A directory `P38_XXXXXX.tmpdir` has been created which contains all files used for the computation, including the individual structures of each frame. You can safely remove this directory if you are only interested in the actual results, i.e., the summary file `*.sum` and the `*.X.nrg` tables, where `X` can be C (for complex), R (for receptor), L (for ligand), and D (for the actual ΔE and ΔG values).

6. sqm: Semi-empirical quantum chemistry

AmberTools now contains its own quantum chemistry program, called *sqm*. This is code extracted from the QM/MM portions of *sander*, but is limited to “pure QM” calculations. A principal current use is as a replacement for MOPAC for deriving AM1-bcc charges, but the code is much more general than that. Right now, it is limited to carrying out single point calculations and energy minimizations (geometry optimizations) for closed-shell systems. It supports a wide variety of semi-empirical Hamiltonians, including many recent ones. An external electric field generated by a set of point charges can be included for single point calculations. Our plan is to add capabilities to subsequent versions. The major contributors are as follows:

- The original semi-empirical support was written by Ross Walker, Mike Crowley, and Dave Case,[91] based on public-domain MOPAC codes of J.J.P. Stewart.
- SCC-DFTB support was written by Gustavo Seabra, Ross Walker and Adrian Roitberg,[92] and is based on earlier work of Marcus Elstner.[93, 94]
- Support for third-order SCC-DFTB was written by Gustavo Seabra and Josh McClellan.
- Various SCF convergence schemes were added by Tim Giese and Darrin York.
- The PM6 Hamiltonian was added by Andreas Goetz and dispersion and hydrogen bond corrections were added by Andreas Goetz and Kyoyeon Park.
- The extension for MNDO type Hamiltonians to support d orbitals was written by Tai-Sung Lee, Darrin York and Andreas Goetz.
- The charge-dependent exchange-dispersion corrections of vdW interactions[95] was contributed by Tai-Sung Lee, Tim Giese, and Darrin York.
- The ability of reading user-defined parameters was added by Tai-Sung Lee and Darrin York.

6.1. Available Hamiltonians

Available MNDO-type semi-empirical Hamiltonians are PM3,[96] AM1,[97] RM1,[98] MNDO,[99] PDDG/PM3,[100] PDDG/MNDO,[100] PM3CARB1,[101], PM3-MAIS[102, 103], MNDO/d[104–106], AM1/d (Mg from AM1/d[107] and H, O, and P from AM1/d-PhoT[108]) and PM6[109].

6. *sqm*: Semi-empirical quantum chemistry

Support is also available for the Density Functional Theory-based tight-binding (DFTB) Hamiltonian,[92, 110, 111] as well as the Self-Consistent-Charge version, SCC-DFTB.[93] DFTB/SCC-DFTB also supports approximate inclusion of dispersion effects,[112] as well as reporting CM3 charges [113] for molecules containing only the H, C, N, O, S and P atoms and third-order corrections[114].

The elements supported by each QM method are:

- MNDO: H, Li, Be, B, C, N, O, F, Al, Si, P, S, Cl, Zn, Ge, Br, Cd, Sn, I, Hg, Pb
- MNDO/d: H, Li, Be, B, C, N, O, F, Na, Mg, Al, Si, P, S, Cl, Zn, Ge, Br, Sn, I, Hg, Pb
- AM1: H, C, N, O, F, Al, Si, P, S, Cl, Zn, Ge, Br, I, Hg
- AM1/d: H, C, N, O, F, Mg, Al, Si, P, S, Cl, Zn, Ge, Br, I, Hg
- PM3: H, Be, C, N, O, F, Mg, Al, Si, P, S, Cl, Zn, Ga, Ge, As, Se, Br, Cd, In, Sn, Sb, Te, I, Hg, Tl, Pb, Bi
- PDDG/PM3: H, C, N, O, F, Si, P, S, Cl, Br, I
- PDDG/MNDO: H, C, N, O, F, Cl, Br, I
- RM1: H, C, N, O, P, S, F, Cl, Br, I
- PM3CARB1: H, C, O
- PM3-MAIS: H, O, Cl
- PM6: H, He, Li, Be, B, C, N, O, F, Ne, Na, Mg, Al, Si, P, S, Cl, Ar, K, Ca, Sc, Ti, V, Cr, Mn, Fe, Co, Ni, Cu, Zn, Ga, Ge, As, Se, Br, Kr, Rb, Sr, Cd, In, Sn, Sb, Te, I, Xe, Cs, Ba, Hg, Tl, Pb, Bi
- DFTB/SCC-DFTB: (Any atom set available from the www.dftb.org website)

The PM6 implementation has not been extensively tested for all available elements. Please check your results carefully, possibly by comparison to other codes that implement PM6, in particular if transition metal elements are present. If the PM6 Hamiltonian is used in a QM/MM simulation with *sander* (see Amber manual) then the electrostatic interactions between QM and MM atoms are modeled using the MNDO type core repulsion function for interactions between QM and MM atoms. Parameters for the exponents α of the QM atoms are taken from PM3 (a default value of five is used for the exponents α of the MM atoms as is the case for MNDO, AM1 and PM3). Since PM3 does not have parameters for all elements that are supported by PM6, the missing exponents were defined in an ad hoc manner (see the source code in \$AMBERHOME/AmberTools/src/sqm/qm2_parameters.F90, variable `alp_pm6`). The magnitude of the coefficients α is probably not critical for the accuracy of QM/MM calculations but this should be tested on a case by case basis. This does not affect QM calculations with *sqm*.

The DFTB/SCC-DFTB code was originally based on the DFT/DYLAX code by Marcus Elstner *et al.*, but has since been extensively re-written and optimized. In order to use DFTB (*qm_theory=DFTB*) a set of integral parameter files are required. These are not distributed

with Amber and must be obtained from the www.dftb.org website and placed in the `$AMBERHOME/dat/slko` directory. Dispersion parameters for H, C, N, O, P and S are available in the `$AMBERHOME/dat/slko/DISPERSION.INP_ONCHSP` file, and CM3 parameters for the same atoms are in the `$AMBERHOME/dat/slko/CM3_PARAMETERS.DAT` file. Parameters for two parametrizations of the third-order SCC-DFTB terms, namely SCC-DFTB-PA and SCC-DFTB-PR are distributed with Amber in the files `DFTB_3RD_ORDER_PA.DAT` and `DFTB_3RD_ORDER_PR.DAT`, located in the same directory.

6.2. Charge-dependent exchange-dispersion corrections of vdW interactions

The sqm program provides a new charge-dependent energy model consisting of van der Waals (vdW) and polarization interactions between the quantum mechanical (QM) and molecular mechanical (MM) regions in a combined QM/MM calculation. vdW interactions are commonly treated using empirical Lennard-Jones (L-J) potentials, whose parameters are often chosen based on the QM atom type (e.g., based on hybridization or specific covalent bonding environment). This strategy for determination of QM/MM nonbonding interactions becomes tedious to parametrize and lacks robust transferability. Problems occur in the study of chemical reactions where the “atom type” is a complex function of the reaction coordinate. This is particularly problematic for reactions, where atoms or localized functional groups undergo changes in charge state and hybridization.

In sqm, this charge-dependent energy model was implemented based on a scaled overlap model for repulsive exchange and attractive dispersion interactions that is a function of atomic charge. The model is chemically significant since it properly correlates atomic size, softness, polarizability, and dispersion terms with minimal one-body parameters that are functions of the atomic charge[95].

This “Charge-dependent exchange-dispersion corrections of vdW interactions” can be invoked by the “qxd=.true.” switch in the sqm namelist. Note that this model currently does not have any effect on pure quantum calculations through sqm, the qxd correction is only added to QM/MM interactions in SANDER. The default values of qxd parameters are set to reproduce the regular L-J interactions of typical atom types (HC for H, C* for C, N for N, OW for O, and parameters for F and Cl are optimized[95]) when the charge dependence parameters are zero. There are eight qxd parameters (symbols used in the reference[95] are indicated in the parentheses): qxd_s (s), qxd_z0 ($\zeta(0)$), qxd_zq (ζ_q), qxd_d0 (α_1), qxd_dq ($3 \times B$), qxd_q0 (α_2), qxd_qq ($3 \times B$), and qxd_neff ($N_{eff}(0)$). All parameters can be modified through external user-defined parameter files (see the usage of ‘parameter_file’ in Section 6.4).

6.3. Dispersion and hydrogen bond correction

An empirical dispersion and hydrogen bonding correction is implemented for the MNDO type Hamiltonians AM1 and PM6[115]. The empirical dispersion correction follows the formalism for DFT-D[116] and consists of a physically sound r^{-6} term that is damped at short

6. sqm: Semi-empirical quantum chemistry

distances to avoid the short-range repulsion which can be written as

$$E_{\text{dis}} = -s_6 \sum_{ij} f_{\text{damp}}(r_{ij}, R_{ij}^0) C_{6,ij} r_{ij}^{-6}, \quad (6.1)$$

where r_{ij} is the distance between two atoms i and j , R_{ij}^0 is the equilibrium van der Waals (vdW) separation derived from the atomic vdW radii, $C_{6,ij}$ the dispersion coefficient, and s_6 a general scaling factor. The damping function is given as

$$f_{\text{damp}}(r_{ij}, R_{ij}^0) = \left[1 + \exp \left(-\alpha \frac{r_{ij}}{s_R R_{ij}^0} - 1 \right) \right]^{-1}. \quad (6.2)$$

Bondi vdW radii[64] are used and for a pair of unlike atoms we have

$$R_{ij}^0 = \frac{R_{ii}^{0^3} + R_{jj}^{0^3}}{R_{ii}^{0^2} + R_{jj}^{0^2}}. \quad (6.3)$$

For the C_6 coefficients the following equation is used,

$$C_{6,ij} = 2 \frac{(C_{6,ii}^2 C_{6,jj}^2 N_{\text{eff},i} N_{\text{eff},j})^{1/3}}{(C_{6,ii} N_{\text{eff},j}^2)^{1/3} + (C_{6,jj} N_{\text{eff},i}^2)^{1/3}}, \quad (6.4)$$

where the Slater-Kirkwood effective number of electrons $N_{\text{eff},i}$ and the C_6 coefficients can easily be found in the literature[116].

An empirical hydrogen bonding correction[115] that is transferable among different semiempirical Hamiltonians and has been parametrized for use with the dispersion correction described above is also available. This correction does not make the assumption of a specific acceptor/hydrogen/donor binding situation. Instead it considers the hydrogen bond as a charge-independent atom-atom term between two atoms capable of serving as an acceptor or donor (for example, O, N) and weights this by a function that accounts for the steric arrangement of the two atoms and the favorable positioning of a hydrogen atom inbetween. A damping function corrects for long- and short-range behavior,

$$E_{\text{H-bond}} = \frac{C_{AB}}{r_{AB}^2} f_{\text{geom}} f_{\text{damp}}, \quad (6.5)$$

$$f_{\text{geom}} = \cos(\theta_A)^2 \cos(\phi_A)^2 \cos(\psi_A)^2 \cos(\phi_B)^2 \cos(\phi_B)^2 \cos(\psi_B)^2 f_{\text{bond}}, \quad (6.6)$$

$$f_{\text{bond}} = 1 - \frac{1}{1 + \exp[-60(r_{\text{XH}}/1.2 - 1)]}, \quad (6.7)$$

$$f_{\text{damp}} = \left(\frac{1}{1 + \exp[-100(r_{AB}/2.4 - 1)]} \right) \left(1 - \frac{1}{1 + \exp[-10(r_{AB}/7.0 - 1)]} \right), \quad (6.8)$$

$$C_{AB} = \frac{C_A + C_B}{2}. \quad (6.9)$$

Here, C_A and C_B are the atomic hydrogen bonding correction parameters and the (torsion) angles in the function f_{geom} are defined similarly to an earlier hydrogen bond correction[117].

The hydrogen bond correction can be used both for single point energy calculations or geometry optimizations with SQM and for molecular dynamics simulations with SANDER. However, we do not recommend the use for molecular dynamics at present since cutoffs needed to be implemented for the calculation of f_{geom} of equation (6.6). This and some other conditional evaluations give rise to discontinuities in the potential energy surface and thus make this method unattractive for MD simulations.

6.4. Usage

The *sqm* program uses the following simple command line:

```
sqm [-O] -i <input-file> -o <output-file>
```

As in other Amber programs, the “-O” flag allows the program to over-write the output file.

An example input file for running a simple minimization is shown here:

```
Run semi-empirical minimization
&qmmm
  qm_theory='AM1',    qmcharge=0,
/
  6      CG      -1.9590      0.1020      0.7950
  6      CD1     -1.2490      0.6020     -0.3030
  6      CD2     -2.0710      0.8650      1.9630
  6      CE1     -0.6460      1.8630     -0.2340
  6      C6      -1.4720      2.1290      2.0310
  6      CZ      -0.7590      2.6270      0.9340
  1      HE2     -1.5580      2.7190      2.9310
 16      S15     -2.7820      0.3650      3.0600
  1      H19     -3.5410      0.9790      3.2740
  1      H29     -0.7870     -0.0430     -0.9380
  1      H30      0.3730      2.0450     -0.7840
  1      H31     -0.0920      3.5780      0.7810
  1      H32     -2.3790     -0.9160      0.9010
```

The *&qmmm* namelist contains variables that allow you to control the options used. Following that is one line per atom, giving the atomic number, atom name, and Cartesian coordinates (free format). The variables in the *&qmmm* namelist are these:

qm_theory Level of theory to use for the QM region of the simulation (Hamiltonian). Default is to use the semi-empirical Hamiltonian PM3. Options are AM1, RM1, MNDO, PM3-PDDG, MNDO-PDDG, PM3-CARB1, MNDO/d (same as MNDO-DOD), AM1/d (same as AM1D), PM6, and DFTB. The dispersion correction can be switched on for AM1 and PM6 by choosing AM1-D* and PM6-D, respectively. The dispersion and hydrogen bond correction will be applied for AM1-DH+ and PM6-DH+.

6. *sqm*: Semi-empirical quantum chemistry

- dftb_disper** Flag turning on (1) or off (0) the use of a dispersion correction to the DFTB/SCC-DFTB energy. Requires *qm_theory*=DFTB. It is assumed that you have the file DISPERSION.INP_ONCHSP in your \$AMBERHOME/dat/slko/ directory. This file must be downloaded from the website www.dftb.org, as described in the beginning of this chapter. Not available for the Zn atom. (Default = 0)
- dftb_3rd_order** Third order correctio to SCC-DFTB. Default=" (no third order correction).
- = **'PA'** Use the SCC-DFTB-PA parametrization, which was developed for proton affinities. The parameters will be read from the \$AMBERHOME/dat/slko/DFTB_3RD_ORDER_PA.DAT file.
 - = **'PR'** Use the SCC-DFTB-PR parametrization, which was developed for phosphate hydrolysis reactions. The parameters will be read from the \$AMBERHOME/dat/slko/DFTB_3RD_ORDER_PR.DAT file.
 - = **'READ'** Parameters will be read from the *mdin* file, in a separate "dftb_3rd_order" namelist, which must have the same format as the files above.
 - = **'filename'** Parameters will be read from the file specified by *filename*, in the "dftb_3rd_order" namelist, which must have the same format as the files above.
- dftb_chg** Flag to choose the type of charges to report when doing a DFTB calculation.
- = **0** (default) - Print Mulliken charges
 - = **2** Print CM3 charges. Only available for H, C, N, O, S and P.
- dftb_telec** Electronic temperature, in K, used to accelerate SCC convergence in DFTB calculations. The electronic temperature affects the Fermi distribution promoting some HOMO/LUMO mixing, which can accelerate the convergence in difficult cases. In most cases, a low *telec* (around 100K) is enough. Should be used only when necessary, and the results checked carefully. Default: 0.0K
- dftb_maxiter** Maximum number of SCC iterations before resetting Broyden in DFTB calculations. (default: 70)
- qmcharge** Charge on the QM system in electron units (must be an integer). (Default = 0)
- spin** Multiplicity of the QM system. Currently only singlet calculations are possible and so the default value of 1 is the only available option. Note that this option is ignored by DFTB/SCC-DFTB, which allows only ground state calculations. In this case, the spin state will be calculated from the number of electrons and orbital occupancy.
- qmqmdx** Flag for whether to calculate QM-QM derivatives analytically or pseudo numerically. The default (and recommended) option is to use ANALYTICAL QM-QM derivatives.
- = **1** (default) - Use analytical derivatives for QM-QM forces.

- = 2** Use numerical derivatives for QM-QM forces. Note: the numerical derivative code has not been optimised as aggressively as the analytical code and as such is significantly slower. Numerical derivatives are intended mainly for testing purposes.
- verbosity** Controls the verbosity of QM/MM related output. *Warning:* Values of 2 or higher will produce a *lot* of output.
- = 0** (default) - only minimal information is printed - Initial QM geometry and link atom positions as well as the SCF energy at every ntpf steps.
- = 1** Print SCF energy at every step to many more significant figures than usual. Also print the number of SCF cycles needed on each step.
- = 2** As 1 but also print info about memory reallocations, number of pairs per QM atom. Also prints QM core - QM core energy, QM core - MM charge energy and total energy.
- = 3** As 2 but also print SCF convergence information at every step.
- = 4** As 3 but also print forces onof the file QM atoms due to the SCF calculation and the coordinates of the link atoms at every step.
- = 5** As 4 but also print all of the info in kJ/mol as well as kcal/mol.
- tight_p_conv** Controls the tightness of the convergence criteria on the density matrix in the SCF.
- =0** (default) - loose convergence on the density matrix (or Mulliken charges, in case of a SCC-DFTB calculation). SCF will converge if the energy is converged to within scfconv and the largest change in the density matrix is within $0.05*\text{sqrt}(\text{scfconv})$.
- = 1** Tight convergence on density(or Mulliken charges, in case of a SCC-DFTB calculation). Use same convergence (scfconv) for both energy and density (charges) in SCF. Note: in the SCC-DFTB case, this option can lead to instabilities.
- scfconv** Controls the convergence criteria for the SCF calculation, in kcal/mol. In order to conserve energy in a dynamics simulation with no thermostat it is often necessary to use a convergence criterion of 1.0d-9 or tighter. Note, the tighter the convergence the longer the calculation will take. Values tighter than 1.0d-11 are not recommended as these can lead to oscillations in the SCF, due to limitations in machine precision, that can lead to convergence failures. Default is 1.0d-8 kcal/mol. Minimum usable value is 1.0d-14.
- pseudo_diag** Controls the use of 'fast' pseudo diagonalisations in the SCF routine. By default the code will attempt to do pseudo diagonalisations whenever possible. However, if you experience convergence problems then turning this option off may help. Not available for DFTB/SCC-DFTB.
- = 0** Always do full diagonalisation.

6. sqm: Semi-empirical quantum chemistry

= 1 Do pseudo diagonalisations when possible (default).

pseudo_diag_criteria Float controlling criteria used to determine if a pseudo diagonalisation can be done. If the difference in the largest density matrix element between two SCF iterations is less than this criteria then a pseudo diagonalisation can be done. This is really a tuning parameter designed for expert use only. Most users should have no cause to adjust this parameter. (Not applicable to DFTB/SCC-DFTB calculations.) Default = 0.05

diag_routine Controls which diagonalization routine should be used during the SCF procedure. This is an advanced option which has no effect on the results but can be used to fine tune performance. The speed of each diagonalizer is both a function of the number and type of QM atoms as well as the LAPACK library that Sander was linked to. As such there is not always an obvious choice to obtain the best performance. The simplest option is to set `diag_routine = 0` in which case Sander will test each diagonalizer in turn, including the pseudo diagonalizer, and select the one that gives optimum performance. This should ideally be the default behavior but this option has not been tested on sufficient architectures to be certain that it will always work. Not available for DFTB/SCC-DFTB.

= 0 Automatically select the fastest routine (recommended).

= 1 Use internal diagonalization routine (default).

= 2 Use lapack dspev.

= 3 Use lapack dspevd.

= 4 Use lapack dspevx.

= 5 Use lapack dsyev.

= 6 Use lapack dsyevd.

= 7 Use lapack dsyevr.

printcharges

= 0 Don't print any info about QM atom charges to the output file (default)

= 1 Print Mulliken QM atom charges to output file every *npr* steps.

qxd Flag to turn on (=true.) or off (=false., default) the charge-dependent exchange-dispersion corrections of vdW interactions[95].

parameter_file

= 'PARAM.FILE' Read user-defined parameters from the file 'PARAM.FILE'. The first three space-separated entries (case insensitive) of each line will be interpreted as a user-modified parameter in the sequence of *parameter name*, *element name*, and *value*. For example, a line contains "USS Cl -111.6139480D0" will cause the USS parameter of the Cl element changed to -111.6139480. A line beginning with "END" will stop the reading. This function currently only works for MNDO, AM1, PM3, MNDO/d, and AM1/d.

Also, when new nuclear core-core parameters (FN, in PM3, AM1, and AM1/d) are re-defined, the number of FNN parameter sets (NUM_FN) also needs to be defined. For example, if FN n 3 ($n = 1, 2$, or 3) is defined, then NUM_FN needs to be set to 3 or 4 .

peptide_corr

- = 0 Don't apply MM correction to peptide linkages. (default)
- = 1 Apply a MM correction to peptide linkages. This correction is of the form $E_{scf} = E_{scf} + h_{type}(i_{type}) \sin^2 \phi$, where ϕ is the dihedral angle of the H-N-C-O linkage and h_{type} is a constant dependent on the Hamiltonian used. (Recommended, except for DFTB/SCC-DFTB.)

itrmax Integer specifying the maximum number of SCF iterations to perform before assuming that convergence has failed. Default is 1000. Typically higher values will not do much good since if the SCF hasn't converged after 1000 steps it is unlikely to. If the convergence criteria have not been met after itrmax steps the SCF will stop and the minimisation will proceed with the gradient at itrmax. Hence if you have a system which does not converge well you can set itrmax smaller so less time is wasted before assuming the system won't converge. In this way you may be able to get out of a bad geometry quite quickly. Once in a better geometry SCF convergence should improve.

maxcyc Maximum number of minimization cycles to allow, using the *xmin* minimizer (see Section 18.4) with the TNCG method. Default is 9999. Single point calculations can be done with *maxcyc* = 0.

ntpr Print the progress of the minimization every *ntpr* steps; default is 10.

grms_tol Terminate minimization when the gradient falls below this value; default is 0.02

ndiis_attempts Controls the number of iterations that DIIS (direct inversion of the iterative subspace) extrapolations will be attempted. Not available for DFTB/SCC-DFTB. The SCF does not even begin to exhaust its attempts at using DIIS extrapolations until the end of iteration 100. Therefore, for example, if *ndiis_attempts*=50, then DIIS extrapolations would be performed at end of iterations 100 to 150. The purpose of not performing DIIS extrapolations before iteration 100 is because the existing code base performs quite well for most molecules; however, if convergence is not met after 100 iterations, then it is presumed that further iterations will not yield SCF convergence without doing something different, i.e., DIIS. Thus, the implementation of DIIS in SQM is a mechanism to try and force SCF convergence for molecules that are otherwise difficult to converge. Default 0. Maximum 1000. Minimum 0. Note that DIIS will automatically turn itself on for 100 attempts at the end of iteration 800 even if you did not explicitly set *ndiis_attempts* to a nonzero value. This is done as a final effort to achieve convergence.

ndiis_matrices Controls the number of matrices used in the DIIS extrapolation. Including only one matrix is the same as not performing an extrapolation. Including an excessive

6. *sqm*: Semi-empirical quantum chemistry

number of matrices may require a large amount of memory. Not available for DFTB/SCC-DFTB. Default 6. Minimum 1. Maximum 20.

errconv SCF tolerance on the maximum absolute value of the error matrix, i.e., the commutator of the Fock matrix with the density matrix. The value has units of hartree. The default value of **errconv** is sufficiently large to effectively remove this tolerance from the SCF convergence criteria. Not available for DFTB/SCC-DFTB. Default 1.d-1. Minimum 1.d-16. Maximum 1.d0.

qmmm_int When running QM calculations in the *sqm* program, an electric field of external point charges can be added. In this way, the electrostatic effect outside of the QM region can be modeled, making the calculation a simplified QM/MM calculation without QM/MM vdW's contribution. Like QM/MM calculations (see AMBER 12 manual), the method to couple QM and MM electrostatic interactions for external charges and semiempirical Hamiltonians can be specified via the *qmmm_int* namelist variable.

The current implementation limits use of external charges to only single point energy calculations. To run such a calculation, an additional field, which begins with **#EXCHARGES** and ends with **#END**, is required to specify the external point charges in the input. Each external point charge must include atomic number, atom name, X, Y, Z coordinates and the charge in units of the electron charge. An example input looks like:

```
single point energy calculation (adenine), with external charges (thymine)
&qmmm
  qm_theory = 'PM3',
  qmcharge = 0,
  maxcyc = 0,
  qmmm_int = 1,
/
7  N   1.0716177  -0.0765366   1.9391390
1  H   0.0586915  -0.0423765   2.0039181
1  H   1.6443796  -0.0347395   2.7619159
6  C   1.6739638  -0.0357766   0.7424316
7  N   0.9350155  -0.0279801  -0.3788916
6  C   1.5490760   0.0012569  -1.5808009
1  H   0.8794435   0.0050260  -2.4315709
7  N   2.8531510   0.0258031  -1.8409596
6  C   3.5646109   0.0195446  -0.7059872
6  C   3.0747955  -0.0094480   0.5994562
7  N   4.0885824  -0.0054429   1.5289786
6  C   5.1829921   0.0253971   0.7872176
1  H   6.1882591   0.0375542   1.1738824
7  N   4.9294871   0.0412404  -0.5567274
1  H   5.6035368   0.0648755  -1.3036811
```



```

#EXCHARGES
6  C  -4.7106131   0.0413373   2.1738637  -0.03140
1  H  -4.4267056   0.9186178   2.7530256   0.06002
1  H  -4.4439282  -0.8302573   2.7695655   0.05964
1  H  -5.7883971   0.0505530   2.0247280   0.03694
6  C  -3.9917387   0.0219348   0.8663338  -0.25383
6  C  -4.6136833   0.0169051  -0.3336520   0.03789
1  H  -5.6909220   0.0269347  -0.4227183   0.16330
7  N  -3.9211729  -0.0009646  -1.5163659  -0.47122
1  H  -4.4017172  -0.0036078  -2.4004924   0.35466
6  C  -2.5395897  -0.0149474  -1.5962357   0.80253
8  O  -1.9416783  -0.0291878  -2.6573783  -0.63850
7  N  -1.9256484  -0.0110593  -0.3638948  -0.58423
1  H  -0.8838255  -0.0216168  -0.3784269   0.35404
6  C  -2.5361367   0.0074651   0.8766724   0.71625
8  O  -1.8674730   0.0112093   1.9120833  -0.60609
#END

```


7. cpptraj

For many years, *ptraj* has been the workhorse for trajectory analysis in Amber. It is able to perform many types of analyses, and can process multiple trajectories. However, one of its limitations is that all coordinates in a given *ptraj* run must correspond to a single topology file. This prevents certain types of analysis, for example calculating the RMSD of a coordinate frame to a reference frame with a different topology.

Cpptraj is a complimentary program to *ptraj* that can process trajectory files with different topology files in the same run. Although certain parts of the *ptraj* code are used in *cpptraj*, it is overall a completely new code base written primarily in C++ with an eye towards making future code development and additions as easy as possible. In addition to reading multiple topology files, *cpptraj* can read multiple reference structures, write multiple output files (for which specific frames to be written can be specified), stripped topology files (currently useable for visualization only), output multiple data sets to the same data file (e.g. two dihedral calculations like phi and psi can be written to one file), and has native support for compressed files along with many other improvements. The code is at least as fast as *ptraj*, and in many cases is faster, particularly when processing NetCDF trajectories. In addition, several actions have been parallelized with OpenMP to take advantage of multi-core machines for even more speedup (see section 7.2.4). Currently all *ptraj* actions and analyses except for clustering are implemented in *cpptraj*.

Cpptraj has been developed with an eye towards making it backwards-compatible with *ptraj* input. In general, if a command in *ptraj* has been implemented in *cpptraj* it should produce similar results, although the output format may be different. A *NOTABLE EXCEPTION* is the **hbond** command, which is quite different (see command syntax for details).

For a description of *ptraj*, see Appendix A.

7.1. Comparison to ptraj - Important Differences

The overall flow of *cpptraj* is similar to *ptraj*. First the run is set up via commands read in from an input file; a limited interactive interface (STDIN) similar to *ptraj* is also supported. Trajectories are then read in one frame at a time. Actions are performed on the coordinates stored in the frame, after which any output coordinates are written. At the end of the run, any data sets generated are written.

Some of the most notable differences from *ptraj* are as follows:

1. *Cpptraj* has several actions not implemented in *ptraj*:
 - a) **nastruct**: basic nucleic acid structure analysis
 - b) **surf**: calculate LCPO surface area.
 - c) **molsurf**: calculate Connolly surface area (same as \$AMBERHOME/bin/molsurf).

7. *cpptraj*

- d) **jcoupling**: calculate J-coupling values from dihedral angles.
 - e) **rotdif**: calculate rotational diffusion tensor
2. Several typically time-consuming actions in *cpptraj* are OpenMP parallelized; see section [7.2.4](#) for more details.
 3. Any file read or written by *cpptraj* can be compressed (with the exception of Netcdf trajectories). So for example gzipped/bzipped topology files can be read, and data files can be written out as gzip/bzip2 files. Compression is detected automatically when reading, and is determined by the filename extension (.gz and .bz2 respectively) on writing.
 4. If two actions specify the same data file with the 'out' keyword, data from both actions will be written to that data file.
 5. Data files can be written in xmgrace format if the filename given has a '.agr' extension. Data files can also be written in a contour map style readable by gnuplot if the filename given has a '.gnu' extension.
 6. Multiple output trajectories can be specified. In addition, output files can be directed to write only specific frames from the input trajectories.
 7. Multiple reference structures can be specified. Specific frames from trajectories may be used as a reference structure.
 8. The rmsd action allows specification of a separate mask for the reference structure. In addition, per-residue RMSD can be calculated easily.
 9. When stripping coordinates with the strip action, a fully-functional stripped topology file can be written out.
 10. Data files declared in actions using the "out" keyword can have their format altered somewhat (for example, the precision of the numbers can be changed). In addition, new data files can be created from existing data sets.

7.1.1. Actions and multiple topologies

Since *cpptraj* supports multiple topology files, actions are set up every time the topology changes in order to recalculate things like what atoms are in a mask etc. Actions that are not valid for the current topology are skipped for that topology. So for example if the first topology file processed includes a ligand named MOL and the second one does not, the action:

```
distance :80 :MOL out D_80-to-MOL.dat
```

will be valid for the first topology but not for the second, so it will be skipped as long as the second topology is active.

7.1.2. Datasets and datafiles

Datafiles can currently be given in three formats: data file, grace file, and gnuplot contour. Data file simply has data in columns, like ptraj. Grace files can be read in by xmgrace. Gnuplot contour files consist of a series of gnuplot commands followed by the actual data; each set is printed to a row.

The format is specified by the file suffix, so that 'filename.agr' will output in grace format, 'filename.gnu' will output in gnuplot contour, and anything else is a normal data file. The xmgrace/gnuplot output is particularly nice for the secstruct sumout and rmsd perresout files.

7.1.3. Enhanced features for actions using the “out” keyword

With all action commands that print out a dataset (e.g. distance, angle, dihedral, rmsd, etc) an additional argument can be given optionally to specify the name of the dataset. For example, the command:

```
distance :1 :2 out d1.dat
```

will write the distance between residues 1 and 2 to d1.dat. The header of d1.dat will be something like

```
#Frame      dataXXXX
```

where XXXX is the internal number of the dataset. In contrast, the command:

```
distance d1 :1 :2 out d1.dat
```

will write the same distance to d1.dat, but the header of d1.dat will be:

```
#Frame      d1
```

Any action using the “out” keyword will allow datasets to be written into the same file. For example, the commands:

```
dihedral phi :1@C :2@N :2@CA :2@C out phipsi.dat
dihedral psi :2@N :2@CA :2@C :3@N out phipsi.dat
```

will produce in phipsi.dat:

```
#Frame      phi      psi
```

7.2. Running cpptraj

7.2.1. Command Line Syntax

The command line syntax is very similar to that of ptraj. To run Cpptraj:

```
cpptraj -i <input file> [-p <parm file1> ...]
```

7. cpptraj

where <input file> is a text file containing one or more keywords recognized by Cpptraj, and <parm file> is a topology file that is recognized by Cpptraj. It is not required that a topology file be specified on the command line as long as one is specified in <input file> with the 'parm' keyword. Multiple topology files can be specified by use of multiple '-p'.

For complete backwards compatibility with Ptraj the following syntax is also recognized:

```
cpptraj <parm file> <input file>
```

If run with no arguments or no specified input file:

```
cpptraj
cpptraj <parm file>
cpptraj -p <parm file>
```

this brings up a command-line style interface (STDIN), similar to that in Ptraj.

The syntax of <input file> is similar to that of ptraj. Keywords specifying actions are given one per line. Lines beginning with '#' are ignored as comments. Lines can also be continued through use of the '\' character.

Other command line options:

- **-help, -help:** print usage info and exit.
- **-V, -version:** print version number and exit.
- **-defines:** print list of defines used in compilation

7.2.2. Mask Syntax

The mask syntax is similar to Ptraj. Note that the characters ':', '@', and '*' are reserved for masks and should not be used in output file or dataset names. All masks are case-sensitive. Either names or numbers can be used. Masks can contain ranges (denoted with '-') and comma separated lists. The logical operands '&' (and), '|' (or), and '!' (not) are also supported.

The syntax for elementary selections is the following:

```
{:residue numlist} e.g. [:1-10] [:1,3,5] [:1-3,5,7-9]
{:residue namelist} e.g. [:LYS] [:ARG,ALA,GLY]
@{atom numlist} e.g. [@12,17] [@54-85] [@12,54-85,90]
@{atom namelist} e.g. [@CA] [@CA,C,O,N,H]
```

Several wildcard characters are supported:

```
'*' -- zero or more characters.
'?' -- one character.
'=' -- same as '*'
```

The wildcards can also be used with numbers or other mask characters, e.g. ':?0' means ":10,20,30,40,50,60,70,80,90", ':*' means all residues and '@*' means all atoms.

Compound expressions of the following type are allowed:

```
{:residue numlist | namelist}@{atom namelist | numlist}
```

and are processed as:

```
{residue numlist | namelist} & @ {atom namelist | numlist}
```

e.g. `:1-10@CA` is equivalent to `“:1-10 & @CA”`.

More examples:

:ALA,TRP All alanine and tryptophan residues.

:5,10@CA CA carbon in residues 5 and 10.

:*!@H= All non-hydrogen atoms (equivalent to `"!@H=`).

@CA,C,O,N,H All backbone atoms.

!@CA,C,O,N,H All non-backbone atoms (=sidechains for proteins only).

:1-500@O&!(:WAT|:LYS,ARG) All backbone oxygens in residues 1-500 but not in water, lysine or arginine residues.

Distance-based Masks

The syntax for selection by distance is `'<'`, `'>'`, (residue based), and `'<@'`, `'>@'`, (atom based). The `'<'` character means within, and `'>'` means without, e.g.

```
[ :11-17 <@ 2.4 ]
```

means all atoms within 2.4 Å distance to `:11-17`. Selection by distance for everything but the **mask** action requires defining a reference frame with **reference**. For example, to strip all residues farther than 3.0 Å (i.e. not within 3.0 Å) from residue 4 using reference coordinates:

```
reference mol.rst7
trajin mol.crd
strip !(:4<:3.0)
```

Distance-based masks that update each frame are currently only supported by the **mask** action.

7.2.3. Ranges

For several commands (trajout, nastruct action, and the per-residue functionality of the rmsd action) some arguments are ranges - THESE ARE NOT MASKS. They are simple number ranges using `'-'` to specify a range and `','` to separate different ranges. For example `1-2,4-6,9` specifies 1 to 2, 4 to 6, and 9.

7.2.4. OpenMP Parallelization

Some of the more time-consuming actions in Cpptraj have been parallelized with OpenMP to take advantage of machines with multiple cores. In order to use OpenMP parallelization Amber should be configured with the `'-openmp'` flag. The following actions have been OpenMP parallelized:

7. *cpptraj*

- closest
- mask (distance-based masks only)
- radial
- rmsavgcorr
- secstruct
- surf

By default Cpptraj will use all available cores. Note that if the `OMP_NUM_THREADS` environment variable is set it will force Cpptraj to use however many cores are specified by the variable.

7.3. Ptraj actions/analysis in Cpptraj

Several actions and analyses have been implemented in Cpptraj directly from ptraj code and function exactly as they do in ptraj; however, they have not yet been completely integrated into the overall framework of cpptraj, which means for example datafile commands cannot operate on data sets generated by these commands etc. The following is a list of ptraj commands recognized by cpptraj; see the corresponding entry in the ptraj section of the manual ([A](#)) for more information:

- atomicfluct
- contacts
- clusterdihedral
- diffusion
- dipole
- grid
- matrix
- principal
- randomizeions
- unwrap
- vector
- watershell
- analyze timecorr
- analyze matrix

7.4. General Commands

Commands in `cpptraj` can be read in from an input file or from STDIN. A '#' anywhere on a line denotes a comment; anything after '#' will be ignored no matter where it occurs. A '\' allows the continuation of one line to another. For example, the input:

```
# Sample input
trajin mdcrd # This is a trajectory
rms first out rmsd.dat \
    :1-10
```

Translates to:

```
[trajin mdcrd]
[rms first out rmsd.dat :1-10]
```

Most commands have a corresponding test which also serves as an example of how to use the command. See `$AMBERHOME/AmberTools/test/cpptraj/README` for more details.

7.4.1. activeref

```
activeref <#>
```

Set which reference structure should be used when setting up distance-based masks for everything but the 'mask' action. Numbering starts from 0, so 'activeref 0' selects the first reference structure read in, 'activeref 1' selects the second, and so on. This should only be specified once.

7.4.2. debug

```
debug <#>
```

Set the level of debug information to print. In general the higher the <#> the more information that is printed. There are also several related commands which can be used to enable debugging information for a specific area of `cpptraj`:

- **actiondebug <#>**: Set debug level for actions.
- **analysisdebug <#>**: Set debug level for analyses.
- **trajindebug <#>**: Set debug level for input trajectories.
- **trajoutdebug <#>**: Set debug level for output trajectories.
- **referencedebug <#>**: Set debug level for reference coordinates/trajectories.
- **parmdebug <#>**: Set debug level for parameter files.
- **datafiledebug <#>**: Set debug level for data files.

7. *cpptraj*

7.4.3. **noexitonerror**

noexitonerror

Normally *cpptraj* will exit if actions fail to initialize properly. If **noexitonerror** is specified, *cpptraj* will attempt to continue past such errors.

7.4.4. **noprogress**

noprogress

Do not display read progress during trajectory processing.

7.4.5. **select**

select <mask>

Prints atom numbers of atoms in <mask> to a line with format:

Selected= <#atom1> <#atom2> ...

This does not affect the state in any way, but is intended for use in scripts etc.

7.5. Parameter and Trajectory File Commands

These commands control the reading and writing of parameter and trajectory files. In *Cpptraj*, trajectories are always associated with a parameter file. If a parameter file is not specified, a trajectory file will be associated with the first parameter file loaded by default. There are three trajectory types in *Cpptraj*: input, output, and reference.

Parameter/Reference Tagging

Note that parameter and reference files may be 'tagged' (i.e. given a nickname); these tags can then be used in place of the file name itself. A tag in *cpptraj* is recognized by being bounded by brackets ('[' and ']'). This can be particularly useful when reading in many parameter or reference files. For example, when reading in multiple reference structures:

```
trajin Test1.crd
reference 1LE1.NoWater.Xray.rst7 [xray]
reference Test1.crd lastframe [last]
reference Test2.crd 225 [open]
rms Xray ref [xray] :2-12@CA out rmsd.dat
rms Last ref [last] :2-12@CA out rmsd.dat
rms Open ref [open] :2-12@CA out rmsd.dat
```

This defines three reference structures and gives them tags [xray], [last], and [open]. These reference structures can then be referred to by their tags instead of their filenames by any action that uses reference structures (in this case the RMSD action).

Similarly, this can be useful when reading in multiple parameter files:

```

parm tz2.ff99sb.tip3p.truncoc.t.parm7 [tz2-water]
parm tz2.ff99sb.mbondi2.parm7 [tz2-nowater]
trajin tz2.run1.explicit.nc parm [tz2-water]
reference tz2.dry.rst7 parm [tz2-nowater] [tz2]
rms ref [tz2] !(:WAT) out rmsd.dat

```

This defines two parm files and gives them tags [tz2-water] and [tz2-nowater], then reads in a trajectory associated with one, and a reference structure associated with the other. Note that in the 'reference' command there are two tags; the first goes along with the 'parm' keyword and specifies what parameter file the reference should use, the second is the tag given to the reference itself (as in the previous example) and is referred to in the subsequent RMSD action.

7.5.1. parm

```
parm <filename> ([tag])
```

Read in parameter file specified by <filename>. Currently can read Amber topology, PDB, TRIPOS MOL2, and Charmm PSF files. An optional tag can be given (bounded in brackets) which can be referred to in place of the parameter file name in order to simplify references to the parameter file (see beginning of the File Commands section for examples of how to use tags).

IMPORTANT NOTES FOR PDB FILES

In some PDB files, certain atoms contain the (*) character in their name (e.g. C1* in a nucleic acid backbone). Since in Cpptraj (*) is a reserved character for atom masks all (*) in PDB atom names are replaced with (') to avoid issues with the mask parser. So in a structure with an atom named C1*, to select it use the mask "@C1'".

Sometimes PDB files can contain alternate coordinates for the same atom in a residue, e.g.:

ATOM	806	CA	ACYS	A	105	6.460	-34.012	-21.801	0.49	32.23
ATOM	807	CB	ACYS	A	105	6.054	-33.502	-20.415	0.49	35.28
ATOM	808	CA	BCYS	A	105	6.468	-34.015	-21.815	0.51	32.42
ATOM	809	CB	BCYS	A	105	6.025	-33.499	-20.452	0.51	35.38

If this is the case Cpptraj will print a warning about duplicate atom names but will take no other action. Both residues are considered 'CYS' and the mask ':CYS@CA' would select both atom 806 and 809.

7.5.2. parminfo

```
parminfo [<parmindex>] [<mask>]
```

Print out parm information for atoms in <mask> for the parm specified by <parmindex> (parmindex 0 i.e. the first parm if not specified). If no mask is given, general information about the parameter file is printed.

7. cpptraj

7.5.3. parmwrite

```
parmwrite out <filename> [<parmindex>]
```

Write out parm specified by <parmindex> (0, first parm loaded by default) to <filename> in Amber topology format.

7.5.4. parmstrip

```
parmstrip <mask> [<parmindex>]
```

Strip atoms in <mask> from parm specified by <parmindex> (0, first parm loaded by default). Note that unlike the strip action, this permanently modifies the parm for all subsequent commands. This command can be used to e.g. quickly created stripped Amber topology files:

```
parm mol.water.parm7
parmstrip :WAT
parmwrite out strip.mol.parm7
```

7.5.5. box | parmbox

```
[parm]box [<parmindex>] [x <xval>] [y <yval>] [z <zval>]
        [alpha <a>] [beta <b>] [gamma <g>] [nobox]
```

Modify the box information for parm <parmindex> (0, first parm loaded by default). Overwrites box information if present with lengths specified by **x**, **y**, and **z**, and angles specified by **alpha**, **beta**, and **gamma**. Any or all may be specified, any that are not specified will remain unchanged. If nobox is specified any existing box information will be removed. This can be useful for e.g. removing box information from a parm when stripping solvent:

```
parm mol.water.parm7
parmstrip :WAT
parmbox nobox
parmwrite out strip.mol.nobox.parm7
```

7.5.6. parmbondinfo

```
parmbondinfo [<parmindex>]
```

Print bond information for parm <parmindex> (0, first parm loaded by default) with format:

```
Atom <atom1> to <atom2>, <index>
```

where <atom1> and <atom2> are the atoms involved in the bond, and <index> is an internal pointer to bond parameters (-1 if no bond parameters present).

7.5.7. parmmolinfo

```
parmmolinfo [<parmindex>]
```

Print molecule information for parm <parmindex> (0, first parm loaded by default) with format:

```
Molecule <mol>, <natom> atoms, first residue <resname>
```

where <mol> is the molecule number, <natom> is the number of atoms in the molecule, and <resname> is the residue name of the first residue in the molecule.

7.5.8. parmresinfo

```
parmresinfo [<parmindex>]
```

Print residue information for parm <parmindex> (0, first parm loaded by default) with format:

```
Residue <res>, <resname>, first atom <atom>
```

where <res> is the residue number, <resname> is the residue name, and <atom> is the number of the first atom in the residue.

7.5.9. bondsearch | nobondsearch

If **bondsearch** is specified, bond information will be determined for parameter files without bond information (e.g. PDB files) based on a distance search upon loading. If **nobondsearch** is specified, bond information will not be automatically determined (default). Note that this does not affect parameter files loaded from the command line.

7.5.10. molsearch | nomolsearch

If **molsearch** is specified, molecule information will be determined for parameter files without molecule information based on bond information upon loading. If bond information is not present, it will be determined automatically (similar to specifying **bondsearch**). If **nomolsearch** is specified, molecule information will not be automatically determined (default). Note that this does not affect parameter files loaded from the command line.

7.5.11. trajin

```
trajin <filename> [<start> [[<stop> | last] [<offset>]]] | lastframe  
[parm <parmfile> | parmindex <#>]  
[remdtraj remdtrajtemp <Temperature> remdout <outprefix> [netcdf]  
trajnames <Comma-separated name list>]
```

7. *cpptraj*

Read in trajectory specified by filename. Currently Cpptraj will recognize Amber trajectories, Amber Netcdf Trajectories, Charmm DCD trajectories, Amber Restart files, PDB files, and TRIPOS MOL2 files. The frames to be read in can be specified with the <start>, <stop>, and <offset> arguments (defaults are first frame, last frame, and 1 respectively). If just the <start> argument is given, all frames from <start> to the last frame of the trajectory will be read. To read in a trajectory with offsets where the last frame # is not known, specify the **last** keyword instead of a <stop> argument, e.g.

```
trajin Test1.crd 10 last 2
```

This will process Test1.crd from frame 10 to the last frame, skipping by 2 frames. To explicitly select only the last frame, specify the **lastframe** keyword:

```
trajin Test1.crd lastframe
```

The trajectory is associated with a parameter file specified by **parm** <parmfile>, where <parmfile> is the parameter filename or associated [tag] name. Parameter files can also be specified by **parmindex** <#>, where <#> is the order in which the parm was read in starting from 0. If no **parm** or **parmindex** keyword is given the first parm read in is used. Example:

```
parm top0.parm7
parm top1.parm7
parm top2.parm7 [top2]
parm top3.parm7
trajin Test0.crd
trajin Test1.crd parm top1.parm7
trajin Test2.crd parm [top2]
trajin Test3.crd parmindex 3
```

Test0.crd is associated with top0.parm7; since no parm was specified it defaulted to the first parm read in. Test1.crd was associated with top1.parm7 by filename, Test2.crd was associated with top2.parm7 by its tag, and finally Test3.crd was associated with top3.parm7 by its index (based on the order it was read in).

Replica Trajectory Processing

If the **remdtraj** keyword is specified the trajectory is treated as belonging to the lowest # replica of a group of REMD trajectories. The remaining replicas can be either automatically detected by following a naming convention of <REMDFILENAME>.X, where X is the replica number, or explicitly specified in a comma-separated list following the **trajnames** keyword. All trajectories will be processed at the same time, but only frames with a temperature matching the one specified by **remdtrajtemp** will be processed. For example, to process replica trajectories rem.001, rem.002, rem.003, and rem.004, grabbing only the frames at temperature 300.0 (assuming that this is a temperature in the ensemble):

```
trajin rem.001 remdtraj remdtrajtemp 300
```

or

```
trajin rem.001 remdtraj remdtrajtemp 300 trajnames rem.002,rem.003,rem.004
```

Replica trajectories can be easily converted to temperature trajectories by using the **remdout** keyword. Trajectory files will be created for each temperature <T> in the ensemble with format <outprefix>.<T>. The default output format is Amber formatted (ASCII text) trajectory; to write out netcdf temperature trajectories specify the **netcdf** keyword. For example, to convert replica trajectories rem.001, rem.002, rem.003, and rem.004 to temperature trajectories with prefix “temperature.” in netcdf format:

```
trajin rem.001 remdtraj remdtrajtemp 300 remdout temperature netcdf
```

7.5.12. trajout

```
trajout <filename> [<fileformat>] [append] [nobox]
      [parm <parmfile> | parmindex <#>] [onlyframes <range>] [title <title>]
      [ pdb | netcdf | restart | ncrestart | mol2 | dcd ]
Options for pdb format: [model | multi] [dumpq] [chainid <ID>]
Options for Amber format: [remdtraj] [highprecision]
Options for Netcdf format: [remdtraj]
Options for Restart/Netcdf Restart format: [remdtraj] [novelocity]
      [time0 <initial time>] [dt <timestep>]
Options for mol2 format: [single | multi]
```

Write trajectory specified by filename in specified file format (Amber trajectory if none specified). Other currently recognized formats are **pdb** (PDB file), **netcdf** (Amber Netcdf trajectory), **restart** (Amber restart), **ncrestart** (Amber netcdf restart), **mol2** (TRIPOS MOL2 format), or **dcd** (Charmm DCD format). Note that now the file type can be determined from the output extension if not specified by a keyword: recognized extensions are “.nc”, “.ncrst”, “.pdb”, “.mol2”, “.dcd”, “.rst7”, and “.crd”. Coordinate output occurs after all actions are completed; for coordinate output during the action list see the **outtraj** command.

The file will be appended to if **append** is specified (only supported Amber trajectory and Amber netcdf trajectory). Box coordinates will not be written if **nobox** specified (only matters when input topology has box coordinates). Associate with parameter file specified by **parm** <parmfile> or **parmindex** <#> (first parm read in if not specified). The output trajectory title can be set using the **title** keyword.

Multiple output trajectories of any format can be specified. Currently frames will be written to the output trajectory when the parameter file being processed matches the parameter file the output trajectory was set up with. So given the input:

```
parm top0.parm7
parm top1.parm7 [top1]
trajin input0.crd
trajin input1.crd parm [top1]
trajout output.crd parm [top1]
```

7. cpptraj

only frames read in from input1.crd (which is associated with top1.parm7) will be written to output.crd. The trajectory input0.crd is associated with top0.parm7; since no output trajectory is associated with top0.parm7 no frames will be written when processing top0.parm7/input0.crd.

If **onlyframes** <range> is given, only input frames matching the specified range will be written out. For example, given the input:

```
trajin input.crd 1 10
trajout output.crd onlyframes 2,5-7
```

only frames 2, 5, 6, and 7 from input.crd will be written to output.crd.

Options for “pdb”

The **model** and **multi** keywords control how the PDB file is written. If **model** is specified each frame will be written to filename, separated by MODEL/ENDMDL records (this is the default behavior). If **multi** is specified, each frame will be written to a separate filename with format *filename.frame#* (the **multi** mode matches the behavior of ptraj).

If **dumpq** is specified the PDB will be written in PQR format, with charges and GB radii written to the occupancy and B-factor columns respectively.

By default PDB files are written without a chain ID. To explicitly include a chain ID, specify '**chainid** <ID>', where ID is a single character.

Options for Amber Trajectory (default)

The **remdtraj** keyword will cause the REMD header to be printed with the current temperature (0.0 if no temperature has been read in). In Sander the REMD header has format:

```
REMD      <Replica>    <Step>    <Total_Steps>  <Temperature>
```

However, since Cpptraj has no concept of replica number, 0 is printed for <Replica>. <Step> and <Total_Steps> are set to the current frame #.

The **highprecision** keyword is intended for expert use only. Cpptraj will attempt to write Amber trajectory coordinates with 6 decimal places of precision instead of the normal 3. Note that however the coordinate width will remain at 8 chars, so in some cases 6 decimal places of precision will not be possible, meaning the precision of different coordinates may not match. A better alternative for higher precision is to use Netcdf trajectories.

Options for “netcdf”

The **remdtraj** keyword will save the current temperature (0.0 if no temperature has been read in).

Options for “restart” / “ncrestart”

The **remdtraj** keyword will write the current temperature to the restart file.

The **novelocity** keyword will prevent velocity information from being written to the restart file.

7.6. Datafile Commands (actions using the “out” keyword only)

The **time0** <initial time> argument will set the initial time for the restart file, and the **dt** <timestep> argument will scale the time by the given factor, so that the restart output <Time> field will be “(<initial time> + currentSet) * <timestep>”.

Options for “mol2”

The **single** and **multi** keywords control how the Mol2 file is written. If **single** is specified, all frames will be written to a single Mol2 file separated by @<TRIPOS>MOLECULE fields (this is the default behavior). If **multi** is specified, each frame will be written to a separate file named *filename.frame#*.

7.5.13. reference

```
reference <filename> [<frame#>]|lastframe [parm <parmfile> | parmindex <#>]
[average <stop> <offset>] ([tag])
```

Read specified trajectory frame (1 if not specified) as reference coordinates. Associate with parmfile or parmindex (first parm read in if not specified). The **lastframe** keyword can be used instead of a frame number to explicitly select the last frame of the trajectory. If the **average** keyword is specified and <filename> contains more than 1 frame, the average structure of <filename> will be stored as reference coordinates. Note that no RMS fitting is performed during the averaging. When specifying the **average** keyword, <stop> and <offset> arguments can be given to control which frames are averaged (similar to trajin). For example:

```
reference mdcrd.crd average 3 21 2
```

will calculate the average structure of mdcrd.crd from frames 2 to 20 with an offset of 2 and use as a reference structure.

An optional tag can be given (bounded in brackets) which can be referred to in place of the reference file name in order to simplify references to the reference file (see beginning of the File Commands section for examples of how to use tags).

7.6. Datafile Commands (actions using the “out” keyword only)

There is a subsection of commands that can be used to either modify datafiles which have been declared with an 'out' keyword or to create new datafiles from declared datasets.

7.6.1. datafile create

```
datafile create <filename> <datasetname0> [<datasetname1> ...]
```

Create a new datafile from one or more existing data sets. In general, actions which allow one to specify <dataset name> can be used to create a datafile.

7. *cpptraj*

7.6.2. datafile xlabel

```
datafile xlabel <filename> <label>
```

Set the x-axis label for the specified datafile to <label>. For regular data files this is the header for the first column of data.

7.6.3. datafile ylabel

```
datafile ylabel <filename> <label>
```

Set the y-axis label for the specified datafile to <label>.

7.6.4. datafile invert

```
datafile invert <filename>
```

Normally, data is written out with X-values pertaining to frames (i.e. data over all trajectories is printed in columns). This command flips that behavior so that X-values pertain to data sets (i.e. data over all trajectories is printed in rows). This command currently has no effect on gnuplot data files.

7.6.5. datafile noxcol

```
datafile noxcol <filename>
```

Prevent printing of indicies (i.e. the #Frame column in most datafiles) for the specified datafile. Useful e.g. if one would like a 2D plot such as phi vs psi. For example, given the input:

```
dihedral phi :1@C :2@N :2@CA :2@C out phipsi.dat
dihedral psi :2@N :2@CA :2@C :3@N out phipsi.dat
datafile noxcol phipsi.dat
```

Cpptraj will write a 2 column datafile containing only phi and psi, no frame numbers will be written.

7.6.6. datafile noheader

```
datafile noheader <filename>
```

Prevent printing of header line (e.g. '#Frame D1') at the beginning of data file specified by <filename>.

7.6.7. datafile precision

```
datafile precision <filename> [<datasetname>|*] [<width>] [<precision>]
```

Set the precision for the given dataset in the specified datafile to *width.precision*, where width is the column width and precision is the number of digits after the decimal point. If '*' or no dataset name is specified instead of a dataset name, the precision of all data sets in the datafile will be set. Note that the <precision> argument only applies to floating-point data sets. This command has no effect on String data sets.

For example, if one wanted to set the precision of the output of an Rmsd calculation to 8.3, the input would be:

```
trajin ../run0.nc
rms first :10-260 out prec.dat
datafile precision prec.dat 8 3
```

and the output would look like:

```
#Frame RMSD_00000
1 0.000
2 0.630
```

7.7. Commands that modify the state

These commands modify the current topology and/or coordinates for every action that follows them. For example, given a solvated system with water residues named WAT and the following commands:

```
rmsd first :WAT out water-rmsd.dat
strip :WAT
rmsd first :WAT out water-rmsd-2.dat
```

the first 'rms' command will be valid, but the second 'rms' command will not since all residues named WAT are removed from the state by the 'strip' command.

7.7.1. atommap

```
atommap <target> <reference> [mapout <filename>] [maponly]
[rmsfit [ rmsout <rmsout> ]]
```

Attempt to map the atoms of <target> to those of <reference> based on structural similarity. This is useful e.g. when there are two files containing the same structure but with different atom names or atom ordering. Both <target> and <reference> need to have been read in with a previous *reference* command. The state will then be modified so that any trajectory read in with the same parameter file as <target> will have its atoms mapped (i.e. reordered) to match those of <reference>. If the number of atoms that can be mapped in <target> are less than those in

7. *cpptraj*

<reference>, the reference structure specified by <reference> will be modified to include only mapped atoms; this is useful if for example the reference structure is protonated with respect to the target.

If the **maponly** keyword is specified, only the atom map will be printed; the state will not be modified.

If the **mapout** keyword is specified, the atom map will be written to <filename> with format:

```
TargetAtomNumber TargetAtomName ReferenceAtomNumber ReferenceAtomName
```

If a target atom cannot be mapped to a reference atom “—” will be printed in the map for that atom.

If the **rmsfit** keyword is specified, any input frames using the same parm as <target> will be RMS fit to <reference> using whatever atoms could be mapped. This is useful in cases where the atom mapping will not be complete (e.g. two ligands with the same scaffold but different substituents). The RMSD of the fit can be written to a file <rmsout> by using the **rmsout** keyword.

For example, say you have the same ligand structure in two files, Ref.mol2 and Lig.mol2, but the atom ordering in each file is different. To map the atoms in Lig.mol2 onto those of Ref.mol2 so that Lig.mol2 has the same ordering as Ref.mol2:

```
parm Lig.mol2
reference Lig.mol2
parm Ref.mol2
reference Ref.mol2 parminx 1
atommap Lig.mol2 Ref.mol2 mapout atommap.dat
trajin Lig.mol2
trajout Lig.reordered.mol2 mol2
```

7.7.2. center

```
center [<mask>] [origin] [mass]
```

Move all atoms so that the geometric center of the atoms in <mask> (all atoms if no mask specified) is at the center of the box (X/2, Y/2, Z/2). If **origin** is specified, center to the origin of the box (0, 0, 0) instead. If **mass** is specified, move using the center of mass of atoms instead of geometric center. This command is not valid for topology files with no box information.

For example, to move all coordinates so that the center of mass of residue 1 is at the center of the box:

```
center :1 mass
```

7.7.3. closest

```
closest <# to keep> <mask> [noimage] [first | oxygen] [closestout <filename>]
[outprefix <parmprefix>]
```

Similar to the **strip** command, but modify coordinate frame and topology by keeping only the specified number of closest solvent molecules to the region specified by the given mask. Distances are calculated between every atom in <mask> and either every atom in a solvent molecule, or only the first atom if the **first** or **oxygen** keyword is specified (it is recommended in most cases to use this keyword as the calculation will be sped up considerably). Imaging is turned on by default; the **noimage** keyword turns it off.

If **outprefix** is specified, for every topology modified the resulting stripped topology will be written in Amber format with filename <parmprefix>.parmname.

If **closestout** is specified information on the closest solvent molecules will be printed to <filename> with format:

Frame	Molecule	Distance	FirstAtom#
-------	----------	----------	------------

For example, to obtain the 10 closest waters to residues 1-268 by distance to the first atom of the waters, write out which waters were closest for each frame to a file called “closestmols.dat”, and write out the stripped topology with prefix “closest” containing only the solute and 10 waters:

```
closest 10 :1-268 first closestout closestmols.dat outprefix closest
```

7.7.4. image

```
image [origin] [center] [triclinic | familiar [com <commask>]] [<mask>]
```

For periodic systems only, image by molecule the atoms in <mask> (or all atoms if no mask specified) that are outside of the box back into the box. Currently both orthorhombic and non-orthorhombic boxes are supported. Right now only imaging by molecule is supported.

Imaging occurs with respect to the center of the box. If the **origin** keyword is specified imaging will occur with respect to the origin. By default a molecule will be imaged if its first atom is outside the box (similar to sander/pmemd). If **center** is specified a molecule will be imaged if its center of mass is outside the box.

If **triclinic** is specified force imaging with triclinic code; this is the default for non-orthorhombic boxes that are not truncated octahedrons. If **familiar** is specified, image with the triclinic code but put the box into the more familiar truncated octahedral shape; this is the default for truncated octahedral boxes. If **com** is specified with **familiar**, image with respect to the center of mass of atoms in <commask>.

A typical use of **image** is to move molecules back into the box after performing **center**. For example, the following commands move all atoms so that the center of residue 1 is at the center of the box, then image so that all molecules that are outside the box after centering are wrapped back inside:

```
center :1
image
```

7. *cpptraj*

7.7.5. runavg

```
runavg [window <window_size>]
```

Note that for backwards compatibility with ptraj “runningaverage” is also accepted.

Replaces the current frame with a running average over a number of frames specified by **window** <window_size> (5 if not specified). This means that in order to build up the correct number of frames to calculate the average, the first <window_size> minus one frames will not be processed by subsequent actions. So for example given the input:

```
runavg window 3
rms first out rmsd.dat
```

the rms command will not take effect until frame 3 since that is the first time 3 frames are available for averaging (1, 2, and 3). The next frame processed would be an average of frames 2, 3, and 4, etc.

7.7.6. strip

```
strip <mask> [outprefix <name>]
```

Strip all atoms specified by <mask> from the frame and modify the topology to match. If *outprefix* is specified, for every topology modified in this way a file <name>.<parmFilename> Amber topology file will be written that matches the stripped system. These topologies are fully-functional Amber topologies

For example, to strip all residues named WAT from the current topology:

```
strip :WAT
```

Note that stripping a system rennumbers all atoms and residues, so for example after this command:

```
strip :1
```

residue 1 will be gone, and the former second residue will now be the first, and so on.

7.7.7. unstrip

```
unstrip
```

Requests that the original topology and frame be used for all following actions. This has the effect of undoing any command that modifies the state (such as strip). For example, the following code takes a solvated complex and uses a combination of strip, unstrip, and outtraj commands to write out separate dry complex, receptor, and ligand files:

```
parm Complex.WAT.pdb
trajin Complex.WAT.pdb
# Remove water, write complex
strip :WAT
```

```

outtraj Complex.pdb pdb
# Reset to solvated Complex
unstrip
# Remove water and ligand, write receptor
strip :WAT,LIG
outtraj Receptor.pdb pdb
# Reset to solvated Complex
unstrip
# Remove water and receptor, write ligand
strip :WAT
strip !(:LIG)
outtraj Ligand.pdb pdb

```

7.7.8. unwrap

7.8. Action Commands

Most actions in Cpptraj function exactly the way they do in ptraj and are backwards-compatible. Some commands have extra functionality (such as the per-residue rmsd function of the rmsd action, or the ability to write out stripped topologies for visualization in the strip action), while other actions produce slightly different output (like the hbond/secstruct actions).

7.8.1. angle

```
angle [<dataset name>] <mask1> <mask2> <mask3> [out <filename>] [mass]
```

Calculate angle (in degrees) between atoms in <mask1>, <mask2>, and <mask3>. If *mass* is specified use the center of mass of atoms in the masks instead of geometric center.

7.8.2. atomicfluct

```
atomicfluct [out <filename>] [<mask>] [byres | byatom | bymask] [bfactor]
           [start <start>] [stop <stop>] [offset <offset>]
```

Compute the atomic positional fluctuations for all the atoms; output is performed only for the atoms specified in the <mask>. If **byatom** is specified, dump the calculated fluctuations by atom (default). If **byres** is specified, dump the average (mass-weighted) for each residue. If **bymask** is specified, dump the average (mass-weighted) over all the atoms in the original *mask*. If **out** is specified, the data will be dumped to *filename* (otherwise the values will be dumped to the standard output).

If the optional **start**, **stop**, and **offset** keywords are specified, only frames from <start> to <stop> (skipping <offset>) will be processed. This paring down of snapshots acts on top of, or in addition to, any offsets specified with “*trajin*”.

7. cpptraj

If the keyword **bfactor** is specified, the data is output as B-factors rather than atomic positional fluctuations (which simply means multiplying the *squared* fluctuations by $\frac{8}{3}\pi^2$). The data is dumped into two columns (n and value) where n goes from 1 to the number of atoms or groups specified and the value is the appropriate RMSF (Å) or B-factor ($\text{\AA}^2 \times \frac{8}{3}\pi$).

So, to dump the mass-weighted B-factors for the protein backbone atoms C, CA, and N, by residue use the command:

```
atomicfluct out back.apf @C,CA,N byres bfactor
```

To dump the RMSF or atomic positional fluctuations of the same atoms, use the command:

```
atomicfluct out backbone-atoms.apf @C,CA,N
```

Note that RMS fitting is not done implicitly. If you want fluctuations without rotations or translations (for example to the average structure), perform an RMS fit to the average structure (best) or the first structure (see *rmsd*) prior to this calculation.

7.8.3. average

```
average <filename> [<mask>] [start <start>] [stop <stop>] [offset <offset>]  
[Trajout Args]
```

Average the coordinates in <mask> (all atoms if none specified) over frames from <start> (default frame 1) to <stop> (default last frame) with an offset of <offset> (default 1). Write the averaged coordinates to a trajectory file specified by <filename>. If the number of atoms in <mask> are less than the total number of atoms, the topology will be stripped to match <mask> for output of this command only - the state will not be modified. Note that since coordinates are being averaged over many frames, resulting structures may appear distorted. For example, if one averages the coordinates of a freely rotating methyl group the average position of the hydrogen atoms will be close to the center of rotation.

Any arguments that are valid for the *trajout* command can be passed to this command in order to control the format of the output coordinates. For example, to write out a PDB file containing the averaged coordinates over all frames:

```
average test.pdb pdb
```

To write out a mol2 file containing only the averaged coordinates of residues 1 to 10 for frames 1 to 100:

```
average test.mol2 mol2 start 1 stop 100 :1-10
```

7.8.4. avgcoord

```
avgcoord [<mask>] [mass] outfile <file> [magnitude]
```

For each frame, calculate the average X, Y, and Z vectors over all coordinates in <mask>. If the **mass** keyword is specified the averages will be mass-weighted. Output is to the file specified by **outfile** with format:


```
<Frame> <X component> <Y component> <Z component>
```

If the **magnitude** keyword is specified an additional column containing the magnitude of the vector will be printed.

7.8.5. check

```
check [<mask>] [reportfile <report>] [noimage] [offset <offset>] [cut <cut>]
```

Check the structure of atoms in <mask> (all atoms if no mask specified) and report problems related to atomic overlap to the file specified by **reportfile** (standard output if not specified). If bond information is present in the topology, problems related to abnormal bond lengths will also be reported. Reports will be made for atoms closer than 0.8 Å (or the value specified by **cut**) and bond lengths greater than the equilibrium value plus 1.0 Å (or the value specified by **offset**).

7.8.6. clusterdihedral

```
clusterdihedral [phibins <N>] [psibins <M>] [out <outfile>] [dihedralfile <dfile>] |
[framefile <framefile>] [clusterinfo <infofile>]
[clustervtime <cvtfil>] [cut <CUT>]
```

Cluster frames in a trajectory using dihedral angles. To define which dihedral angles will be used for clustering either an atom mask or an input file specified by the **dihedralfile** keyword should be used. If dihedral file is used, each line in the file should contain a dihedral to be binned with format:

```
ATOM#1 ATOM#2 ATOM#3 ATOM#4 #BINS
```

where the ATOM arguments are the atom numbers (starting from 0) defining the dihedral and #BINS is the number of bins to be used (so if #BINS=10 the width of each bin will be 36°. If an atom mask is specified, only protein backbone dihedrals (Phi and Psi defined using atom names C-N-CA-C and N-CA-C-N) within the mask will be used, with the bin sizes specified by the phibins and psibins keywords (default for each is 10 bins).

Output will either be written to STDOUT or the file specified by the **out** keyword. First, information about which dihedrals were clustered will be printed. Then the number of clusters will be printed, followed by detailed information of each cluster. The clusters are sorted from most populated to least populated. Each cluster line has format

```
Cluster CLUSTERNUM CLUSTERPOP [ dihedral1bin, dihedral2bin ... dihedralNbin ]
```

followed by a list of frame numbers that belong to that cluster. If a cutoff is specified by **cut**, only clusters with population greater than CUT will be printed.

If specified by the **clustervtime** keyword, the number of clusters for each frame will be printed to <cvtfil>. If specified by the **framefile** keyword, a file containing cluster information for each frame will be written with format

7. cpptraj

Frame CLUSTERNUM CLUSTERSIZE DIHEDRALBINID

where DIHEDRALBINID is a number that identifies the unique combination of dihedral bins this cluster belongs to (specifically it is a 3*number-of-dihedral-characters long number composed of the individual dihedral bins).

If specified by the **clusterinfo** keyword, a file containing information on each dihedral and each cluster will be printed. This file can be read by SANDER for use with REMD with a structure reservoir (-rremd=3). The file, which is essentially a simplified version of the main output file, has the following format:

```
#DIHEDRALS
dihedrall_atom1 dihedrall_atom2 dihedrall_atom3 dihedrall_atom4
...
#CLUSTERS
CLUSTERNUM1 CLUSTERSIZE1 DIHEDRALBINID1
...
```

7.8.7. cluster

```
cluster [[<mask>] [nofit] [mass]] | data <dsetname>
[clusters <n>] [epsilon <e>] [out <cnumvtime>]
[ linkage | averagelinkage | complete ] [noload] [gracecolor]
[summary <summaryfile>] [summaryhalf <halffile>] [info <infofile>]
[ clusterout <trajfileprefix> [clusterfmt <trajformat>] ]
[ singlerepout <trajfilename> [singlerepfmt <trajformat>] ]
[ repout <repprefix> [repfmt <trajformat>] ]
```

Cluster input frames by the atoms in <mask> (all atoms if none specified). If the **out** keyword is given, the cluster number (starting from 0) for each frame will be written to a file specified by <cnumvtime>. If **gracecolor** is specified, a number between 1 and 15 (with 1=most populated cluster and 15 given to all clusters beyond the 15th) will be written to <cnumvtime> instead of the cluster number, which may be suitable for use with plotting programs such as Xmgrace. Currently, the only algorithm is bottom-up (agglomerative) hierarchical. For more clustering methods, users are encouraged to use the Ptraj 'cluster' command.

The distance metric can either be RMSD, or a previously defined dataset specified by the **data** keyword. For example, to cluster on a distance:

```
distance endToEnd :1 :255
cluster data endToEnd clusters 10 epsilon 3.0 summary summary.dat info info.dat
```

By default cpptraj will use frame to frame RMSD of atoms in <mask> as the metric between clusters. Like ptraj clustering, the RMSD metric in cpptraj now uses best-fit RMSD by default. This is in contrast to version 1.X of cpptraj which used no-fit RMSD as default. To use no-fit RMSD in the current version of cpptraj specify the **nofit** keyword. If **mass** is specified the RMSD will be mass-weighted.

In order to speed up subsequent clustering calculations, pair distances between each frame are written to a file called CpptrajPairDist. If this file is present, pair distances will be loaded from it. This behavior can be prevented by specifying the **noload** keyword.

Three types of linkages are supported for calculating distances between clusters. The **linkage** keyword specifies single-linkage, which means the shortest distance between members of two clusters will be used. The **averagelinkage** keyword specifies average-linkage (default), meaning the average distance between members of two clusters will be used. The **complete** keyword specifies complete-linkage, meaning the maximum distance between members of two clusters will be used.

Clustering will complete when either a target number of clusters is reached, specified by the **clusters** keyword (10 if not specified), or when the minimum distance between clusters exceeds a certain value, specified by the **epsilon** keyword. If both **clusters** and **epsilon** are specified, clustering will complete when either criterion is satisfied.

If the **summary** keyword is given, a summary of each cluster will be printed to <summaryfile> with format:

```
#Cluster Frames Frac AvgDist Stdev Centroid AvgCDist
```

where #Cluster is the cluster number (starting from 0), Frames is the number of frames in the cluster, Frac is the fraction of read-in frames represented by the cluster, AvgDist and Stdev are the average and standard deviation of the distances between frames in the cluster, Centroid is the frame with the lowest cumulative distance to every other frame in the cluster (i.e. the frame # of the representative structure), and AvgCDist is the average distance of the cluster to every other cluster. If **summaryhalf** is specified a summary comparing the first half of the clustered trajectory to the second half will be written to <halffile> with similar formatting. If **info** is specified cluster information will be written to <infofile> in a style similar to ptraj. Each cluster is given a line that has # characters equal to the number of frames read, with 'X' representing a frame present in that cluster and '.' representing a frame not in that cluster. Representative frames are written at the end, e.g.:

```
#Clustering: 2 clusters 10 frames
..XX.X....
XX..X.XXXX
#Representative frames: 3 8
```

If the **clusterout** keyword is given, the frames in each cluster will be written to a file named <trajfileprefix>.cX, where X is the cluster number, with a format specified by **clusterfmt** (default is Amber trajectory if not specified). If the **singlerepout** keyword is given, the representative frame (centroid) of each cluster will be written to a file named <trajfilename> with format specified by **singlerepfmt** (default is Amber trajectory if not specified). If the **repout** keyword is given, the representative frame of each cluster will be written to a file named <rep-prefix>.X.<ext> with a format specified by **repfmt** (default is Amber trajectory if not specified), where X is the cluster number and <ext> is assigned automatically based on the file format.

For example, to cluster on the CA atoms of residues 2-10 using average-linkage, stopping when either 3 clusters are reached or the minimum distance between clusters is 4.0, writing the cluster number vs time to "cnumvtime.dat" and a summary of each cluster to "avg.summary.dat":

```
cluster C1 :2-10 clusters 3 epsilon 4.0 out cnumvtime.dat summary avg.summary.dat
```

7. cpptraj

7.8.8. contacts

7.8.9. diffusion

7.8.10. dihedral

```
dihedral [<dataset name>] <mask1> <mask2> <mask3> <mask4> [out <filename>] [mass]
```

Calculate dihedral angle (in degrees) between the planes defined by atoms in <mask1>, <mask2>, <mask3> and <mask2>, <mask3>, <mask4>. If **mass** is specified use the center of mass of atoms in the masks instead of geometric center.

7.8.11. dipole

7.8.12. distance

```
distance [<dataset name>] <mask1> <mask2> [out <filename>] [geom] [noimage]
```

Calculate distance between the center of mass of atoms in <mask1> to atoms in <mask2>. If **geom** is specified use the geometric center instead. For periodic systems imaging is turned on by default; the **noimage** keyword disables imaging.

7.8.13. drmsd (distance RMSD)

```
drmsd [<dataset name>] [<mask> [<refmask>]] [out <filename>]  
[ first | ref <reffilename> | reindex <#> |  
  reftraj <trajname> [parm <trajparm> | parmindex <parm#>] ]
```

Calculate the distance RMSD (i.e. the RMSD of all pairs of internal distances) between atoms in the frame defined by <mask> (all if no <mask> specified) to atoms in a reference defined by <refmask> (<mask> if <refmask> not specified). Both <mask> and <refmask> must specify the same number of atoms, otherwise an error will occur. The Reference structure is defined by one of the following keywords (of which only one should be specified):

- **first**: Use the first trajectory frame processed as reference.
- **reference**: Use the first previously read in reference structure (reindex 0).
- **ref**: Use previously read in reference structure specified by <reffilename>.
- **reindex**: Use previously read in reference structure specified by <#> (based on order read in).
- **reftraj**: Use frames read in from <trajname> with associated parmfile specified by name <trajparm> or index <parm#>; if parm is not specified the first parm read in is used. Each frame from <trajname> is used in turn, so that frame 1 is compared to frame 1 from <trajname>, frame 2 is compared to frame 2 from <trajname> and so on. If <trajname>

runs out of frames before processing is complete, the last frame of <trajname> continues to be used as the reference.

Because this method compares pairs of internal distances and not absolute coordinates, it is not sensitive to translations and rotations the way that a no-fit RMSD calculation is. It can be more time consuming however, as $(N^2-N)/2$ distances must be calculated and compared for both the target and reference structures.

For example, to get the DRMSD of a residue named LIG to its structure in the first frame read in:

```
drmsd :LIG first out drmsd.dat
```

7.8.14. dnaiontracker

7.8.15. grid

7.8.16. gfe (GridFreeEnergy)

7.8.17. hbond

```
hbond [<dataset name>] [out <filename>] <mask> [angle <cut>] [dist <cut>]
      [avgout <avgfilename>] [donormask <dmask>] [acceptormask <amask>]
      [solventdonor <mask>] [solventacceptor <mask>] [solvout <sfilename>] [bridgeout <bfilename>]
```

Determine hydrogen bonds (currently only solute-solute). Search for hydrogen bond donor and acceptor atoms in the region specified by <mask> (all solute atoms if no mask specified), following the simplistic criterion that “hydrogen bonds are FON”, i.e., hydrogens bonded to F, O, and N atoms are considered. Hydrogen bonding atoms can also be specified with the **donormask** and/or **acceptormask** keywords:

1. If just <mask> is specified donors and acceptors will be automatically determined from <mask>.
2. If **donormask** is specified donors will be determined from <dmask> (only atoms bonded to hydrogen will be considered valid). Acceptors will be automatically determined from <mask>.
3. If **acceptormask** is specified acceptors will be determined from <amask>. Donors will be automatically determined from <mask>.
4. If both **acceptormask** and **donormask** are specified only <amask> and <dmask> will be used; no searching will occur in <mask>.

The number of hydrogen bonds present at each frame will be determined and written to the file specified by **out**. Hydrogen bonds are considered to have the form:

```
Acceptor ... Hydrogen-Donor
```

7. cpptraj

and are determined via the distance between the heavy atoms using a cutoff of 3.0 Å (or the value specified by **dist**) and the angle between the acceptor, hydrogen, and donor atoms using a cutoff of 135° (or the value specified by **angle**). Note that imaging is not employed when calculating distance (since this makes the calculation orders of magnitude slower); if imaging is required an **image** command should be performed prior to the **hbond** command to ensure atoms that will hydrogen bond are not separated by periodic boundaries.

If **avgout** is specified the average of each hydrogen bond (sorted by population) formed over the course of the trajectory is printed to <avgfilename>. The output file has the format:

```
Accepter  DonorH  Donor  Frames  Frac  AvgDist  AvgAng
```

where *Accepter*, *DonorH*, and *Donor* are the residue and atom name of the atoms involved in the hydrogen bond, *Frames* is the number of frames the bond is present, *Frac* is the fraction of frames the bond is present, *AvgDist* is the average distance of the bond, and *AvgAng* is the average angle of the bond.

For example, to search for all hydrogen bonds within residues 1-22, writing the number of hydrogen bonds per frame to “nhb.dat” and information on each hydrogen bond found to “avghb.dat”:

```
hbond :1-22 out nhb.dat avgout avghb.dat
```

To search for all hydrogen bonds formed between donors in residue 1 and acceptors in residue 2:

```
hbond donormask :1 acceptormask :2 out nhb.dat avgout avghb.dat
```

If masks are specified with the **solventdonor** and/or **solventacceptor** keywords, solute-solvent hydrogen bonds will also be tracked. The number of solute-solvent hydrogen bonds and number of “bridging” solvent molecules (i.e. solvent that is hydrogen bonded to two or more different solute residues at the same time) will also be printed to the file specified by **out**. If **solvout** is specified the average of each solute-solvent hydrogen bond formed over the course of the trajectory will be written to <sfilename> in a manner analogous to **avgout**. Note that for solute-solvent hydrogen bonds the ‘Frames’ column becomes ‘Count’ since for any given frame more than 1 solvent molecule can bind to the same place on solvent and vice versa. If **bridgeout** is specified information on residues that were bridged by a solvent molecule over the course of the trajectory will be written to <bfilename> with format:

```
Bridge Res <N0:RES0> <N1:RES1> ... , <X> frames.
```

where ‘<N0:RES0> ...’ is a list of residues that were bridged (residue # followed by residue name) and <X> is the number of frames the residues were bridged.

7.8.18. jcoupling

```
jcoupling [<mask1>] [outfile <filename>]
```

Calculate J-coupling values for all dihedrals found within <mask1> (all atoms if no mask given). In order to use this function, Karplus parameters for all dihedrals which will be calculated must

be loaded. By default cpptraj will use the data found in \$AMBERHOME/dat/Karplus.txt; if this is not found cpptraj will look for the file specified by the \$KARPLUS environment variable.

In the Karplus parameter file each parameter set consists of two lines for each dihedral with the format:

```
[<Type>] <Name1><Name2><Name3><Name4><A><B><C> [<D>]
<Resname1> [<Resname2> . . .]
```

The first line defines the parameter set for a dihedral. <Type> is optional; if not given the form for calculating the J-coupling will be as described by Chou et al.[118]; if 'C' the form will be as described by Perez et al.[119]. The <NameX> parameters define the four atoms involved in the dihedral. Each <NameX> parameter is 5 characters wide, starting with a plus '+', minus '-' or space ' ' character indicating the atom belongs to the next, previous, or current residue. The remaining 4 characters are the atom name. The parameters <A>, , <C>, and <D> are floating point values 6 characters wide describing the Karplus parameters. For the 'C' form A, B, and C correspond to C0, C1, and C2; D is unused and should not be specified. The second line is a list of residue names (4 characters each) to which the dihedral applies. For example:

```
C HA  CA  CB  HB      5.40 -1.37  3.61
ILE VAL
```

Describes a dihedral between atoms HA-CA-CB-HB using the Perez et al. form with constants C0=5.40, C1=-1.37, C2=3.61 applied to ILE and VAL residues.

Output is sent to <filename>. Each dihedral that is defined from <mask1> is printed along with its calculated J-coupling value for each frame, e.g.:

```
#Frame 1
1 SER HA CA CB HB2 45.334742 4.024759
1 SER HA CA CB HB3 -69.437134 1.829510
...
```

First the frame number is printed, then for each dihedral: Residue number, residue name, atom names 1-4 in the dihedral, the value of the dihedral, the J-coupling value.

7.8.19. mask

```
mask <mask> [maskout <filename>] [maskpdb <pdbname>]
```

For each frame determine all atoms that correspond to <mask>. This is most useful when using distance-based masks, since the atoms in the mask are updated for every frame read in. If **maskout** is specified information on all atoms in <mask> will be written to <filename>. If **maskpdb** is specified a PDB file corresponding to <mask> will be written out every frame with name "<pdbname>.frame#".

For example, to write out all atoms within 3.0 Angstroms of residue 195 that are part of residues named WAT to "Res195WAT.dat", as well as write out corresponding PDB files:

```
mask "(:195<:3.0)&:WAT" maskout Res195WAT.dat maskpdb Res195WAT.pdb
```

7.8.20. molsurf

```
molsurf [<dataset_name>] [<mask>] [out <filename>]
        [probe <probe_rad>] [offset <rad_offset>]
```

Calculate the Connolly surface area[120] of atoms in <mask> (default all atoms if no mask specified) using routines from molsurf (originally developed by Paul Beroza) using the probe radius specified by **probe** (1.4 Å if not specified). This routine currently requires radius information to be present in the topology file. If **offset** is given <rad_offset> will be added to radii.

7.8.21. nastruct

```
nastruct [resrange <range>] [naout <suffix>] [noheader]
        [resmap <ResName>:{A,C,G,T,U} ...]
        [hbcut <hbcut>] [origincut <origincut>]
        [ reference | refindex <#> | ref <REF> ]
```

Calculate basic nucleic acid (NA) structure parameters for all residues in the range specified by **resrange** (or all NA residues if no range specified). Residue names are recognized with the following priority: standard Amber residue names DA, DG, DC, DT, RA, RG, RC, and RU; 3 letter residue names ADE, GUA, CYT, THY, and URA; and finally 1 letter residue names A, G, C, T, and U. Non-standard/modified NA bases can be recognized by using the **resmap** keyword. For example, to make *cpptraj* recognize all 8-oxoguanine residues named '8OG' as a guanine-based residue:

```
nastruct naout nastruct.dat resrange 274-305 resmap 8OG:G
```

The **resmap** keyword can be specified multiple times, but only one mapping per unique residue name is allowed. Note that **resmap** may fail if the residue is missing heavy atoms normally present in the specified base type.

Base pairs can either be determined each frame, or one time from a reference structure; the **reference** keyword uses the first reference read in, the **refindex** keyword specifies reference structure by index (starting from 0) and **ref** specifies reference by filename/tag. Base pairing is determined first by base reference axis origin distance (cutoff is 2.5 Å or the value specified by **origincut**), then by Watson-Crick hydrogen bonding (cutoff 3.5 Å or the value specified by **hbcut**). Base pair parameters will only be written for determined base pairs. Note that currently only anti-parallel Watson-Crick base-pairs are recognized; future releases will include support for recognizing more types of base pairs.

The procedure used to calculate NA structural parameters is the same as 3DNA[121], with algorithms adapted from Babcock et al.[122] and reference frame coordinates from Olson et al.[123]. Given the same base pairs are determined, *Cpptraj* nastruct gives the exact same numbers as 3DNA.

Calculated NA structure parameters are written to three separate files, the suffix of which is specified by **naout**. Base pair parameters (shear, stretch, stagger, buckle, propeller twist, and opening) are written to BP.<suffix>, along with the number of WC hydrogen bonds detected. Base pair step parameters (shift, slide, rise, tilt, roll, and twist) are written to BPstep.<suffix>.

and helical parameters (X-displacement, Y-displacement, rise, inclination, tip, and twist) are written to `Helix.<suffix>`. If **noheader** is specified a header will not be written to the output files.

7.8.22. outtraj

```
outtraj <filename> [ trajout args ] [maxmin <datasetname> min <min> max <max>]
```

The outtraj command is similar in function to trajout, and takes all of the same arguments. However, instead of writing a trajectory frame after all actions are complete outtraj writes the trajectory frame at its position in the action stack. For example, given the input:

```
trajin mdcrd.crd
trajout output.crd
outtraj BeforeRmsd.crd
rms R1 first :1-20@CA out rmsd.dat
outtraj AfterRmsd.crd
```

three trajectories will be written: output.crd, BeforeRmsd.crd, and AfterRmsd.crd. The output.crd and AfterRmsd.crd trajectories will be identical, but the BeforeRmsd.crd trajectory will contain the coordinates of mdcrd.crd before they are RMS-fit.

The **maxmin** keyword can be used to control what frames are written based on the values of a previously defined dataset. For example, to only print out structures with an RMSD between 0.0 and 1.0 in netcdf format:

```
trajin mdcrd.crd
rms R1 first :1-20@CA
outtraj Rmsd_0.0-1.0.nc netcdf maxmin R1 min 0.0 max 1.0
```

7.8.23. principal

7.8.24. projection

7.8.25. pucker

```
pucker [<dataset name>] <mask1> <mask2> <mask3> <mask4> <mask5> [mass]
      [out <filename>] [range360] [amplitude] [altona | cremer] [offset <offset>]
```

Calculate the pucker (in degrees) for atoms in <mask1>, <mask2>, <mask3>, <mask4>, <mask5> using the method of Altona & Sundarlingam[124, 125] (default, or if **altona** specified), or the method of Cremer & Pople[126] if **cremer** is specified.

If the **amplitude** keyword is given, amplitudes will be calculated instead of the pseudorotation. If **mass** is specified use the center of mass of atoms in the masks instead of geometric center.

By default, pucker values are wrapped to range from -180 to 180 degrees. If the **range360** keyword is specified values will be wrapped to range from 0 to 360 degrees.

7. cpptraj

Note that the Cremer & Pople convention is offset from Altona & Sundarlingam convention (with nucleic acids) by +90.0 degrees; the **offset** keyword will add an offset to the final value and so can be used to convert between the two. For example, to convert from Cremer to Altona specify “**offset 90**”.

To calculate nucleic acid pucker specify C1' first, followed by C2', C3', C4' and O4'. For example, to calculate the sugar pucker for nucleic acid residues 1 and 2 using the method of Altona & Sundarlingam, with final pseudorotation values ranging from 0 to 360:

```
pucker p1 :1@C1' :1@C2' :1@C3' :1@C4' :1@O4' range360 out pucker.dat
pucker p2 :2@C1' :2@C2' :2@C3' :2@C4' :2@O4' range360 out pucker.dat
```

7.8.26. radgyr

```
radgyr [<dataset name>] [<mask>] [out <filename>] [mass] [nomax]
```

Calculate the radius of gyration of atoms in <mask> (all atoms if no mask specified). Output radius of gyration and max radius of gyration to <filename>. If the **nomax** keyword is specified do not print max radius of gyration. If the **mass** keyword is specified the radius of gyration will be calculated with respect to the center of mass of atoms in <mask>, and will be mass-weighted.

For example, to calculate only the mass-weighted radius of gyration (not the max as well) of the non-hydrogen atoms of residues 4 to 10 and print the results to “RoG.dat”:

```
radgyr :4-10&!(@H=) out RoG.dat mass nomax
```

7.8.27. radial

```
radial <output_filename> <spacing> <maximum> <mask1> [<mask2>] [noimage]
      [density <density> | volume] [center1]
```

Calculate the radial distribution function (RDF, aka pair correlation function) of atoms in <mask1> to atoms in <mask2> (or <mask1> if a second mask is not given), with a bin spacing of <spacing> and a bin maximum of <maximum>, and write the resulting RDF to a file named <output_filename>. If the **center1** keyword is given, the RDF of all atoms in <mask2> to the geometric center of atoms in <mask1> is calculated.

The RDF is essentially a histogram of the number of particles found as a function of distance R, normalized by the expected number of particles at that distance, which is calculated from:

$$Density * \left(\left[\frac{4\pi}{3} (R + dR)^3 \right] - \left[\frac{4\pi}{3} dR^3 \right] \right)$$

where dR is equal to the bin spacing. Some care is required by the user in order to normalize the RDF correctly. The default density value is 0.033456 molecules Å⁻³, which corresponds to a density of water approximately equal to 1.0 g mL⁻¹. To convert a standard density in g mL⁻¹, multiply the density by $\frac{0.6022}{M_r}$, where M_r is the mass of the molecule in atomic mass

units. Alternatively, if the **volume** keyword is specified the density is determined from the average volume of the system over all Frames.

Note that correct normalization of the RDF depends on the number of atoms in each mask; if multiple topology files are being processed that result in changes in the number of atoms in each mask, the normalization will be off.

7.8.28. randomizeions

7.8.29. rmsavgccorr

```
rmsavgccorr [<dataset name>] [<mask>] [mass] [stop <maxwindow>]
            [out <filename>]
```

Calculate correlation of RMSD by calculating the average RMSD of running-averaged coordinates over increasing window sizes. Output has format:

```
<WindowSize> <AvgRmsd>
```

The first entry has window size of 1, and so is just the average RMSD of the structure to the first frame. The second entry has a window size of two, so it is the average RMSD of all frames averaged over two adjacent windows to the average of the first two frames, and so on. Average RMSDs will be calculated up to the number of frames minus 1 or the value specified by **stop**, whichever is lower. To calculate mass-weighted RMSD specify **mass**. Note that to reduce memory costs it can be useful to strip all coordinates not involved in the RMS fit from the system prior to specifying 'rmsavgccorr'. For example, to calculate the correlation of C-alpha RMSD of residues 2 to 12:

```
strip !(:2-12@CA)
rmsavgccorr out rmscorr.dat
```

7.8.30. rmsd

```
rmsd [<dataset name>] [<mask> [<refmask>]] [out <filename>] [nofit] [mass]
    [ first | reference | ref <reffilename> | refindex <#> |
      reftraj <trajname> [parm <trajparm> | parmindex <parm#>] ]
    [ perres perresout <perresfile> | perresavg <avgfile>
      [range <resRange>] [refrange <refRange>]
      [perresmask <pmask>] [perrescenter] [perresinvert] ]
```

Note: For backwards compatibility with ptraj the command 'rms' will also work.

Calculate the RMSD between atoms in Frame defined by <mask> (all atoms if no <mask> specified) to atoms in Reference defined by <refmask> (<mask> if no <refmask> specified). Both <mask> and <refmask> must specify the same number of atoms, otherwise an error will occur. The Reference structure is defined by one of the following keywords (of which only one should be specified):

- **first**: Use the first trajectory frame processed as reference.

7. cpptraj

- **reference**: Use the first previously read in reference structure (refindex 0).
- **ref**: Use previously read in reference structure specified by <reffilename>/<tag>.
- **refindex**: Use previously read in reference structure specified by <#> (based on order read in).
- **reftraj**: Use frames read in from <trajname> with associated parmfile specified by name <trajparm> or index <parm#>; if parm is not specified the first parm read in is used. Each frame from <trajname> is used in turn, so that frame 1 is compared to frame 1 from <trajname>, frame 2 is compared to frame 2 from <trajname> and so on. If <trajname> runs out of frames before processing is complete, the last frame of <trajname> continues to be used as the reference.

If **nofit** is specified the Frame coordinates will not be best-fit to reference coords prior to RMSD calculation. If **mass** keyword is specified the RMSD will be mass-weighted.

For example, say you have a trajectory and you want to calculate RMSD to two separate reference structures. To calculate the best-fit RMSD of the C, CA, and N atoms of residues 1 to 20 in each frame to the C, CA, and N atoms of residues 3 to 23 in StructX.crd, and then calculate the no-fit RMSD of residue 7 to residue 7 in another structure named Struct-begin.rst7, writing both results to Grace-format file “rmsd1.agr”:

```
reference StructX.crd [structX]
reference md_begin.rst7 [struct0]
rmsd BB :1-20@C,CA,N ref [structX] :3-23@C,CA,N out rmsd1.agr
rmsd Res7 :7 ref [struct0] out rmsd1.agr nofit
```

Per-residue RMSD calculation

If the **perres** keyword is specified, after the initial RMSD calculation the no-fit RMSD of each residue in each Frame specified by <resRange> (or all solute residues if range not specified) will be calculated to each residue in Reference specified by <refRange> (or each residue in <resRange> if reference range not specified). The results will be written to the file specified by **perresout**, or time-averaged results can be written to the file specified with **perresavg**. By default all atoms of each residue are used. An additional mask can be specified by using **perresmask**; this mask is appended to the default mask of each residue, so the resulting mask is “:X<pmask>”, where X is residue number. If **perrescenter** is specified residues will be centered to a common point of reference before no-fit RMSD is calculated (this will emphasize changes in the local structure of the residue). So for example:

```
rmsd :10-260 refindex 0 perres perresout PRMS.dat range 190-211 perresmask &!(@H=)
```

will first perform an rms-fit calculation on residues 10-260, then calculate the per-residue no-fit rmsd of residues 190 to 211 (excluding any hydrogen atoms), writing the results to PRMS.dat.

If **perresinvert** is specified, data for each residue in <perresfile> (not <avgfile>) will be written in rows instead of columns, i.e.:

```
RES1    RmsdFrame1 RmsdFrame2 RmsdFrame3 ...
RES2    RmsdFrame1 RmsdFrame2 RmsdFrame3 ...
```

7.8.31. rms2d

```
rms2d [<mask> [<refmask>]] rmsout <filename> [mass] [nofit]
      [reftraj <traj> [parm <parmname> | parmindex <parm#>]]
      [corr <corrfilename>]
```

Note: For backwards compatibility with ptraj the command '2drms' will also work.

Calculate the RMSD of atoms in <mask> (all atoms if none specified) to atoms in <refmask> (<mask> if no reference mask specified) between all frames read in. If **reftraj** is specified, calculate RMSD between all frames read in and all frames of <traj>, which will be associated with the parm file specified by <parmname> or <parm#> (the first parm read in if none specified).

If the **mass** keyword is specified the RMSD will be mass-weighted. Note that mass-weighted rmsd will only be calculated if one topology file is used for input trajectories; for multiple topology files **mass** is disabled. The best-fit RMSD is calculated by default; if the **nofit** keyword is specified the RMSD without fitting will be calculated.

The output of the rms2d command can be best-viewed using gnuplot; a gnuplot-formatted file can be produced by giving <filename> a '.gnu' extension. For example, to calculate the RMSD of non-hydrogen atoms of each frame in trajectory "test.nc" to each other frame, writing to a gnuplot-viewable file "test.2drms.gnu":

```
trajin test.nc
rms2d !(@H=) rmsout test.2drms.gnu
```

To calculate the RMSD of atoms named CA of each frame in trajectory "test.nc" to each frame in "ref.nc" (assuming test.nc and ref.nc are using the default topology file):

```
trajin test.nc
rms2d @CA rmsout test.2drms.gnu reftraj ref.nc
```

7.8.32. rotdif

```
rotdif [rseed <rseed>] [nvecs <nvecs>]
      [ref <refname> | refindex <refindex> | reference] [<refmask>]
      [ncorr <ncorr>] [nmesh <nmesh>] dt <tfac> [ti <ti>] tf <tf>
      [itmax <itmax>] [tol <delmin>] [d0 <d0>] [order <olegendre>]
      [delqfrac <delqfrac>] [gridsearch] [rvecout <randvecOut>]
      [rmout <rmOut>] [deffout <deffOut>] [outfile <outfilename>]
      [rvecin <randvecIn>]
```

Evaluate rotational diffusion properties over a trajectory according to the procedure laid out by Wong & Case[127]. Briefly, random vectors (can be thought of as analogous to e.g. N-H bond vectors) are rotated according to rotation matrices obtained from an RMS fit to a reference structure (typically an averaged structure). For each random vector the time correlation function of the rotated vector is calculated. The time correlation function can then be used to solve for the effective value of the diffusion constant (deff) for that vector. Given an effective diffusion constant for each vector, solve for the diffusion tensor D assuming small anisotropy. Finally,

7. *cpptraj*

based on D in the small anisotropic limit, a downhill simplex minimizer is used to optimize D with full anisotropy.

Rotation matrices are generated via an RMS fit to the reference structure specified by name (**ref**) or index (**refindex**); **reference** uses the first reference structure read in. It is recommended that the RMS fit be done to an average structure (see the *average* command). Rotation matrices can be written (row-major) to a file specified by **rmout**.

The number of random vectors to generate is specified by **nvecs**; alternatively random vectors can be read in from a file specified by **rvecin** with format:

```
<VectorNum> <Vx> <Vy> <Vz>
```

where <VectorNum> is an integer (not used internally), and Vx, Vy, and Vz are the x, y, and z components of the vector. Random vectors can be written out in the same format to a file specified by **rvecout**.

The order of Legendre polynomials to use (correlation and tau) is specified by **olegendre**; currently this must be 1 or 2 (default 2). The maximum length of the correlation function (or lag) is specified by **ncorr** (in frames). The default is to use all frames; however it is recommended that **ncorr** be set to a number less than the total number of frames since noise tends to increase as **ncorr** approaches the # of frames. The integration over the correlation function is from **ti** (in ns) (0.0 if not specified) to **tf** (also in ns), with a timestep specified by **dt**; the final time should be less than **ncorr * dt**. The relative size of the mesh used with cubic spline interpolation for integration is controlled by **nmesh** (size of the mesh is **ncorr** points * **nmesh**); **nmesh** = 1 means no interpolation, default is 2. The iterative solver for effective value of the diffusion constant from the correlation functions is controlled by **itmax**, **tol**, and **d0**, where **itmax** specifies the number of iterations to perform (default 500), **tol** specifies the tolerance (default 1E-6), and **d0** specifies the initial guess for the diffusion constant (default 0.03). Effective diffusion constants for each random vector can be written out to a file specified by **deffout**.

The random number generator (used in generating random vectors and by the simplex minimizer) is seeded with the value given by **rseed** (80531 by default). If the random seed is less than 1 the wallclock time is used. **delqfrac** controls the scaling of simplexes when fitting D with full anisotropy (default 0.5).

Results are printed to the file specified by **outfile**. Details on the Q and D tensors are given, as well as observed and calculated tau for each random vector. First, results are printed for analysis in the limit of small anisotropy. Next, results are printed for analysis with full anisotropy. The results of the full anisotropic calculation are first given using results from the small anisotropic analysis as an initial guess, followed by the final results after minimization using the downhill simplex (amoeba) minimizer.

For example, given a trajectory 'mdcrd.nc' containing 100 frames, to calculate rotational diffusion using 100 vectors using rotation matrices generated via an RMS fit to 'avgstruct.pdb', computing the correlation function for each vector using a max lag of 90 frames, integrating from 0 ns to 0.180 ns with a timestep of 0.002 ns, and writing out the effective diffusion constants and results to 'deffs.dat' and 'rotdif.out' respectively:

```
reference avgstruct.pdb [avg]
trajin mdcrd.nc 1 100
rotdif nvecs 100 ref [avg] @CA,C,N,O \
```

```
ncorr 90 ti 0.0 tf 0.180 dt 0.002 deffout deffs.dat \
itmax 500 tol 0.000001 d0 0.03 order 2 \
outfile rotdif.out
```

7.8.33. scale

7.8.34. secstruct

```
secstruct [<dataset name>] [out <filename>] [<mask>] [sumout <sumfilename>]
[ptrajformat] [namen <N name>] [nameh <H name>]
[namec <C name>] [nameo <O name>]
```

Calculate secondary structural propensities for residues in <mask> (or all solute residues if no mask given) using the DSSP method of Kabsch and Sander[128], which assigns secondary structure types for residues based on backbone amide (N-H) and carbonyl (C=O) atom positions. By default cpptraj assumes these atoms are named “N”, “H”, “C”, and “O” respectively. If a different naming scheme is used (e.g. amide hydrogens are named “HN”) the backbone atom names can be customized with the **nameX** keywords (e.g. 'nameH HN'). Note that it is expected that some residues will not have all of these atoms (such as proline); in this case cpptraj will print an informational message but the calculation will proceed normally.

Results will be written to filename specified by **out** with format:

```
<#Frame>    <ResX SS> <ResX+1 SS> ... <ResN SS>
```

where <#Frame> is the frame number and <ResX SS> is an integer representing the calculated secondary structure type for residue X. If the keyword **ptrajformat** is specified, the output format will instead be:

```
<#Frame>    STRING
```

where STRING is a string of characters (one for each residue) where each character represents a different structural type (this format is similar to what ptraj outputs). The various secondary structure types and their corresponding integer/character are listed below:

Character	Integer	SS type
0	0	None
b	1	Parallel Beta-sheet
B	2	Anti-parallel Beta-sheet
G	3	3-10 helix
H	4	Alpha helix
I	5	Pi (3-14) helix
T	6	Turn

Average structural propensities over all residues for each frame will be written to the file specified by **sumout** (or “<filename>.sum” if **sumout** is not specified).

7. cpptraj

The output of `secstruct` command in particular is amenable to visualization with `gnuplot`. To generate a 2D map-style plot of secondary structure vs time, with each residue on the Y axis simply give the output file a “.gnu” extension. For example, to generate a 2D map of secondary structure vs time, with different colors representing different secondary structure types for residues 1-22:

```
secstruct :1-22 out dssp.gnu
```

The resulting file can be visualized with `gnuplot`:

```
gnuplot dssp.gnu
```

Similarly, the `sumout` file can be nicely visualized using `xmgrace` (use “.agr” extension).

```
secstruct :1-22 out dssp.gnu sumout dssp.agr
xmgrace dssp.agr
```

7.8.35. surf

```
surf [<dataset name>] [<mask>] [out <filename>]
```

Calculate the surface area in \AA^2 of atoms in <mask> (all solute atoms if no mask specified) using the LCPO algorithm of Weiser et al.[129]. In order for this to work, the topology needs to have bond information and atom type information. For topologies with no bond information (e.g. PDB files), bond information can be set up by specifying ‘**bondsearch**’ prior to the ‘*parm*’ command.

Note that even if <mask> does not include all solute atoms, the neighbor list is still calculated for all solute atoms so the surface area calculated reflects the contribution of atoms in <mask> to the overall surface area, not the surface area of <mask> as an isolated system. As a result, it may be possible to obtain a negative surface area if only a small fraction of the solute is selected.

For example, to calculate the overall surface area of all solute atoms, as well as the contribution of residue 1 to the overall surface area, writing both results to “surf.dat”:

```
surf out surf.dat
surf :1 out surf.dat
```

7.8.36. watershed

7.9. Matrix and Vector Actions

7.9.1. matrix

7.9.2. vector

7.10. Analysis Commands

Similar to `ptraj`, analysis occurs after all trajectories have been read in and processed and all actions have completed their ‘print’ phase. In general, any dataset created by an action with an ‘out <datafile>’ command is available for analysis.

7.10.1. corr / analyze correlationcoe

```
corr out <outfilename> <dataset1> <dataset2> [lagmax <lagmax>]
```

Calculate the correlation between datasets named <dataset1> and <dataset2> for lag = 0 to <lagmax> frames (all if **lagmax** not specified), writing the result to file specified by **out**. The two datasets must have the same # of datapoints. If <dataset1> and <dataset2> are the same dataset this is the auto-correlation.

7.10.2. analyze crank

7.10.3. hist(ogram)

```
hist <dataset_name>[:min:max:step:bins] ...
    [free <temperature>] [norm] [circular] out <filename>
    min <min> max <max> step <step> bins <bins>
```

Create an N-dimensional histogram, where N is the number of datasets specified, and write to file named <filename>. For 1-dimensional histograms the xmgrace '.agr' file format is recommended; for 2-dimensional histograms the gnuplot '.gnu' file format is recommended; for all other dimensions standard '.dat' format is recommended.

The min, max, step, and/or number of bins can be specified after the dataset name separated by colons. It is only necessary to specify the step or number of bins, an asterisk '*' indicates the value should be calculated from available data. Default values of **max**, **min**, **step**, and **bins** can also be specified with the corresponding keywords; defaults will be used if not specified after the dataset name. For example, to create a two dimensional histogram of two datasets 'phi' and 'psi':

```
dihedral phi :2@C :3@N :3@CA :3@C
dihedral psi :3@N :3@CA :3@C :4@N
hist phi:-180:180:*:72 psi:-180:180:*:72 out hist.gnu
```

In this case the number of bins (72) has been specified for each dimension and '*' has been given for the step size, indicating it should be calculated based on min/max/bins. The following 'hist' command is equivalent:

```
hist phi psi min -180 max 180 bins 72 out hist.gnu
```

If '**free** <temperature>' is specified, the population based free energy for each bin (G_i) will be calculated based on:

$$G_i = -k_B T \ln \left(\frac{N_i}{N_{Max}} \right)$$

Where K_B is Boltzmann's constant, T is the temperature specified by <temperature>, N_i is the population of bin i and N_{Max} is the population of the most populated bin. Bins with no population are given an artificial barrier equivalent to a population of 0.5.

If **norm** is specified bin populations will be normalized. If **circular** is specified the data is assumed to wrap (e.g. in the case of a dihedral angle) and an extra bin will be printed before min and after max in each direction for visualization purposes.

7. *cpptraj*

7.10.4. analyze matrix

7.10.5. analyze modes

7.10.6. analyze stat

7.10.7. analyze timecorr

8. PBSA

Several efficient finite-difference numerical solvers, both linear [130, 131] and nonlinear,[132] are implemented in *pbsa* for various applications of the Poisson-Boltzmann method. In the following, a brief introduction is given on the method, the numerical solvers, and numerical energy and force calculations. This is followed by a detailed description of the usage and keywords. Finally example input files are explained for typical *pbsa* applications. For more information on the background and how to use the method, please consult cited references and online *Amber* tutorial pages.

8.1. Introduction

Solvation interactions, especially solvent-mediated dielectric screening and Debye-Hückel screening, are essential determinants of the structure and function of proteins and nucleic acids.[133] Ideally, one would like to provide a detailed description of solvation through explicit simulation of a large number of solvent molecules and ions. This approach is frequently used in molecular dynamics simulations of solution systems. In many applications, however, the solute is the focus of interest, and the detailed properties of the solvent are not of central importance. In such cases, a simplified representation of solvation, based on an approximation of the mean-force potential for the solvation interactions, can be employed to accelerate the computation.

The mean-force potential averages out the degrees of freedom of the solvent molecules, so that they are often called implicit or continuum solvents. The formalism with which implicit solvents can be applied in molecular mechanics simulations is based on a rigorous foundation in statistical mechanics, at least for additive molecular mechanics force fields. Within the formalism, it is straightforward to understand how to decompose the total mean-field solvation interaction into electrostatic and non-electrostatic components that scale quite differently and must be modeled separately (see for example [134]).

The Poisson-Boltzmann (PB) solvents are a class of widely used implicit solvents to model solvent-mediated electrostatic interactions.[133] They have been demonstrated to be reliable in reproducing the energetics and conformations as compared with explicit solvent simulations and experimental measurements for a wide range of systems.[133] In these models, a solute is represented by an atomic-detail model as in a molecular mechanics force field, while the solvent molecules and any dissolved electrolyte are treated as a structure-less continuum. The continuum treatment represents the solute as a dielectric body whose shape is defined by atomic coordinates and atomic cavity radii.[135] The solute contains a set of point charges at atomic centers that produce an electrostatic field in the solute region and the solvent region. The electrostatic field in such a system, including the solvent reaction field and the Coulombic field, may be computed by solving the PB equation:[136, 137]

8. PBSA

$$\nabla \cdot [\epsilon(\mathbf{r}) \nabla \phi(\mathbf{r})] = -4\pi\rho(\mathbf{r}) - 4\pi\lambda(\mathbf{r}) \sum_i z_i c_i \exp(-z_i \phi(\mathbf{r})/k_B T) \quad (8.1)$$

where $\epsilon(\mathbf{r})$ is the dielectric constant, $\phi(\mathbf{r})$ is the electrostatic potential, $\rho(\mathbf{r})$ is the solute charge, $\lambda(\mathbf{r})$ is the Stern layer masking function, z_i is the charge of ion type i , c_i is the bulk number density of ion type i far from the solute, k_B is the Boltzmann constant, and T is the temperature; the summation is over all different ion types. The salt term in the PB equation can be linearized when the Boltzmann factor is close to zero. However, the approximation apparently does not hold in highly charged systems. Thus, it is recommended that the full nonlinear PB equation solvers be used in such systems.

The non-electrostatic or non-polar (NP) solvation interactions are typically modeled with a term proportional to the solvent accessible surface area (SASA).[138] An alternative and more accurate method to model the non-polar solvation interactions is also implemented in *pbsa*. [90] The new method separates the non-polar solvation interactions into two terms: the attractive (dispersion) and repulsive (cavity) interactions. Doing so significantly improves the correlation between the cavity free energies and solvent accessible surface areas or molecular volumes enclosed by SASA for branched and cyclic organic molecules.[139] This is in contrast to the commonly used strategy that correlates total non-polar solvation energies with solvent accessible surface areas, which only correlates well for linear aliphatic molecules.[138] In the alternative method, the attractive free energy is computed by a numerical integration over the solvent accessible surface area that accounts for solvation attractive interactions with an infinite cutoff.[140]

8.1.1. Numerical solutions of the PB equation

In *pbsa* both the linear form and the full nonlinear form of the PB equation are supported. Many strategies may be used to discretize the PB equation, but only the finite-difference (FD) method, or more rigorously, the finite-volume method [141–143] is fully supported in *pbsa* for both the linear and nonlinear PB equations. A FD method involves the following steps: mapping atomic charges to the FD grid points (termed grid charges below); assigning non-periodic/periodic boundary conditions, *i.e.*, electrostatic potentials on the boundary surfaces of the FD grid; and applying a dielectric model to define the high-dielectric (*i.e.*, water) and low-dielectric (*i.e.*, solute interior) regions and mapping it to the FD grid edges.

These steps allow the partial differential equation to be converted into a linear or nonlinear system with the electrostatic potential on grid points as unknowns, the charge distribution on the grid points as the source, and the dielectric constant on the grid edges (and the salt-related term for the linear case) wrapped into the coefficient matrix, which is a seven-banded symmetric matrix. In *pbsa*, four common linear FD solvers are implemented: modified ICCG, geometric multigrid, conjugate gradient, and successive over-relaxation (SOR).[131] In addition, we have also implemented six nonlinear FD solvers: Inexact Newton(NT)/modified ICCG, NT/geometric multigrid, conjugate gradient, and SOR and its improved versions - adaptive SOR and damped SOR.[132]

In addition to the FD method, a new discretization strategy is also introduced to solve the linear PB equation.[144] The Immersed Interface method (IIM) is a second-order accurate numerical method developed for systems with interface, *i.e.* solute/solvent boundary in this case.

In the IIM discretization scheme, the linear equations on regular grid points, i.e. grid points away from the interface, are the same as the standard finite-difference method, but the linear equations on irregular grid points, i.e. grid points nearby the interface, are constructed by minimizing the magnitude of the local truncation error in the discretization of the PB equation.[145] It can be proven that the errors of calculated potentials are at the order of $O(h^2)$ on the regular grid points and $O(h)$ on the irregular grid points.[145]

8.1.2. Numerical interpretation of energy and forces

PB solvents approximate the solvent-induced electrostatic mean-force potential by computing the reversible work in the process of charging the atomic charges in a solute molecule or complex. The charging free energy is a function of the electrostatic potential ϕ , which can be computed by solving the linear or nonlinear system.

It has been shown (see for example [134]) that the total electrostatic energy of a solute molecule can be approximated through the FD approach by subtracting the self FD Coulombic energy ($G_{coul,shelf}^{FD}$) and the short-range FD Coulombic energy ($G_{coul,short}^{FD}$) from the total FD electrostatic energy ($G_{coul,total}^{FD}$), and adding back the analytical short-range Coulombic energy ($G_{coul,short}^{ana}$). The self FD Coulombic energy is due to interactions of grid charges within one single atom. The self energy exists even when the atomic charge is exactly positioned on one grid point. It also exists in the absence of solvent and any other charges. It apparently is a pure artifact of the FD approach and must be removed. The short-range FD Coulombic energy is due to interactions between grid charges in two different atoms that are separated by a short distance, usually less than 14 grid units. The short-range Coulombic energy is inaccurate because the atomic charges are mapped onto their eight nearest FD grids, thus causing deviation from the analytical Coulomb energy. The correction of $G_{coul,shelf}^{FD}$ and $G_{coul,short}^{FD}$ is made possible by the work of Luty and McCammon's analytical approach to compute FD Coulombic interactions.[146]

Therefore, the PB electrostatic interactions include both Coulombic interactions and reaction field interactions for all atoms of the solute. The total electrostatic energy is given in the energy component EEL in the output file. The term that is reserved for the reaction field energy, EPB, is zero if this method is used. If you want to know how much of EEL is the reaction field energy, you can set the BCOPT keyword (to be explained below) to compute the reaction field energy only by using a Coulombic field (or singularity) free formulation.[147]

When the full nonlinear Poisson-Boltzmann equation is used, an additional energy term, the ionic energy, should also be included. This energy term disappears in the symmetrical linear system because the effects due to opposite ions cancel out. It is currently approximated by calculation up to the space boundary of the FD grid. It should be noted that the NBUFFER keyword may need increasing to obtain good precision in the ionic energy for small molecules with a large FILLRATIO.

An alternative method of computing the electrostatic interactions is also implemented in *pbsa*. In this method, the reaction field energy is computed directly after the induced surface charges are first computed at the dielectric boundary (i.e., the surface that separates solute and solvent). These surface charges are then used to compute the reaction field energy,[133] and is given as the EPB term. It has been shown that doing so improves the convergence of reaction field energy with respect to the FD grid spacing. However, a limitation of this method is that

8. PBSA

the Coulombic energy has to be recomputed analytically with a pairwise summation procedure. When this method is used, the EEL term only gives the Coulombic energy with a cutoff distance provided in the input file. The two ways of computing electrostatic interactions are controlled by the keywords ENEOPT and FRCOPT to be described below.

The non-polar solvation free energy is returned by the ECAVITY term, which is either the total non-polar solvation free energy or the cavity solvation free energy in the two different models described above. The EDISPER term returns the dispersion solvation free energy. Of course it is zero if the total non-polar solvation free energy has been returned by ECAVITY. The word INP can be used to choose one of the two treatments of non-polar solvation interactions.[90] Specifically, you can use SASA to correlate total non-polar solvation free energy, i.e., $G_{np} = NP_TENSION * SASA + NP_OFFSET$ as in PARSE.[138] You can also use SASA to correlate the cavity term only and use a surface-integration approach to compute the dispersion term.[90] i.e., $G_{np} = G_{disp} + G_{cavity}$, with $G_{cavity} = CAVITY_TENSION * SASA + CAVITY_OFFSET$. When this option is used, RADIOPT has to be set to 1, i.e., the radii set optimized by Tan and Luo to mimic G_{np} in the TIP3P explicit solvent.[90] Otherwise, there is no guarantee of consistency between the implemented non-polar implicit solvent and the TIP3P explicit solvent. See the discussion of keywords in 8.2.8. These options are described in detail in Ref. [90].

Finally, in this release, the PB forces are now correctly interpreted for the widely used SES molecular surface definition, i.e., the partition of dielectric boundary pressure/force can now reproduce the virtual work principle. This is achieved by proper decomposition of the dielectric boundary force on the reentrant portion of the molecular surface. Specifically, the molecular surface is computed more accurately by considering the cases when the solvent probe touches three atoms simultaneously. Next the reentrant force is also distributed onto the three atoms forming the reentrant surface following the virtual work principle.[148]

8.1.3. Numerical accuracy and related issues

Note that the accuracy of any numerical PB procedure is determined by the discretization resolution specified in the input, i.e., the grid spacing. The convergence criterion for the iteration procedures also plays some role for the numerical PB solvers. Finally the accuracy is highly dependent upon the methods used for computing total electrostatic interactions. In Lu and Luo,[134] the accuracy of the first method for total electrostatic interactions is discussed in detail. In Ye and Luo [Manuscript in preparation], the accuracy of the second method is discussed.

It is recommended that the second method for total electrostatic interactions be used for most calculations. Apparently the cutoff distance for charge-charge interactions strongly influences the accuracy of electrostatic interactions. The default setting is infinity, i.e., no cutoff is used. In this method, the convergence of the reaction field energy with respect to the grid spacing is much better than that of the first method. Our experience shows that the reaction field energies converge to within ~2% for tested proteins at the grid spacing of 0.5 Å when the weighted harmonic average of dielectric constants is used at the solute/solvent interface (when SMOOTHOPT = 1, see below).[149]

The reaction field energies computed with the second method (when SMOOTHOPT = 2) are also in excellent agreement (differences in the order of 0.1%) with those computed with the *Delphi* program which uses the same method for energy calculation. For example, see

the computational set up documented in test case *pbsa_delphi* in this release [Wang and Luo, Manuscript in preparation].

The accuracy of non-polar solvation energy depends on the quality of SASA which is computed numerically by representing each atomic surface by spherically distributed dots. Thus a higher dot density gives more accurate atomic surface and molecular surface. However, it is found that the default setting for the dot density is quite sufficient for typical applications.[90] Should you encounter any memory allocation error for surface calculation, you are advised to enlarge the surface dot resolution if the physical memory of your computer is limited.

Numerical solvation calculations are memory intensive for macromolecules due to the fine grid resolution required for sufficient accuracy. Thus, the efficiency of *pbsa* depends on how much memory is allocated for it and the performance of the memory subsystem. The option that is directly related to its memory allocation is the FD grid spacing for the PB equation and the surface dot resolution for molecular surface. Apparently the geometric dimension and the number of atoms are also important for predicting the memory usage. In general for a typical computer configuration with 8GB memory, the geometric dimension can be as large as $180 \times 180 \times 180 \text{ \AA}^3$ at the default grid spacing of 0.5 \AA before the computer responds too slowly.

8.2. Usage and keywords

8.2.1. File usage

pbsa has a very similar user interface as the *Amber/sander* program, though much simpler.

```
pbsa [-O] -i mdin -o mdout -p prmtop -c inpcrd
```

Here is a brief description of the files referred to above.

mdin *input* control data for the run.

mdout *output* user readable state info and diagnostics “-o stdout” will send output to stdout (to the terminal) instead of to a file.

prmtop *input* molecular topology, force field, atom and residue names, and (optionally) periodic box type.

inpcrd *input* initial coordinates and (optionally) velocities and periodic box size.

8.2.2. Basic input options

The layout of the input file is in the same way as that of *Amber/sander* for backward compatibility with previous releases in *Amber*. The keywords are put in the the namelist of &cntrl for basic controls and &pb for more detailed manipulation of the numerical procedures. This subsection discusses the basic keywords, either retained from *sander* or newly created to invoke different energetic analyses. To reduce confusion most keywords from *sander* have been removed from the namelist so they can no longer be read since the current implementation in *pbsa* only performs single-structure calculations with the coordinates from **inpcrd** and exits.

8. PBSA

However, the current release is compatible with the **mdin** file generated with the *mmpbsa* script in previous releases in *Amber*. Users interested in energy minimization and molecular dynamics with the PB implementation are referred to *sander* in the release of *Amber*. Nevertheless, for purposes of validation and development, the atomic forces can be dumped out in a file when requested as described below.

The numerical electrostatic procedures can be turned on by setting IPB to either 1, 2 or 4. The flag IGB = 10 is phased out in this release. The numerical non-polar procedures can be turned on by setting INP to either 1 or 2. The backward compatible flag NPOPT is also phased out in this release.

imin Flag to run minimization. Both options give the same output energies though the output formats are slightly different. This option is retained from previous releases in the *Amber* package for backward compatibility. The current release of *pbsa* only supports single point energy calculation.

= 0 No minimization. Dynamics is available with *sander* and NAB.

= 1 Single point energy calculation. Default. Multiple-step PB minimization is also available with *sander* and NAB.

ntx Option to read the coordinates from the “inpcrd” file. Only options 1 and 2 are supported in this releases. Other options will cause *pbsa* to issue a warning though it does not affect the energy calculation.

= 1 X is read formatted with no initial velocity information. Default.

= 2 X is read unformatted with no initial velocity information.

ipb Option to set up a dielectric model for all numerical PB procedures. IPB = 1 corresponds to a classical geometric method, while a level-set based algebraic method is used when IPB \geq 2. The default IPB is 2.

= 0 No electrostatic solvation free energy is computed.

= 1 The dielectric interface between solvent and solute is built with a geometric approach.

= 2 The dielectric interface is implemented with the level set function. Use of a level set function simplifies the calculation of the intersection points of the molecular surface and grid edges and leads to more stable numerical calculations. Default.

= 4 The dielectric interface is also implemented with the level set function. However, the linear equations on the irregular points are constructed using the IIM. In this option, The dielectric constant do not need to be smoothed, that is, SMOOTHOPT is useless. Only the linear PB equation is supported, that is, NPBOPT = 0. And the different solvers are used to solve the generated linear equation set, that is, the meaning of SOLVOPT is changed as shown below.

inp Option to select different methods to compute non-polar solvation free energy.

- = 0** No non-polar solvation free energy is computed.
- = 1** The total non-polar solvation free energy is modeled as a single term linearly proportional to the solvent accessible surface area, as in the PARSE parameter set, that is, if INP = 1, USE_SAV must be equal to 0. See Introduction.
- = 2** The total non-polar solvation free energy is modeled as two terms: the cavity term and the dispersion term. The dispersion term is computed with a surface-based integration method [90] closely related to the PCM solvent for quantum chemical programs.[140] Under this framework, the cavity term is still computed as a term linearly proportional to the molecular solvent-accessible-surface area (SASA) or the molecular volume enclosed by SASA. With this option, please do not use RADIOPT = 0, i.e., the radii in the prmtop file. Otherwise, a warning will be issued in the output file. Default.

Once the above basic options are specified, *pbsa* can proceed with the default options to compute the solvation free energies with the input coordinates. Of course, this means that you only want to use default options for default applications.

More PB options described below can be defined in the &pb namelist, which is read immediately after the &cntrl namelist. We have tried hard to make the defaults for these parameters appropriate for calculations of solvated molecular systems. Please use caution when changing any default options.

8.2.3. Options to define the physical constants

- | | |
|-----------|---|
| epsin | Sets the dielectric constant of the solute region, default to 1.0. The solute region is defined to be the solvent excluded volume. |
| epsout | Sets the implicit solvent dielectric constant, default to 80. The solvent region is defined to be the space not occupied by the solute region. i.e., only two dielectric regions are allowed in the current release. |
| smoothopt | <p>Instructs PB how to set up dielectric values for finite-difference grid edges that are located across the solute/solvent dielectric boundary.</p> <ul style="list-style-type: none"> = 0 The dielectric constants of the boundary grid edges are always set to the equal-weight harmonic average of EPSIN and EPSOUT. = 1 A weighted harmonic average of EPSIN and EPSOUT is used for boundary grid edges. The weights for EPSIN and EPSOUT are fractions of the boundary grid edges that are inside or outside the solute surface.[150] Default. = 2 The dielectric constants of the boundary grid edges are set to either EPSIN or EPSOUT depending on whether the midpoints of the grid edges are inside or outside the solute surface. |
| istrng | Sets the ionic strength (in mM) for the PB equation. Default is 0 mM. Note the unit is different from that (in M) in the generalized Born methods implemented in <i>Amber</i> . Note also that we are only dealing with symmetrical solution, so the ionic |

8. PBSA

strength should be equal to the square of the valence of the symmetrical ions times the ion concentration (in mM).

pbtemp	Temperature (in K) used for the PB equation, needed to compute the Boltzmann factor for salt effects; default is 300 K.
radiopt	<p>Option to set up atomic radii.</p> <p>= 0 Use radii from the prmtop file for both the PB calculation and for the NP calculation (see INP).</p> <p>= 1 Use atom-type/charge-based radii by Tan and Luo [151] for the PB calculation. Note that the radii are optimized for <i>Amber</i> atom types as in standard residues from the <i>Amber</i> database. If a residue is built by <i>antechamber</i>, i.e., if GAFF atom types are used, radii from the prmtop file will be used. Please see [151] on how these radii are optimized. The procedure in [151] can also be used to optimize radii for nonstandard residues. These optimized radii can be read in if they are incorporated into the radii section of the prmtop file (of course via RADIOPT = 0). This option also instructs <i>pbsa</i> to use van der Waals radii from the prmtop file for non-polar solvation energy calculations. These van der Waals radii are really the half sigma/rmin values for pairs of atoms of the same types. So they are computed from the van der Waals coefficients in the prmtop file. (see INP and USE_RMIN). Default.</p>
dprob	Solvent probe radius for molecular surface used to define the dielectric boundary between solute and solvent. DPROB = 1.4 by default.
iprob	Mobile ion probe radius for ion accessible surface used to define the Stern layer. Default to 2.0 Å.
sasopt	<p>Option to determine which kind of molecular surfaces to be used in the Poisson-Boltzmann implicit solvent model. Default is 0.</p> <p>= 0 Use the solvent excluded surface. [Wang, Cai, and Luo, Manuscript in preparation]</p> <p>= 1 Use the solvent accessible surface.</p> <p>= 2 Use the smooth surface defined by a revised density function.[152] This must be combined with IPB \geq 2.</p>
saopt	<p>Option to compute the surface area of a molecule. Default is 0. Once the computation is enabled, the surface area will be reported in the output file with the subtitle “Total molecular surface”. Note that only the surface areas for the solvent excluded surface and the solvent accessible surface are supported in this release.</p> <p>= 0 Do not compute any surface area.</p> <p>= 1 Use the field-view method to compute the surface area.[153]</p>
triopt	Option to add trimer arc dots for a more accurate and lower memory mapping method of the analytical solvent excluded surface.

= 0 Trimer arc dots are not used.

= 1 Trimer arc dots are used. Default.

arcres *pbsa* uses a numerical method to compute solvent accessible arcs,[Wang, Cai, and Luo, Manuscript in preparation]. The ARCREs keyword gives the resolution (in the unit of Å) of dots used to represent these arcs, default to 0.25 Å. These dots are first checked against nearby atoms to see whether any of the dots are buried. The exposed dots represent the solvent accessible portion of the arcs and are used to define the dielectric constants on the grid edges. It should be pointed out that ARCREs should be reduced to (0.125 Å) when the TRIOPT option is turned off to achieve a similar accuracy in the reaction field energies. More generally, ARCREs should be set to *max*(0.125 Å, 0.5*h*) when the TRIOPT option is turned on, or *max*(0.0625 Å, 0.25*h*) when the TRIOPT option is turned off (*h* is the grid spacing).[Wang, Cai, and Luo, Manuscript in preparation]

8.2.4. Options for Implicit Membranes

membraneopt Option to turn implicit membrane on and off. Membrane is implemented as a slab like region with same dielectric constant as solute. Other membrane setup schemes will be made available in the future.

= 0 No implicit membrane used (default).

= 1 Use a slab-like implicit membrane.

mthick Membrane thickness in Å, default to 20.0.

mctrdz Distance in Å to offset membrane along the z direction. Default is 0 - membrane centered at the center of the finite difference grid.

poretype Option to control use of exclusion region for channel proteins. Only cylindrical region is supported currently.

= 0 Do not use a cylindrical exclusion region (Default).

= 1 Use cylindrical exclusion region.

poreradius Controls the radius, in Å, of the cylindrical exclusion region.

8.2.5. Options to select numerical procedures

npbopt Option to select the linear or the full nonlinear PB equation.

= 0 Linear PB equation is solved. Default.

= 1 Nonlinear PB equation is solved.

solvopt Option to select iterative solvers.

= 1 Modified ICCG. Default. If IPB = 4, an algebraic multigrid solver is used.

8. PBSA

	<ul style="list-style-type: none">= 2 Geometric multigrid. A four-level v-cycle implementation is applied. Each dimension of the finite-difference grid is $2^4 \times n-1$. If IPB = 4, preconditioned GMRES.= 3 Conjugate gradient. This option requires a large MAXITN to converge. If IPB = 4, preconditioned BiCG.= 4 SOR. This option requires a large MAXITN to converge.= 5 Adaptive SOR. This is only compatible with NPBOPT = 1. This option requires a large MAXITN converge.= 6 Damped SOR. This is only compatible with NPBOPT = 1. This option requires a large MAXITN to converge.
accept	Sets the iteration convergence criterion (relative to the initial residue). Default to 0.001.
maxitn	Sets the maximum number of iterations for the finite difference solvers, default to 100. Note that MAXITN has to be set to a much larger value, like 10,000, for the less efficient solvers, such as conjugate gradient and SOR, to converge.
fillratio	The ratio between the longest dimension of the rectangular finite-difference grid and that of the solute. Default is 2.0. It is suggested that a larger FILLRATIO, for example 4.0, be used for a small solute, such as a ligand molecule. Otherwise, part of the small solute may lie outside of the finite-difference grid, causing the finite-difference solvers to fail.
space	Sets the grid spacing for the finite difference solver; default is 0.5 Å.
nbuffer	Sets how far away (in grid units) the boundary of the finite difference grid is away from the solute surface; default is 0 grids, i.e., automatically set to be at least a solvent probe or ion probe (diameter) away from the solute surface.
nfocus	Set how many successive FD calculations will be used to perform an electrostatic focussing calculation on a molecule. Default to 2, the maximum. When NFOCUS = 1, no focusing is used. It is recommended that NFOCUS = 1 when the multigrid solver is used.
fscale	Set the ratio between the coarse and fine grid spacings in an electrostatic focussing calculation. Default to 8.
npbgrid	Sets how often the finite-difference grid is regenerated; default is 1 step. For molecular dynamics simulations, it is recommended to be set to at least 100. Note that the PB solver effectively takes advantage of the fact that the electrostatic potential distribution varies very slowly during dynamics simulations. This requires that the finite-difference grid be fixed in space for the code to be efficient. However, molecules do move freely in simulations. Thus, it is necessary to set up the finite-difference grid once in a while to make sure a molecule is well within the grid.

8.2.6. Options to compute energy and forces

ENEOPT is the option to set a method to compute electrostatic energy and forces, and DBFOPT is phased out in this release.

bcopt Boundary condition options.

- = 1** Boundary grid potentials are set as zero. Total electrostatic potentials and energy are computed.
- = 5** Computation of boundary grid potentials using all grid charges. Total electrostatic potentials and energy are computed. Default.
- = 6** Computation of boundary grid potentials using all grid charges. Reaction field potentials and energy are computed with the charge singularity free formulism.[\[147\]](#)
- = 10** Periodic boundary condition is used. Total electrostatic potentials and energy are computed.

eneopt Option to compute total electrostatic energy and forces.

- = 1** Compute total electrostatic energy and forces with the particle-particle particle-mesh (P3M) procedure outlined in Lu and Luo.[\[134\]](#) In doing so, energy term EPB in the output file is set to zero, while EEL includes both the reaction field energy and the Coulombic energy. The van der Waals energy is computed along with the particle-particle portion of the Coulombic energy. The electrostatic forces and dielectric boundary forces can also be computed.[\[134\]](#) This option requires a non-zero CUTNB and BCOPT = 5.
- = 2** Use dielectric boundary surface charges to compute the reaction field energy. Default. Both the Coulombic energy and the van der Waals energy are computed via summation of pairwise atomic interactions. Energy term EPB in the output file is the reaction field energy. EEL is the Coulombic energy.
- = 3** Similar to the first option above, a P3M procedure is applied for both solvation and Coulombic energy and forces for larger systems.

frcopt Option to compute and output electrostatic forces to a file named *force.dat* in the working directory.

- = 0** Do not compute or output atomic and total electrostatic forces. This is default.
- = 1** Reaction field forces are computed by trilinear interpolation. Dielectric boundary forces are computed using the electric field on dielectric boundary. The forces are output in the unit of kcal/mol·Å.
- = 2** Use dielectric boundary surface polarized charges to compute the reaction field forces and dielectric boundary forces [\[148\]](#) The forces are output in the unit of kcal/mol·Å.
- = 3** Reaction field forces are computed using dielectric boundary polarized charge. Dielectric boundary forces are computed using the electric field on dielectric

8. PBSA

boundary [Ye *et al.* Manuscript submitted]. The forces are output in the unit of kcal/mol-Å.

scalec	Option to compute reaction field energy and forces. = 0 Do not scale dielectric boundary surface charges before computing reaction field energy and forces. Default. = 1 Scale dielectric boundary surface charges using Gauss's law before computing reaction field energy and forces.
cutfd	Atom-based cutoff distance to remove short-range finite-difference interactions, and to add pairwise charge-based interactions, default is 5 Å. This is used for both energy and force calculations. See Eqn (20) in Lu and Luo.[134]
cutnb	Atom-based cutoff distance for van der Waals interactions, and pairwise Coulombic interactions when ENEOPT = 2. Default to 0. When CUTNB is set to the default value of 0, no cutoff will be used for van der Waals and Coulombic interactions, i.e., all pairwise interactions will be included. When ENEOPT = 1, this is the cutoff distance used for van der Waals interactions only. The particle-particle portion of the Coulombic interactions is computed with the cutoff of CUTFD.
nsnba	Sets how often atom-based pairlist is generated; default is 1 step. For molecular dynamics simulations, a value of 5 is recommended.

8.2.7. Options for visualization and output

phiout	<i>pbsa</i> can be used to output spatial distribution of electrostatic potential for visualization. = 0 No potential file is printed out. Default. = 1 Electrostatic potential is printed out in a file named <i>pbsa.phi</i> in the working directory. Please refer to examples in the next section on how to display electrostatic potential on molecular surface.
phiform	Controls the format of the electrostatic potential file. = 0 The electrostatic potential (kT/mol- <i>e</i>) is printed in the <i>Delphi</i> binary format. Default. = 1 The electrostatic potential (kcal/mol- <i>e</i>) is printed in the <i>Amber</i> ASCII format. = 2 The electrostatic potential (kcal/mol- <i>e</i>) is printed in the DX volumetric data format for use with <i>VMD</i> .
outlvlset	<i>pbsa</i> can be set to write the total level set, used in locating interfaces between regions of differing dielectric constant, to a DX format volumetric data file. This option will control printing of the total level set (i.e. both solute-solvent and membrane level sets combined if membrane present) = false No level set file printed out. Default.

- = true** Level set printed out in a file named `pbsa_lvlset.dx`
- `outmlvlset` *pbsa* can be set to write the membrane level set, used in locating interfaces between regions of differing dielectric constant, to a DX format volumetric data file. This option controls printing a separate file for the membrane level set. Does nothing if `membraneopt` is not turned on.
- = false** No level set file printed out. Default.
- = true** Level set printed out in a file named `pbsa_lvlset.dx`
- `npbverb` When set to 1, turns on verbose mode in *pbsa*; default is 0.

8.2.8. Options to select a non-polar solvation treatment

- `decompopt` Option to select different decomposition schemes when `INP = 2`. See [90] for a detailed discussion of the different schemes. The default is 2, the σ decomposition scheme, which is the best of the three schemes studied.[90] As discussed in Ref. [90], `DECOMPOPT = 1` is not a very accurate approach even if it is more straightforward to understand the decomposition.
- = 1** The 6/12 decomposition scheme.
- = 2** The σ decomposition scheme. Default
- = 3** The WCA decomposition scheme.
- `use_rmin` The option to set up van der Waals radii. The default is to use *rmin* to improve the agreement with TIP3P [90].
- = 0** Use atomic van der Waals σ values.
- = 1** Use atomic van der Waals *rmin* values. Default.
- `sprob` Solvent probe radius for solvent accessible surface area (SASA) used to compute the dispersion term, default to 0.557 Å in the σ decomposition scheme as optimized in Ref. [90] with respect to the TIP3P solvent and the PME treatment. Recommended values for other decomposition schemes can be found in Table 4 of [90]. If `USE_SAV = 0` (see below), `SPROB` can be used to compute SASA for the cavity term as well. Unfortunately, the recommended value is different from that used in the dispersion term calculation as documented in Ref. [90] Thus two separate *pbsa* calculations are needed when `USE_SAV = 0`, one for the dispersion term and one for the cavity term. Therefore, please carefully read Ref. [90] before proceeding with the option of `USE_SAV = 0`. Note that `SPROB` was used for ALL three terms of solvation free energies, i.e., electrostatic, attractive, and repulsive terms in previous releases in *Amber*. However, it was found in the more recent study [90] that it was impossible to use the same probe radii for all three terms after each term was calibrated and validated with respect to the TIP3P solvent. [90, 151]
- `vprob` Solvent probe radius for molecular volume (the volume enclosed by SASA) used to compute non-polar cavity solvation free energy, default to 1.300 Å, the value

8. PBSA

optimized in Ref. [90] with respect to the TIP3P solvent. Recommended values for other decomposition schemes can be found in Tables 1-3 of Ref. [90].

rho_w_effect Effective water density used in the non-polar dispersion term calculation, default to 1.129 for DECOMPOPT = 2, the σ scheme. This was optimized in Ref. [90] with respect to the TIP3P solvent in PME. Optimized values for other decomposition schemes can be found in Table 4 of Ref. [90].

use_sav The option to use molecular volume (the volume enclosed by SASA) or to use molecular surface (SASA) for cavity term calculation. The default is to use the molecular volume enclosed by SASA. Recent study shows that the molecular volume approach transfers better from small training molecules to biomacromolecules.

= 0 Use SASA to estimate cavity free energy.

= 1 Use the molecular volume enclosed by SASA. Default.

cavity_surften The regression coefficient for the linear relation between the total non-polar solvation free energy (INP = 1) or the cavity free energy (INP = 2) and SASA/volume enclosed by SASA. The default value is for INP = 2 and set to the best of three tested schemes as reported in Ref. [90], i.e. DECOMPOPT = 2, USE_RMIN = 1, and USE_SAV = 1. See recommended values in Tables 1-3 for other schemes.

cavity_offset The regression offset for the linear relation between the total non-polar solvation free energy (INP = 1) or the cavity free energy (INP = 2) and SASA/volume enclosed by SASA. The default value is for INP = 2 and set to the best of three tested schemes as reported in Ref. [90], i.e. DECOMPOPT = 2, USE_RMIN = 1, and USE_SAV = 1. See recommended values in Tables 1-3 for other schemes.

maxsph *pbsa* uses a numerical method to compute solvent accessible surface area.[90] MAXSPH variable gives the approximate number of dots to represent the maximum atomic solvent accessible surface, default to 400. These dots are first checked against covalently bonded atoms to see whether any of the dots are buried. The exposed dots from the first step are then checked against a non-bonded pair list with a cutoff distance of 9 to see whether any of the exposed dots from the first step are buried. The exposed dots of each atom after the second step then represent the solvent accessible portion of the atom and are used to compute the SASA of the atom. The molecular SASA is simply a summation of the atomic SASA's. A molecular SASA is used for both PB dielectric map assignment and for NP calculations.

8.2.9. Options to enable active site focusing

Active site focusing is an extension to the electrostatic focusing method. Electrostatic focusing can be regarded as a multi-level FDPB calculation (two levels currently implemented) in which a coarse-grid solution is conducted to set up the boundary condition for the requested fine-grid solution. In the original implementation of electrostatic focusing, the fine grid always covers all the solute atoms. However in the enhanced implementation, the fine grid is allowed to

cover only a local region of interest, such as an enzyme active site or ligand docking site. In such applications, most or all of the protein atoms are held frozen during a calculation while only the active site side chain and the substrate ligand are allowed to move. In principle, energies computed with the local electrostatic focusing method should correlate with those computed with the original electrostatic focusing method if the movable substrate/ligand atoms are well within the local region of interest. The “active site” or the local region is specified as a rectangular box by the following six variables:

xmax	The upper boundary of the box in x direction.
xmin	The lower boundary of the box in x direction, XMAX has to be greater than XMIN.
ymax	The upper boundary of the box in y direction.
ymin	The lower boundary of the box in y direction, YMAX has to be greater than YMIN.
zmax	The upper boundary of the box in z direction.
zmin	The lower boundary of the box in z direction, ZMAX has to be greater than ZMIN.

Of course, these keywords are zero by default, i.e. the original electrostatic focusing would be invoked if these keywords remain to be the default value of zero.

8.2.10. Options to enable multiblock focusing

In order to handle large molecular systems with typical computer hardwares available to our end users, the basic principle of the electrostatic focusing discussed in the previous subsection is extended for the multiblock electrostatic focusing method. Briefly, the time-limiting step of FDPB, the fine-grid calculation, is divided into a series of smaller jobs, with each solving only a small local region of a large molecular system. Once all the smaller jobs are finished, the solutions are combined to obtain the final energy for the large molecular system. Note that this is an approximated method, just like the original electrostatic focusing method. In this implementation, overlapping/padding grid points are used to preserve accuracy. Most of the settings for this feature are hidden from end users except the dimensions of the multi-blocks. [154]

Before your production runs, please activate NPBVERB = 1 and check in the mdout file to see if your multi-block settings are indeed reasonable. Here are some hints. First, the blocksize should be around 64^3 to 96^3 for typical computers with 8GB memory. Secondly, the grid dimension, xm , should be divisible by $(ngrdblks - 1)$, or slightly larger, for the x direction. The same applies for y and z directions as well. Keep in mind that the incentive for choosing this method is to be able to work with large systems on typical computer hardwares.

ngrdblks	The number of fine-grid points for a focusing block in x direction, $(ngrdblks - 1)$ should be divisible by FSCALE.
ngrdblky	The number of fine-grid points for a focusing block in y direction, $(ngrdblky - 1)$ should be divisible by FSCALE.

8. PBSA

`ngrdblz` The number of fine-grid points for a focusing block in z direction, (*ngrdblz* − 1) should be divisible by FSCALE.

pbsa can also be run in parallel environment with *pbsa.MPI* executable but for multiblock focusing only. Do make sure that the number of nodes is less than the number of focusing blocks.

8.3. Example inputs and demonstrations of functionalities

8.3.1. Single-point calculation of solvation free energies

Normally the default *pbsa* options are capable of dealing with most situations. Users should be fully aware of the meaning of an option before they change its default value. In all the following example inputs, only the options that are different from their default values will be shown, and the explanations on the changes will be given in detail. Here is a sample input file that might be used to perform single structure calculations.

```
Sample single point PB calculation
&cntrl
/
&pb
npbverb=1, istrng=150, fillratio=1.5, saopt=1,
/
```

Note that NPBVERB = 1 above. This generates much detailed information in the output file for the PB and NP calculations. A useful printout is atomic SASA data for both PB and NP calculations which may or may not use the same atomic radius definition. Since the FD solver for PB is called twice to perform electrostatic focus calculations, two PB printouts are shown for each single point calculation. For the PB calculation, a common error message can be generated when FILLRATIO is set to the default value of 2.0 for small molecules. This may cause a solute to lie outside of the focusing finite-difference grid.

In this example INP is not set and equal to the default value of 2, which calls for non-polar solvation calculation with the new method that separates cavity and dispersion interactions. The EDISPER term gives the dispersion solvation free energy, and the ECAVITY term gives the cavity solvation free energy. The default options for the NP calculation are set to the recommended values for the σ decomposition scheme and to use molecular volume to correlate with cavity free energy. You can find recommended values for other decomposition schemes and other options in Tables 1-4 of Ref. [90]. If INP is set to 1, the ECAVITY term would give the total non-polar solvation free energy.

The ion strength option ISTRNG is set to 150 in unit mM, a typical value for a physiological environment. The FILLRATIO option is set to 1.5 because the biomolecule is relatively large. We set saopt to 1 because we need the information of the molecular surface area (the molecular surface is defined as the solvent excluded surface since SASOPT is set to its default value 0) .

8.3.2. Implicit membrane model

pbsa now supports inclusion of an implicit membrane region in implicit solvation calculations. This feature is enabled by setting MEMBRANEOPT to 1 (default value is 0, for off). The membrane will extend the solute dielectric region to include a slab-like planar region running parallel to the xy plane. The thickness is controlled by the MTHICK option. The default is 20 Å. The membrane region will be centered on the center of the finite-difference grid by default, and can be offset along the z-axis using the MCTRDZ option (default is 0). Neither option will have any effect unless MEMBRANEOPT is set to 1. Currently, setting a unique dielectric constant for the membrane region is not supported but should be available in the next release.

When using the implicit membrane model, only SASOPT = 2, i.e. the smooth molecular surface based on the revised density function, is currently supported. It is also suggested that the periodic conjugate gradient solver is used to avoid any unphysical edge effect. This is accomplished by setting IPB = 2 (default), BCOPT = 10, and SOLVOPT = 3. Besides, ENEOPT needs to be set to 1 because the charge-view method (ENEOPT = 2) has not been verified for this application.

```
Sample single point PB calculation with membrane region
&cntrl
inp=0
/
&pb
radiopt=0, nfocus=1, maxitn=10000,
bcopt=10, eneopt=1, solvopt=3,
sasopt=2, membraneopt=1,
outlvlset=true, outmlvlset=true
/
```

The MAXITN option is set to a much bigger value, 10000, than the default one, 100, because the conjugate gradient solver converges slowly. To aid in visualization of the dielectric model, the level set function, which is used to locate the interfacial surfaces between regions of differing dielectric constant, can be written to output files. Output of the total level set function, including both the solute-solvent and membrane contributions, can be written to a DX formatted volumetric data file by setting the OUTLVLSET option to “true”. The membrane contribution can be written to a separate file by setting the OUTMLVLSET option to “true”. Accordingly, NFOCUS is set to 1 because we want the electrostatic potential and the level set function in both the solute and the solvent region.

8.3.3. Single point calculation of forces

Since *pbsa* is released for single point calculations in *AmberTools*, no energy minimization or molecular dynamics is supported. However, the PB procedure can be invoked to print out the numerical electrostatic forces for developmental purposes. Here is a sample input:

```
Sample PB force computation
&cntrl
```

8. PBSA

```
inp=0
/  
&pb  
npbverb=1, radiopt=0, frcopt=2  
/
```

Note that INP is set to 0 to turn off non-polar solvation interactions. RADIOPT = 0 means the atomic radii from the topology files will be used. FRCOPT is set to 2, *i.e.*, induced surface charges are used to compute the electrostatic energy and forces. Since CUTNB is equal to the default value of zero, an infinite cutoff distance is used for both Coulombic and van der Waals interactions.

8.3.4. Comparing with *Delphi* results

Under identical condition, *pbsa* is highly consistent with *Delphi* in term of computed reaction field energies. In this subsection, we briefly go over the details on how you can obtain comparable energies from both programs. Apparently, you need coordinates, atomic charges, and atomic radii that have exactly the same numerical values but in both the *Amber* format and the *Delphi* format, *i.e.*, the pqr format.

For a *Delphi* computation with the following input parameters:

```
salt=0.150  
ionrad=2.0  
exdi=80.0  
indi=1.0  
scale=2.0  
prbrad=1.5  
perfil=50  
bndcon=4  
linit=1000
```

A comparable computation in *pbsa* can be obtained by using the following input file:

```
Sample PB for delphi comparison  
&cntrl  
ipb=1, inp=0  
/  
&pb  
istrng=150, ivalence=1, iprob=2.0, dprob=1.5,  
radiopt=0, bcopt=5, smoothopt=2, nfocus=1,  
/
```

IPB is set to 1 to make sure *pbsa* is using the exactly same surface definition as *Delphi*. Note that the values of *exdi*, *indi*, *prbrad*, and *ionrad* in *Delphi* should be consistent with the values of EPSOUT, EPSIN, DPROB, and IPROB in *pbsa*, respectively. In *Delphi* *salt*=0.150 is set in the unit of M, while in *pbsa* ISTRNG = 150 is in the unit of mM. In *Delphi* the grid spacing is set as the number of grids per Å, *i.e.*, *scale*=2.0, while in *pbsa* the grid spacing is set straight in

Å as `SPACE = 0.5`. In *Delphi* the grid dimension is set as percentage of the solute dimension over the grid dimension, i.e., `perfil=50`, which is equivalent to the ratio of solute dimension over grid dimension set as `FILLRATIO = 2` in *pbsa*. Finally, *Delphi* sets the boundary condition by `bndcon=4` and *pbsa* sets the boundary condition as `BCOPT = 5`; both programs mean to use the Debye-Huckel limitation behavior for each atomic charged sphere. There are additional options in *pbsa* that do not have corresponding counterparts in *Delphi*. For example, `SMOOTHOPT` is used to instruct the program to use a specific dielectric boundary smoothing option, which is equivalent to that used in *Delphi* when set to 2. (see Section 8.1.3).

8.4. Visualization functions in *pbsa*

AMBER *pbsa* can produce volumetric data files to allow visualization of electrostatic potential and level set maps. There are two points to note before continuing.

1. The data files generated can become quite large if small grid spacings are used since they will scale as the cube of the inverse of grid spacing
2. Unless singularity removal methods are used, the potential at grid nodes corresponding to atom centers may be quite large when compared to the potential at the molecular / atomic surface. This will often result in poor contrast during visualization of the potential map, particularly when it is used as a color map for a molecular surface.

These two points should be kept in mind when determining grid spacing. For visualization purposes, a grid spacing of about one angstrom should provide good results. If finer spacing is needed, singularity removal (`BCOPT = 6`) can be used to prevent poor contrast that could result from the presence of singularities. Lastly, when using grid spacings of 0.5 Å or lower, the output files may become quite large (tens, or even hundreds of megabytes each) and may take a significant amount of time (up to several seconds each) to generate.

8.4.1. Visualization of electrostatic potential using *PyMol*

pbsa can produce an electrostatic potential map for visualization in *PyMol* when setting `PHIOUT = 1`. By default, *pbsa* outputs a file `pbsa.phi` in the *Delphi* binary format. The sample input file is listed below:

```
Sample PB visualization input
&cntrl
inp=0
/
&pb
npbverb=1, space=1.,
phiout=1, phiorm=0
/
```

To be consistent with the surface routine of *PyMol*, the option `PHIOUT = 1` instructs *pbsa* to use the radii as defined in *PyMol*. The finite-difference grid is also set to be cubic as in *Delphi*.

8. PBSA

The default DPROB value is equal to that used in *PyMol*, 1.4 Å. A large grid spacing, e.g. 1 Å or higher, is recommended for visualization purposes, as commented above.

Here is an example of loading the potential map in *PyMol*. First load the molecule in the form of prmtop and inpcrd. In our case we need to rename our prmtop file to molecule.top and inpcrd file to molecule.rst and load the molecule with commands

```
PyMol> load molecule.top
PyMol> load molecule.rst
```

The molecule will appear as an object “molecule”. Next display the surface of the molecule in the *PyMol* menu by clicking “S” and then select surface. Now import the potential map generated by *pbsa* with the command in *PyMol*

```
PyMol> load pbsa.phi
```

to create a value map object called “pbsa”. After this, create a value ramp called e_lvl from the potential map with the command

```
PyMol> ramp_new e_lvl, pbsa, [-7, 0, 7]
```

You can assign surface_color to the e_lvl ramp with the command

```
PyMol> set surface_color, e_lvl, molecule
```

This will display the surface with the color scale according to the potential. You can adjust the value scale, such as [-5, 0, 5], to change the color scale and use “rebuild” command to redraw the surface.

8.4.2. Writing electrostatic potential to DX format volumetric data file

To visualize the *pbsa* potential using *VMD*, you will need to set the output to DX format by changing PHIFORM = 0 to PHIFORM = 2.

```
Sample PB visualization input
&cntrl
inp=0
/
&pb
npbverb=1, space=1., sasopt=2,
phiout=1, phiorm=2
/
```

The program will now generate a file called pbsa_phi.dx. This format should be automatically recognized by *VMD*. It can be either loaded directly into your molecule or as a separate file.

8.4.3. Loading DX format electrostatic potential data in *VMD*

1. go to the “File” menu in the *VMD* Main window.

2. Select “New Molecule...”.
 - This will bring up the “Molecule File Browser” window
3. Click on the “Browse...” button in the “Molecule File Browser” window
4. Select the file “pbsa_phi.dx” that was generated by *pbsa* using the file selection dialogue that pops up.
 - The “Determine file type:” drop down menu should now read “DX”.
5. Click the “Load” button.

VMD will, by default, display the data with an isosurface representation.

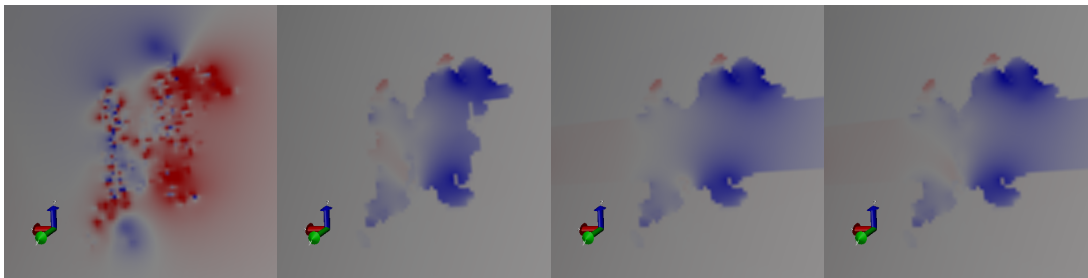
8.4.4. Changing the representation model

1. Select “Representations...” from the “Graphics” menu in the “*VMD* Main” window
 - The “Graphical Representations” window should pop up
2. Select the object corresponding to the volumetric data you loaded from the “Selected Molecule” pull down menu
3. Click on the representation you wish to change
 - There should be one present for the isosurface being displayed
4. Click on the “Draw style” tab if it is not already selected
5. Select “Volume” from the “Coloring Method” pull down menu if it is not already chosen
 - Another pull down menu will appear next to it.
 - If you have multiple data files loaded for the same object you can choose which is used to color your chosen draw method representation here
6. The “Drawing Method” pull down menu will let you choose a different visual representation model.
 - To directly visualize potential data, use either “Isosurface” or “Volume Slice”
 - *VMD* can also be used to visualize the corresponding electric field by choosing “Field Lines”.

Displayed below are Volume Slice representations of electrostatic potential maps generated for an aquaporin system. Computations were run using the periodic conjugate gradient solver for a 1 Å grid spacing, and FILLRATIO of 2.0. For the systems using implicit water, the charge singularity removal methodology was used.

From Left to right: Vacuum, Water only, Water and 20 Å slab-like membrane, Water and 20 Å slab-like membrane with 6 Å cylindrical channel region removed.

8. PBSA



Often, the data ranges will not be consistent between potential distributions for different implicit solvent setups. E.g. the range of the electrostatic values seen for vacuum will likely be larger than the range for implicit water. The range of values displayed can be set manually to provide consistent color scaling for comparison.

8.4.5. Adjusting the color scale of the color map

1. Select “Colors...” from the “Graphics” menu in the “VMD Main” window
 - This should cause the “Color Controls” window to pop up
2. Select the “Color Scale” tab
 - The color scheme can be selected from the “Method” pull down menu
 - The “Offset” and “Midpoint” sliders can be used to adjust the scaling of the color map.
 - If singularities are present, it may be difficult to get a good scaling for volume maps generated with fine grid spacings. In this case, either re-run with singularity removal on, or set the color scale range manually as shown in the next section.

When singularity removal is not employed, the presence of singularities will cause the range of the electrostatic potential distribution near the atom centers to be much wider than near the molecular surface. This typically results in very poor contrast particularly for implicit solvent since the high dielectric constant in the solvent region will amplify the effect. This can be compensated for by manually setting the Color Scale Data Range.

8.4.6. Changing the color scale range

1. Select desired representation to modify
2. Select “Volume” Coloring Method and Select the desired volumetric map to rescale from the pull down menu.
 - Each time you change the volumetric map being displayed, you will need to repeat this, so it is a good idea to make multiple representations for each potential data set rather than switching between them on the same representation.
3. Select the “Trajectory” tab

4. You should see the automatically computed range in the “Color Scale Data Range:” boxes. The left hand box controls the minimum value for the range, the right hand box controls the maximum value for the range.
5. Set the minimum and maximum values as needed to improve the contrast. Often the inner 10% to 30% of the total (automatic) range will give good contrast for a one angstrom grid spacing.
6. Click on the “Set” button when you are finished
7. To return to the automatic scaling that was originally calculated by *VMD*, click the “Autoscale” button.

Electrostatic potential data can also be used as a color map for other drawing methods. You will need to first load the data into the molecule you wish to display.

8.4.7. Loading electrostatic potential data into an existing molecule

The names of the files are used as labels, so it is useful to rename them from “pbsa_phi.dx” to something more descriptive before loading.

1. Select the molecule you wish to display the potential color map on in the “*VMD* Main” window
2. Go to the “File” menu in the *VMD* Main window.
3. Select “Load Data Into Molecule...”.
 - This will bring up the “Molecule File Browser” window
4. Click on the “Browse...” button in the “Molecule File Browser” window
5. Select the file “pbsa_phi.dx” that was generated by pbsa using the file selection dialogue that pops up.
 - The “Determine file type:” drop down menu should now read “DX”.
6. Click the “Load” button.

The data should now be loaded into the molecule you selected.

8.4.8. Using the electrostatic potential data as a color map

Once you have loaded a volumetric data file into a molecule, it can be used to generate a color map for any representations of that molecules model.

1. Open the “Graphical Representations” window if it is not already open
 - Select “Representations...” from the “Graphics” menu in the “*VMD* Main” window
2. Select the molecule you loaded the data into from the “Selected Molecule” pull down menu

8. PBSA

3. Select the representation you wish to map the potential color map onto
4. Select the “Draw Style” tab if it is not already selected
5. Select “Volume” from the “Coloring Method” pull down menu
 - Another pull down menu should appear next to it
 - Choose the selection that corresponds to the data you just loaded, it should be the last one on the list if it is the last one that was loaded.

VMD will attempt to automatically scale the color mapping used for Volumetric data that you load. The color scale may be manually adjusted if needed (see previous section)

8.4.9. Loading and displaying the level set map

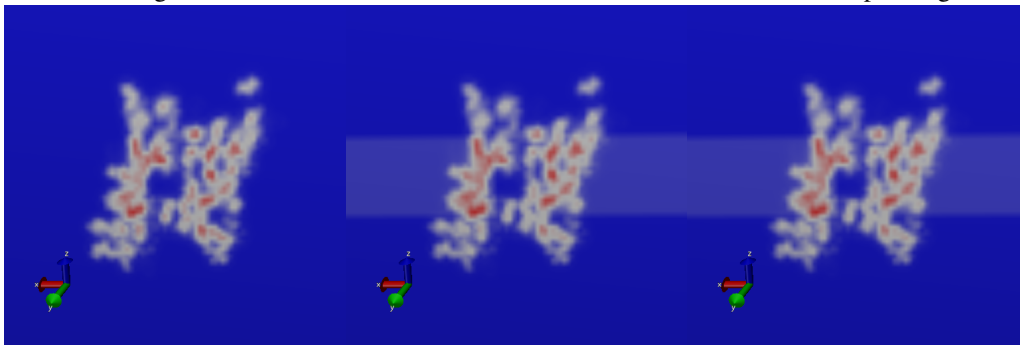
The level set used by *pbsa* to model the solute - solvent interface can be written to an output file in DX format by setting OUTLVLSET to “true” in the input file.

```
Sample PB visualization input
&cntrl
inp=0
/
&pb
npbverb=1, space=1., sasopt=2,
phiout=1, phiorm=2,
outlvlset=true
/
```

The level set will be written to a DX format volumetric data file named “pbsa_lvlset.dx”. This file can be used to visualize the corresponding molecular surface. The level set file is loaded into *VMD* in the same manner as an electrostatic potential data file. Cross sections can be viewed using the “Volume Slice” representation.

Shown below are the level sets for the aquaporin systems shown previously (no level set is shown for vacuum as there is no dielectric interface being modeled in that system)

From left to right: Water, Water + Slab-like membrane, Water + Membrane with pore region



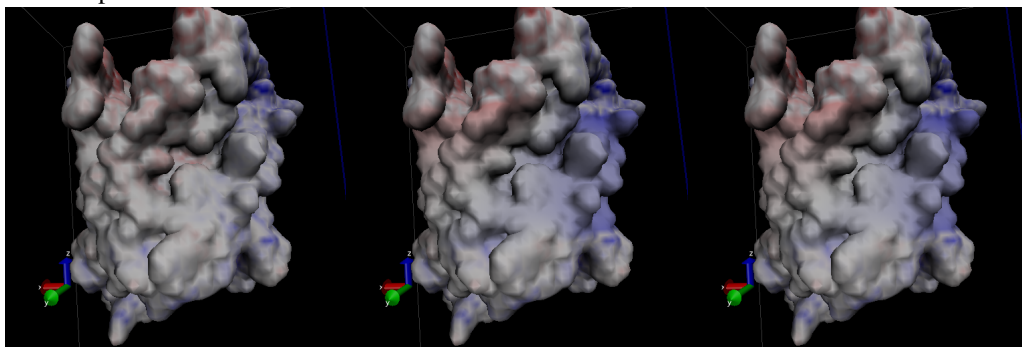
8.4.10. Visualizing the molecular surface as an isosurface of the level set

The level set is constructed such that the molecular surface is the locus of all points where the level set is zero. This allows us to use the Isosurface representation in *VMD* to display the solvent excluded surface by setting the “Isovalue” to 0. Alternatively, if we wish to view the potential just outside the surface, we can set the “Isovalue” to a number slightly higher than 0. E.g. 0.1 or 0.01.

1. Load the level set data file into the molecule.
 - This is done using the same procedure as loading an electrostatic potential data file, but the level set data file will be chosen instead of the potential data file.
2. Create a new Isosurface representation in the “Graphical Representations” window.
3. Select the volume map for the level set from the pull down menu
4. Choose an “Isovalue” at or slightly above 0.
5. Using the “Coloring Method” pull down menu, you may also use a previously loaded electrostatic potential data file as a color map by selecting “Volume” and then selecting the appropriate volume map from the pull down menu that appears.
 - *VMD* will automatically assign color scale range every time.
 - To compare multiple potential maps, it is often desirable to use the same color scale range for each. The best way to do this is to make a new representation for each potential map and manually assign the same color scale range to be identical for each (see previous section)

The examples below were generated for Aquaporin (1IH5 in the protein data bank) under various implicit solvent options using a FILLRATIO of 2.0, grid spacing of 1 Å. For each calculation, the periodic conjugate gradient solver with singularity removal was used. The level set for the system modeling implicit water was used to build the isosurfaces. The electrostatic potential data files were then overlayed as color maps with the color scale ranges set to [-80000,80000].

From Left to right: Water only, Water + Slab Like Membrane, Water + Membrane with 6 Å cylindrical pore.



8. PBSA

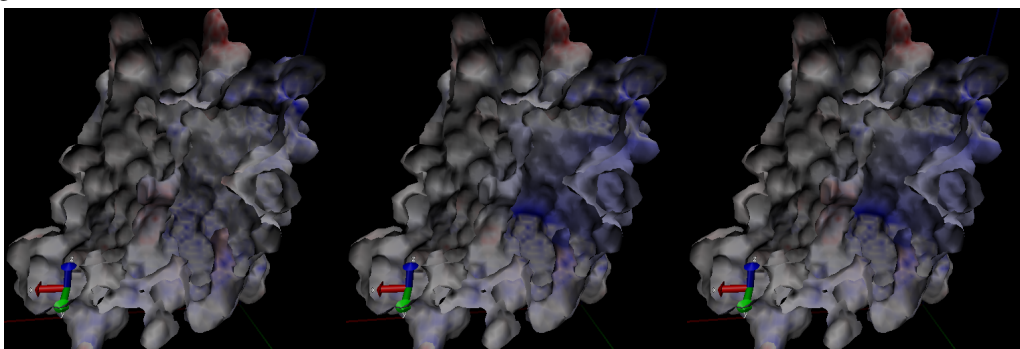
8.4.11. Visualizing interior channels, voids, and solvent pockets

One of the common roles for membrane proteins is to act as a transmembrane channel, to allow specific substance to pass from one side of a membrane to another. Features such as solvent / ion channels or internal voids will often be occluded from view by the exterior surface. One option that can allow these to be viewed is to use the clipping plane tool in VMD.

1. Open the “Extensions” pull down menu in the “VMD Main” window and go to the “Visualization” submenu and select “Clipping Plane Tool”.
2. The “Clip Tool” window should pop up.
3. The “Distance” slider allows clipping to be set
4. The “Normal” slider sets the normal of the clipping plane.
 - The “flip” button on the right will let you clip from front to back, which will be useful to clip the occluding exterior surface from the view and reveal the interior.

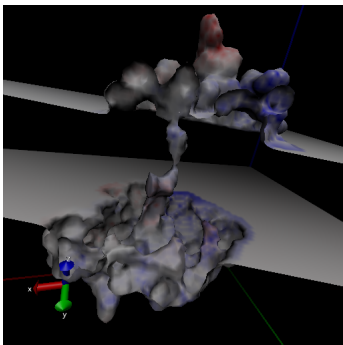
The clipping tool was used to reveal the internal pore region for the aquaporin system setups used in the previous section.

From Left to right: Water only, Water + Slab like Membrane, Water + Membrane with pore region excluded.



As an alternative, the level set map generated using PORTYPE=1 with the implicit membrane option will allow a cylindrical region to be excluded from the membrane level set. The corresponding isosurface will show any interior cavities or voids which fall within this region for isovalues at or slightly above 0 (since the level set at the membrane-solute interface will be below 0). See the previous section for details on writing and loading the level set file.

Shown below is the level set isosurface for the aquaporin system with implicit water plus a membrane with a cylindrical region removed. The corresponding potential data was again overlaid as a color map. The surface of the channel region, and the membrane-solvent interface planes are now clearly visible.



8.4.12. Importing / Modifying Atomic Radii to VMD from the prmtop file

Currently, VMD does not support loading radii for atoms directly from the prmtop file when it loads a molecule. These values can be loaded relatively easily using the tkconsole, however. To do so:

1. select “Tk Console” from the “Extensions” menu in the “VMD Main” window.
 - The “VMD TkConsole” window will then open
2. Be sure that the atom you want to import radii for is the top molecule on the list in the VMD Main window. If it is not, you will need to replace “top” with the appropriate ID
3. Type or copy and paste the following lines, but DO NOT hit enter yet.

```
set prot [atomselect "top" all]
$prot set radius {#RadiiList#}
```

4. You will now need to replace #RadiiList# with the one from the prmtop file.
 - a) Open the prmtop file for the molecule using a text editor
 - b) find the section that starts with “%FLAG RADII”
 - c) Highlight/Select the list of numbers that follows “%FORMAT(5E16.8)”
 - d) Copy the list (usually done by selecting “Copy” from the “Edit” menu in your text editor)
 - e) Go back to the “VMD TkConsole” window
 - f) Highlight/Select #RadiiList#
 - g) Select “Paste Ctrl-v” from the “Edit” menu in the “VMD TkConsole” window
5. Now hit return
 - If this was successful, you should now have the correct radii for each atom in the molecule.
 - you can have the console print the list of all radii by typing:

8. PBSA

```
$prot get radius
```

- For a more human readable printout, use:

```
for {set ind 0} {$ind<[llength $rad]} {incr ind} \  
{puts "Atom $ind radius is [lindex $rad $ind]}
```

These radii are used by VMD to display the VDW surface (made by selecting “VDW” from the “Drawing Method” pull down menu in the “Graphical Representations” window). One useful trick is to set them to be a small amount larger (say .01 Å) than those used to generate the surface. This will ensure that the color map will represent the external field just outside of the molecule. To modify the radii type or copy the following in the Tk Console:

```
set rad [$prot get radius]  
for {set ind 0} {$ind<[llength $rad]} {incr ind} \  
{lset rad $ind [expr [lindex $rad $ind] +.01]}
```

The above code will increase all atomic radii by .01 angstroms. This can be changed if a different amount is desired. (The code assumes you already followed steps 1 through 5 otherwise \$prot will be undefined!)

8.5. *pbsa* in *sander* and NAB

8.5.1. Electrostatic forces/gradients in *pbsa*

Force calculation in the finite-difference Poisson-Boltzmann method is straightforward, though not a trivial issue. It can be shown, by using the variation of the electrostatic free energy, that the electrostatic force density consists of three components, viz., the reaction field force, the dielectric boundary force, and the ionic force. [155] Since the ionic force is much smaller in absolute value than the other two components, we only include the reaction field force and the dielectric boundary force in this release.

The reaction field force only exists where there are atomic charges, so that it is straightforward to be mapped onto atoms. In contrast, the dielectric boundary force exists on the molecular surface where the dielectric constant changes. The surface force, or pressure, cannot be easily mapped onto atoms. This is because a force-mapping procedure from the molecular surface to atoms apparently needs the derivatives of molecular surface with respect to atomic positions. However such derivatives do not exist for the widely used molecular surface definition, i.e., the solvent excluded surface (SES). We are actively developing an analytical molecular surface definition that is consistent with the widely used SES definition for the numerical PB methods so that this difficulty will be overcome in future releases.

Temporarily, a partial solution in the mapping of dielectric boundary force as described by Gilson et al[155] is implemented for PB dynamics and minimization when the SES definition is used. The stability of the MD simulation has been much improved with a more accurate mapping method of analytical SES.

8.5.2. Example for *pbsa* in *sander*

All *pbsa* functionalities are available in *sander* and all input options are exactly the same as in the standalone *pbsa*. An apparent exception is IPB: you need to really set IPB to nonzero in order to invoke *pbsa* functionalities. All other default values of PB options in *sander* are same as those in *pbsa* for single point calculations, whereas there are some options that have different recommended or default values when PB minimization or dynamics is enabled. These options are

```
space=0.25
arcres=0.125
fscale=4
eneopt=1
bcopt=6
frcopt=1
```

The SPACE, ARCREC and FSCALE are all set for higher resolution of the grid so that the force calculation can be more accurate. The field view method (ENEOPT = 1, FRCOPT = 1) is used here because it has been tested to be able to run stable molecular dynamics simulations. Plus, BCOPT is set to 6 to remove charge singularity for the same stability purpose. An example input for PBMD is given as follows

```
Sample PB visualization input
&cntrl
imin=0, ntx=1, irest=0,
ipb=2, ntb=0,
ntc=2, ntf=2,
temp1=100, temp0=100, ntt=3, gamma_ln=1,
nstlim=100000, dt=0.002,
ntpr=100, ntwr=100, ntwx=100,
/
&pb
npbgrid=500, nsnba=5,
/
```

IPB is explicitly set to 2 to enable PB dynamics. The NPBGRID option is set to 500, which means the finite difference grid is regenerated every 500 dynamics steps. NSNBA = 5 means the atom-based pairlist is generated every 5 steps. Please refer to the *Amber* manual for the other &cntrl options. Note that the above input can be used with *sander* only.

8.5.3. Example for *pbsa* in NAB

pbsa functionalities are available in NAB as a part of the standard build. However the available input options are limited, please refer to the table in Section 18.1 for the list of available *pbsa* input options. The structures and parameters are supplied by NAB's facility. Here is a sample of calls in a NAB program to the *mm_options()* routine, in order to run *pbsa*:

8. PBSA

```
mm_options("ntpr=1, cut=99.0"); // No solute-solute cutoff
mm_options("ipb=2"); // Use PBSA
mm_options("accept=0.000001"); // Convergence criterion
mm_options("dprob=1.6"); // Solvent probe radius for SASA
mm_options("radiopt=1"); // Use atom-type/charge-based radii
mm_options("fillratio=4"); // Coarse/Fine ratio of electrostatic focusing
```


9. Reference Interaction Site Model

In addition to explicit and continuum implicit solvation models, Amber also has a third type of solvation model for molecular mechanics simulations, the reference interaction site model (RISM) of molecular solvation[156–169]. In AmberTools, 1D-RISM is available as `rism1d`. 3D-RISM is available as an option in NAB, MMPBSA.py and `sander`. `rism3d.snglpnt` is a simplified, standalone interface, ideal for calculating solvation thermodynamics on individual structures and trajectories. Details specific to using `sander` and `sander.MPI` can be found in the Amber manual.

9.1. Introduction

RISM is an inherently microscopic approach, calculating the equilibrium distribution of the solvent, from which all thermodynamic properties are then arrived at. Specifically, RISM is an approximate solution to the Ornstein-Zernike (OZ) equation[157, 166, 167, 170, 171]

$$h(r_{12}, \Omega_1, \Omega_2) = c(r_{12}, \Omega_1, \Omega_2) + \rho \int d\mathbf{r}_3 d\Omega_3 c(r_{13}, \Omega_1, \Omega_3) h(r_{32}, \Omega_3, \Omega_2), \quad (9.1)$$

where r_{12} is the separation between particles 1 and 2 while Ω_1 and Ω_2 are their orientations relative to the vector \mathbf{r}_{12} . The two functions in this relation are h , the total correlation function, and c , the direct correlation function. The total correlation function is defined as

$$h_{ab}(r_{ab}, \Omega_a, \Omega_b) \equiv g_{ab}(r_{ab}, \Omega_a, \Omega_b) - 1,$$

where g_{ab} is the pair-distribution function, which gives the conditional density distribution of species b about a . In cases where only radial separation is considered, for example by orientational averaging over site α of species a and site γ of species b , gives the familiar one dimensional site-site radial distribution function, $g_{\alpha\gamma}(r_{\alpha\gamma})$.

For real mixtures, it is often convenient to speak in terms of a solvent, V, of high concentration and a solute, U, of low concentration. A generic case of solvation is infinite dilution of the solute, i.e., $\rho^U \rightarrow 0$. We can rewrite Equation (9.1), in the limit of infinite dilution, as a set of

9. Reference Interaction Site Model

three equations:

$$h^{VV}(r_{12}, \Omega_1, \Omega_2) = c^{VV}(r_{12}, \Omega_1, \Omega_2) + \rho^V \int d\mathbf{r}_3 d\Omega_3 c^{VV}(r_{13}, \Omega_1, \Omega_3) h^{VV}(r_{32}, \Omega_3, \Omega_2), \quad (9.2)$$

$$h^{UV}(r_{12}, \Omega_1, \Omega_2) = c^{UV}(r_{12}, \Omega_1, \Omega_2) + \rho^V \int d\mathbf{r}_3 d\Omega_3 c^{UV}(r_{13}, \Omega_1, \Omega_3) h^{VV}(r_{32}, \Omega_3, \Omega_2), \quad (9.3)$$

$$h^{UU}(r_{12}, \Omega_1, \Omega_2) = c^{UU}(r_{12}, \Omega_1, \Omega_2) + \rho^V \int d\mathbf{r}_3 d\Omega_3 c^{UV}(r_{13}, \Omega_1, \Omega_3) h^{VU}(r_{32}, \Omega_3, \Omega_2). \quad (9.4)$$

Equation (9.3) is directly relevant for biomolecular simulations where we are often interested in the properties of a single, arbitrarily complex solute in the solution phase. Solutions to Equation (9.3) can be obtained using 3D-RISM. However, a solution to Equation (9.2) for pure solvent is a necessary prerequisite and is readily obtained from 1D-RISM.

To obtain a solution to the OZ equations it is necessary to have a second equation that relates h and c or uniquely defines one of these functions. The general closure relation is [170]

$$g(r_{12}, \Omega_1, \Omega_2) = \exp[-\beta u(r_{12}, \Omega_1, \Omega_2) + h(r_{12}, \Omega_1, \Omega_2) - c(r_{12}, \Omega_1, \Omega_2) + b(r_{12}, \Omega_1, \Omega_2)] \quad (9.5)$$

u is the potential energy function for the two particles and b is known as the bridge function (a non-local functional, representable as infinite diagrammatic series in terms of h [170]). It should be noted that u is the only point at which the interaction potential enters the equations. Depending on the method used to solve the OZ equations, u is generally an explicit potential. In principle, it should now be possible to solve our two equations. For example, we may wish to use SPC/E as a water model. Inputting the relevant aspects of the SPC/E model into u , 1D-RISM can be used to calculate the equilibrium properties of the SPC/E model. A different explicit water model will yield different properties.

A fundamental problem for all OZ-like integral equation theories is the bridge function, which contains multiple integrals that are readily solved only in special circumstances. In practice, an approximate closure relation must be used. While many closures have been developed, at this time only three are implemented in 3D-RISM: hypernetted-chain approximation (HNC), Kovalenko-Hirata (KH) and the partial series expansion of order- n (PSE- n).

For HNC, we set $b = 0$, giving [170]

$$\begin{aligned} g^{\text{HNC}}(r_{12}, \Omega_1, \Omega_2) &= \exp(-\beta u(r_{12}, \Omega_1, \Omega_2) + h(r_{12}, \Omega_1, \Omega_2) - c(r_{12}, \Omega_1, \Omega_2)) \\ &= \exp(t^*(r_{12}, \Omega_1, \Omega_2)) \end{aligned} \quad (9.6)$$

where t^* is the renormalize-indirect correlation function. HNC works well in many situations, including charged particles, but has difficulties when the size ratios of particles in the system are highly varied and may not always converge on a solution when one should exist. Also, as the bridge term is generally repulsive, HNC allows particles to approach too closely, overestimating non-Coulombic interactions [167].

KH is a combination of HNC and the mean spherical approximation (MSA), the former being applied to the spatial regions of solvent density depletion ($g < 1$), including the repulsive core,

and the latter to those of solvent density enrichment ($g > 1$), such as association peaks[166, 167]

$$g^{\text{KH}}(r_{12}, \Omega_1, \Omega_2) = \begin{cases} \exp(t^*(r_{12}, \Omega_1, \Omega_2)) & \text{for } g(r_{12}, \Omega_1, \Omega_2) \leq 1 \\ 1 + t^*(r_{12}, \Omega_1, \Omega_2) & \text{for } g(r_{12}, \Omega_1, \Omega_2) > 1 \end{cases}. \quad (9.7)$$

Like HNC, KH handles Coulombic systems well but overestimates non-Coulombic interactions. Unlike HNC, it does not have difficulties with highly asymmetric particle sizes and readily converges to stable solutions for almost all systems of practical interest. The reliability of the KH closure makes it particularly suitable for molecular mechanics calculations.

PSE- n offers the ability to interpolate between KH and HNC. Here, the exponential regions of solvent density enrichment are treated as a Taylor expansion,

$$g^{\text{PSE-}n}(r_{12}, \Omega_1, \Omega_2) = \begin{cases} \exp(t^*(r_{12}, \Omega_1, \Omega_2)) & \text{for } g(r_{12}, \Omega_1, \Omega_2) \leq 1 \\ \sum_{i=0}^n (t^*(r_{12}, \Omega_1, \Omega_2))^i / i! & \text{for } g(r_{12}, \Omega_1, \Omega_2) > 1 \end{cases}. \quad (9.8)$$

In the case of $n = 1$, the KH closure is obtained, while in the limit of $n \rightarrow \infty$ HNC is recovered. This allows a balance between the numerical stability of KH and the often better accuracy of HNC.

9.1.1. 1D-RISM

1D-RISM is used to calculate bulk properties of the solvent and is a prerequisite for 3D-RISM, for which the primary result is the bulk solvent site-site susceptibility in reciprocal space, $\chi^{\text{VV}}(k)$. As its name would suggest, 1D-RISM is a one-dimensional calculation. The six-dimensional OZ equations are reduced to one dimension (radial separation) via the fundamental RISM approximation[157–160, 170, 171], which produces the intramolecular pair correlation matrix,

$$\omega_{\alpha\gamma}(k) = \sin(kr_{\alpha\gamma}) / (kr_{\alpha\gamma}) \quad (9.9)$$

where α and γ label the different atom types in the model. Note that atoms of the same type in RISM theory have the same Lennard-Jones and Coulomb parameters. For example, most three site water models have two RISM types, oxygen and hydrogen. Depending on the model, propane, C_3H_8 , may have two carbon types and two hydrogen types. Equation (9.2) then becomes

$$\begin{aligned} h_{\alpha\gamma}(r) &= \sum_{\mu\nu} \int d\mathbf{r}' d\mathbf{r}'' \omega_{\alpha\mu}(|r - r'|) c_{\mu\nu}(|r' - r''|) [\omega_{\nu\gamma}(r'') + \rho_{\nu} h_{\nu\gamma}(r'')] \\ &= \frac{1}{(2\pi)^3} \int e^{i\mathbf{k}\cdot\mathbf{r}} d\mathbf{k} \left[\omega \mathbf{c} [\mathbf{1} - \rho \omega \mathbf{c}]^{-1} \omega \right]_{\alpha\gamma} \\ &= \sum_0^\infty \omega(k) \mathbf{c}(k) \omega(k) [\rho \mathbf{c}(k) \omega(k)]^n. \end{aligned} \quad (9.10)$$

Equation (9.10) must be complemented with one of the five closures currently supported by rism1d (see §9.4.1.1). In 1d, these are site-site closures and there is no orientational dependence. For example, the HNC closure (Eq. (9.6)) becomes,

9. Reference Interaction Site Model

$$g_{\alpha\gamma}^{\text{HNC}}(r) = \exp \left[-\beta u_{\alpha\gamma}(r) + h_{\alpha\gamma}(r) - c_{\alpha\gamma}(r) \right]. \quad (9.11)$$

Equation (9.10), with KH, HNC or PSE- n closures, is readily applicable to liquid mixtures, with site indices of the site-site correlation functions enumerating interaction sites on all (different) species in the solution and the intramolecular matrix (9.9) set equal to zero for sites α, γ belonging to different species.

A dielectrically consistent version of 1D-RISM theory (DRISM) enforces the proper dielectric asymptotics of the site-site correlation functions, and so provides the self-consistent dielectric properties of electrolyte solution with polar solvent and salt in a range of concentrations, including the given dielectric constant of the solution [172].

The 1D-RISM integral equations are then solved for the site-site direct correlation function in an iterative manner, accelerated by the modified direct inversion of the iterative subspace (MDIIS) [167, 173]. All correlation functions are represented as one-dimensional grids and the convolution integrals in Equation (9.10) are performed in reciprocal space by making use of a fast Fourier transform applied to the short-range parts of all the correlations, while the electrostatic asymptotics are separated out and Fourier transformed analytically [167–169].

9.1.2. 3D-RISM

With the results from 1D-RISM, a 3D-RISM calculation for a specific solute can be carried out. For 3D-RISM calculations, only the solvent orientational degrees of freedom are averaged over and Equation (9.3) becomes [165, 166]

$$h_{\gamma}^{\text{UV}}(\mathbf{r}) = \sum_{\alpha} \int d\mathbf{r}' c_{\alpha}^{\text{UV}}(\mathbf{r} - \mathbf{r}') \chi_{\alpha\gamma}^{\text{VV}}(r'), \quad (9.12)$$

where $\chi_{\alpha\gamma}^{\text{VV}}(r)$ is the site-site susceptibility of the solvent, obtained from 1D-RISM and given by

$$\chi_{\alpha\gamma}^{\text{VV}}(r) = \omega_{\alpha\gamma}^{\text{VV}}(r) + \rho_{\alpha} h_{\alpha\gamma}^{\text{VV}}(r).$$

3D-RISM supports HNC, KH and PSE- n closures (see §9.6.1, 18.1 and 10.3.1). As with the 1D-RISM closures, these are constructed by analogy from Eqs. 9.6–9.8. For example, HNC becomes

$$g_{\gamma}^{\text{HNC,UV}}(\mathbf{r}) = \exp \left(-\beta u_{\gamma}^{\text{UV}}(\mathbf{r}) + h_{\gamma}^{\text{UV}}(\mathbf{r}) - c_{\gamma}^{\text{UV}}(\mathbf{r}) \right). \quad (9.13)$$

As with 1D-RISM, correlation functions are represented on (3D) grids, convolution integrals are performed in reciprocal space and a self-consistent solution is iteratively converged upon using the MDIIS accelerated solver. There is one 3D grid for each solvent type for each correlation function. For example, for a solute in SPC/E water there will be both $g_{\text{H}}^{\text{UV}}(\mathbf{r})$ and $g_{\text{O}}^{\text{UV}}(\mathbf{r})$ grids. Each point on the $g_{\text{H}}^{\text{UV}}(\mathbf{r})$ will give the fractional density of water hydrogen at that location of real-space.

To properly treat electrostatic forces in electrolyte solution with polar molecular solvent and ionic species, the electrostatic asymptotics of all the correlation functions (both the 3D and radial ones) are treated analytically [167, 168, 174]. The non-periodic electrostatic asymptotics are separated out in the direct and reciprocal space and the remaining short-range terms of the

correlation functions are discretized on a 3D grid in a non-periodic box large enough to ensure decay of the short-range terms at the box boundaries [174]. The convolution of the short-range terms in the integral equation (9.12) is calculated using 3D fast Fourier transform [175, 176]. Accordingly, the electrostatic asymptotics terms in the thermodynamics integral (9.15) below are handled analytically and reduced to one-dimensional integrals easy to compute [174].

With a converged 3D-RISM solution for h^{UV} and c^{UV} it is straightforward to calculate solvation thermodynamics. From the perspective of molecular simulations, the most important thermodynamic values are the excess chemical potential of solvation (solvation free energy), μ^{ex} and the mean solvation force, $\mathbf{f}_i^{\text{UV}}(\mathbf{R}_i)$, on each solute atom, i . μ^{ex} can be obtained through analytical thermodynamic integration for HNC,

$$\mu^{\text{ex,HNC}} = k_{\text{B}}T \sum_{\alpha} \rho_{\alpha}^{\text{V}} \int d\mathbf{r} \left[\frac{1}{2} (h_{\alpha}^{\text{UV}}(\mathbf{r}))^2 - c_{\alpha}^{\text{UV}}(\mathbf{r}) - \frac{1}{2} h_{\alpha}^{\text{UV}}(\mathbf{r}) c_{\alpha}^{\text{UV}}(\mathbf{r}) \right], \quad (9.14)$$

KH,

$$\mu^{\text{ex,KH}} = k_{\text{B}}T \sum_{\alpha} \rho_{\alpha}^{\text{V}} \int d\mathbf{r} \left[\frac{1}{2} (h_{\alpha}^{\text{UV}}(\mathbf{r}))^2 \Theta(-h_{\alpha}^{\text{UV}}(\mathbf{r})) - c_{\alpha}^{\text{UV}}(\mathbf{r}) - \frac{1}{2} h_{\alpha}^{\text{UV}}(\mathbf{r}) c_{\alpha}^{\text{UV}}(\mathbf{r}) \right], \quad (9.15)$$

and PSE- n ,

$$\mu^{\text{ex,PSE-}n} = k_{\text{B}}T \sum_{\alpha} \rho_{\alpha}^{\text{V}} \int d\mathbf{r} \left[\frac{1}{2} (h_{\alpha}^{\text{UV}}(\mathbf{r}))^2 - c_{\alpha}^{\text{UV}}(\mathbf{r}) - \frac{1}{2} h_{\alpha}^{\text{UV}}(\mathbf{r}) c_{\alpha}^{\text{UV}}(\mathbf{r}) - \frac{(t^*(\mathbf{r}))^{n+1}}{(n+1)!} \Theta(h_{\alpha}^{\text{UV}}(\mathbf{r})) \right], \quad (9.16)$$

where Θ is the Heaviside function.

Analogous versions of Eqns. 9.6, 9.15 and 9.16 are used in 1D-RISM. While these are used for DRISM they have been derived for XRISM. Furthermore, these equations have been derived a number of different ways with slightly different functional forms of the $-\frac{1}{2}hc$ term [166, 177–180]. These different functional forms are equivalent in XRISM but not in DRISM. The form introduced by Pettitt and Rossky [178] is the most popular in the literature and the default selection in `rismld`. It is possible to have `rismld` evaluate and output all three functional forms (see [Output](#)) but, for DRISM, none of these expressions are correct.

The force equation

$$\mathbf{f}_i^{\text{UV}}(\mathbf{R}_i) = -\frac{\partial \mu^{\text{ex}}}{\partial \mathbf{R}_i} = -\sum_{\alpha} \rho_{\alpha} \int d\mathbf{r} g_{\alpha}^{\text{UV}}(\mathbf{r}) \frac{\partial u_{\alpha}^{\text{UV}}(\mathbf{r} - \mathbf{R}_i)}{\partial \mathbf{R}_i}$$

is valid for all closures with a path independent expression for the excess chemical potential, such as HNC, KH and PSE- n closures implemented in 3D-RISM [156, 181–183].

In addition to closure specific expressions for the solvation free energy, other approximations also exist. The Gaussian fluctuation (GF) approximation[184, 185] is given as

$$\mu^{\text{ex,GF}} = k_{\text{B}}T \sum_{\alpha} \rho_{\alpha}^{\text{V}} \int d\mathbf{r} \left[-c_{\alpha}^{\text{UV}}(\mathbf{r}) - \frac{1}{2} h_{\alpha}^{\text{UV}}(\mathbf{r}) c_{\alpha}^{\text{UV}}(\mathbf{r}) \right]$$

9. Reference Interaction Site Model

and has been shown to yield improved absolute solvation free energies for both polar and non-polar solutes[185, 186] but not necessarily for relative free energies[187]. It is not associated with a particular closure but is typically used in place of the expression for a given closure.

Eqs. (9.14)-(9.16) give the total solvation free energy, ΔG_{sol} , but it is often useful to decompose this into electrostatic (solvent polarization), ΔG_{pol} , and non-electrostatic (dispersion and cavity formation), $(\Delta G_{\text{dis}} + \Delta G_{\text{cav}})$, terms. Conceptually, we can divide the path of the thermodynamic integration into two steps: first the solute without partial charges is inserted into the solvent (dispersion and cavity formation) and then partial charges are introduced, which polarize the solvent,

$$\mu^{\text{ex}} = \Delta G_{\text{sol}} = \Delta G_{\text{pol}} + \Delta G_{\text{dis}} + \Delta G_{\text{cav}}.$$

ΔG_{sol} is produced by a 3D-RISM calculation on the charged solute. ΔG_{pol} is then the difference of the two calculations. As a point of reference, generalized-Born and Poisson-Boltzmann methods calculate only ΔG_{pol} and, typically, use a calculation involving solvent accessible surface area to predict $\Delta G_{\text{dis}} + \Delta G_{\text{cav}}$.

9.1.3. Analytic Temperature Derivatives

For the thermodynamic analysis of solvation, it is often useful to calculate the energetic and entropic contributions, ϵ^{solv} and $-TS^{\text{solv}}$ respectively, to the solvation free energy. It has been shown that it is possible to analytically decompose the solvation free energy into these two contributions when the solvation free energy has a closed analytical form, such as with HNC and KH closure [188]. In what follows, the analytical expression of energetic and entropic contributions to the solvation free energy are derived in the framework of 1D-RISM theory with HNC closure. The similar derivation can be applied to other closures as well as to the framework of 3D-RISM theory. At this time, temperature derivatives are implemented for `rism1d` with HNC, KH and PSE- n closures.

The solvation free energy of species U in a solution consisting of N total species is expressed in the RISM-HNC framework as

$$\mu_{\text{HNC}}^{\text{ex,U}} = k_{\text{B}} T \sum_{\alpha}^{\text{on U}} \sum_{M=1}^N \sum_{\gamma}^{\text{on M}} \rho_{\gamma} \int d\mathbf{r} \left[\frac{1}{2} (h_{\alpha\gamma}(r))^2 - c_{\alpha\gamma}(r) - \frac{1}{2} h_{\alpha\gamma}(r) c_{\alpha\gamma}(r) \right].$$

The differentiation of the solvation free energy with respect to the temperature T leads to

$$\delta_T \mu_{\text{HNC}}^{\text{ex,U}} = \mu_{\text{HNC}}^{\text{ex,U}} + k_{\text{B}} T \sum_{\alpha}^{\text{on U}} \sum_{M=1}^N \sum_{\gamma}^{\text{on M}} \rho_{\gamma} \int d\mathbf{r} \left[h_{\alpha\gamma}(r) \cdot \delta_T h_{\alpha\gamma}(r) - \delta_T c_{\alpha\gamma}(r) - \frac{1}{2} \delta_T h_{\alpha\gamma}(r) \cdot c_{\alpha\gamma}(r) - c_{\alpha\gamma}(r) - \frac{1}{2} h_{\alpha\gamma}(r) \cdot \delta_T c_{\alpha\gamma}(r) \right].$$

where δ_T is $T \frac{\partial}{\partial T}$. Since $\mu_{\text{HNC}}^{\text{ex,U}} = \epsilon^{\text{solv,U}} - TS^{\text{solv,U}}$, we have $\delta_T \mu_{\text{HNC}}^{\text{ex,U}} = -TS^{\text{solv,U}}$ and therefore the above equation can be rearranged as

$$\epsilon^{\text{solv,U}} = -k_{\text{B}} T \sum_{\alpha}^{\text{on U}} \sum_{M=1}^N \sum_{\gamma}^{\text{on M}} \rho_{\gamma} \int d\mathbf{r} \left[h_{\alpha\gamma}(r) \cdot \delta_T h_{\alpha\gamma}(r) - \delta_T c_{\alpha\gamma}(r) - \frac{1}{2} \delta_T h_{\alpha\gamma}(r) \cdot c_{\alpha\gamma}(r) - \frac{1}{2} h_{\alpha\gamma}(r) \cdot \delta_T c_{\alpha\gamma}(r) \right].$$

It is noted that the solvation energy $\epsilon^{solv,U}$ can be viewed as consisting of two contributions: one arising from creation of a polarized cavity (in pure solvent) and the other corresponding to the energy of embedding the solute molecule into the cavity. The former is the solvent reorganization energy and the latter is the average solute-solvent interaction energy that is obtained as $\sum_{\alpha} \sum_{\gamma} \rho_{\gamma} \int d\mathbf{r} u_{\alpha\gamma} g_{\alpha\gamma}$.

The temperature derivatives of correlation functions $\delta_T h(r)$ and $\delta_T c(r)$ can be obtained by solving the temperature derivative of RISM-HNC equations

$$\delta_T \mathbf{h}(k) = \mathbf{w}(k) \delta_T \mathbf{c}(k) \mathbf{w}(k) + \rho \mathbf{w}(k) \delta_T \mathbf{c}(k) \mathbf{h}(k) + \rho \mathbf{w}(k) \mathbf{c}(k) \delta_T \mathbf{h}(k)$$

and

$$\delta_T h_{\alpha\gamma}(r) = \left[\frac{u_{\alpha\gamma}(r)}{k_B T} + \delta_T h_{\alpha\gamma}(r) - \delta_T c_{\alpha\gamma}(r) \right] (h_{\alpha\gamma}(r) + 1).$$

Some practical examples can be found in [189] and [190].

9.2. Practical Considerations

9.2.1. Computational Requirements and Parallel Scaling

Calculating a 3D-RISM solution for a single solute conformation typically requires about 100 times more computer time than the same calculation with explicit solvent or PB. While there are other factors to consider, such as sampling confined solvent or overall efficiency of sampling in the whole statistical ensemble at once, this can be prohibitive for many applications. Memory is also an issue as the 3D correlation grids require anywhere from a few megabytes for the smallest solutes to gigabytes for large complexes. A lower bound and very good estimate for the total memory required is

$$\text{Total memory} \geq 8 \text{ bytes} \times \left[N_{\text{box}} N^V \left(\underbrace{2N_{\text{MDIIS}}}_{c, \text{residual}} + \underbrace{1}_u + \underbrace{N_{\text{decomp}}}_{\text{polar decomp}} \underbrace{N_{\text{propagate}}}_{\text{past solutions}} \right) \right. \\ \left. (N_{\text{box}} + 2N_y N_z) \left\{ \underbrace{4}_{\text{asymptotics}} + \underbrace{1}_{\text{FFT scratch}} + \underbrace{2}_{g,h} N^V \right\} \right]$$

where $N_{\text{box}} = N_x \times N_y \times N_z$ is the total number of grid points, N^V is the number of solvent atom species and N_{MDIIS} is the number of MDIIS vectors used to accelerate convergence. u^{UV} , c^{UV} and the residual of c^{UV} are stored in real-space only and require a full grid for each solvent. c^{UV} and its residual also require N_{MDIIS} grids for the MDIIS routine (see the `mdiis_nvec` keyword) and $N_{\text{propagate}}$ grids to make use of solutions from previous solute configurations to improve the initial guess (see the `npropagate` keyword). If a polar/non-polar decomposition is requested (see the `polardecomp` keyword) an additional set of grids for past solutions with no solute charges is kept ($N_{\text{decomp}} = 2$); by default this is turned off ($N_{\text{decomp}} = 1$). The full real

9. Reference Interaction Site Model

space grid plus an additional $2N_yN_x$ grid points are needed (due to the FFT) for g and h for each solvent species and for the four grids required to compute the long range asymptotics. Memory, therefore, scales linearly with N_{box} while computation time scales as $O(N_{\text{box}} \log(N_{\text{box}}))$ due to the requirements of calculating the 3D fast Fourier transform (3D-FFT). To overcome these requirements, two options are available beyond optimizations already in place, multiple time steps and parallelization. Multiple time step methods are available only in `sander` (see the Amber manual) and are applicable to molecular dynamics calculations only. Parallelization is available for all calculations but is limited by system size and computational resources.

Both `sander` and `NAB` have MPI implementations of 3D-RISM (see Section 9.5.5 for `NAB` compiling instructions) that distribute both memory requirements and computational load. As memory is distributed, the aggregate memory of many computers can be used to perform calculations on very large systems. Memory distribution is handled by the FFTW 3.3 library so decomposition is done along the z-axis. If a variable solvation box size is used, the only consideration is to avoid specifying a large, prime number of processes (≥ 7). For fixed box sizes, the number of grids points in each dimension must be divisible by two (a general requirement) and the number of grid points in the z-axis must be divisible by the number of processes. `sander.MPI` also has the additional consideration that the number of processes cannot be larger than the number of solute residues; `NAB` does not suffer from this limitation.

9.2.2. Output

g^{UV} , h^{UV} and c^{UV} files can be output for 3D-RISM calculations and are useful for visualization and calculation of thermodynamic quantities. These use the ASCII Data Explorer (DX) file format (Section 9.7.6) so there is one file for each solvent atom type for each requested frame. Each file is $(348 + N_{\text{box}} \times 16\frac{1}{3})$ bytes, which can quickly fill disk space. Also, very few visualization programs are capable of displaying both molecular and volumetric trajectories.

9.2.3. Numerical Accuracy

Numerical accuracy depends on the specified residual tolerance for the solution and the solvation box physical size and grid spacing. Almost all applications should use a grid spacing of 0.5 Å. A larger grid spacing quickly leads to severe errors in thermodynamic quantities. Smaller grid spacing may be necessary for some applications (e.g., mapping potentials of mean force) but this is rare and computationally expensive. A buffer distance between the solute and the edges of the solvent box should typically be 14 Å for water or larger for ionic solutions. Molecular dynamics[156], minimization and trajectory post-processing[187] have different requirements for the maximum residual tolerance. Molecular dynamics does well with a tolerance of 10^{-5} and `npropagate=5`. Minimization requires tolerances of 10^{-11} or lower and `drms` $\geq 10^{-4}$. Trajectory post-processing for MM/RISM type calculations typically have high statistical noise from the trajectory itself and it is possible to use a tolerance of 10^{-3} and `npropagate=1`. However, this should be compared against a tolerance of 10^{-5} on a subset of the data before committing to this level of accuracy.

9.3. Work Flow

Using 3D-RISM with SANDER or NAB for molecular dynamics, minimization or snapshot analysis is very similar to using implicit solvent models like GBSA or PBSA. However, some additional preliminary setup is required, the extent of which depends on the solvent to be used.

3D-RISM requires detailed information of the bulk solvent in the form of the site-site susceptibility, χ^{VV} , and properties such as the temperature and partial charges. This is read in as an `.xvv` file, which is produced by a 1D-RISM calculation. If another 3D-RISM calculation is to be preformed with any details of the bulk solvent changed (e.g., temperature or pressure) a new `.xvv` file must be produced. Examples of precomputed `.xvv` files for SPC/E and TIP3P water can be found in `$AMBERHOME/AmberTools/test/rism1d`.

Special care must be taken when producing `.xvv` files for use with 3D-RISM, particularly with respect to grid parameters. It is important that the spatial extent of the grid be large enough to capture the essential long range features of the solvent while the spacing must be fine enough to sample the short-range structure. A grid spacing of 0.025 Å is sufficient for most applications. The number of grid points required, which will determine the physical length of the grid in Å, generally depends on the properties of the solvent. Low concentration aqueous salt solutions typically require much larger grids than pure bulk water. A good indicator that the grid is large enough is convergence of `delhv0` in the `.xvv` file. When converged, `delhv0` should retain four to five digits of precision when the number of grid points is doubled.

1D-RISM calculations require details of the some bulk properties of the solvent, such as temperature and dielectric constant, and an explicit model of the molecular components. These are read in from one or more `.mdl` files, depending on the composition of the solvent. Several `.mdl` files are included in the Amber11 distribution and can be found in `$AMBERHOME/dat/rism1d/mdl`. These include many of the explicit models for solvent and ions used with the Amber force fields. Other solvents models may be used by creating appropriate MDL files. See Section 9.7.1 for format details.

9.4. rism1d

1D-RISM calculations are carried out with `rism1d`, and require only one input file with an `.inp` suffix. The input file is listed on the command line without this suffix.

rism1d inputfile

Parameters for the calculation are read in from `parameters` name list.

9.4.1. Parameters

Note that these keywords are not case sensitive.

9.4.1.1. Theory

theory [DRISM] The 1D-RISM theory to use.

DRISM Dielectrically consistent RISM (recommended).

9. Reference Interaction Site Model

XRISM Extended RISM.

closure [KH] The type of closure to use.

KH Kovalenko-Hirata (recommended).

PSE n Partial serial expansion of order n . E.g., “PSE3”.

HNC Hyper-netted chain equation.

PY Percus-Yevick.

temperature_deriv [1] Solve another set of integral equations to calculate the temperature derivative. This typically adds less than 50% to the compute time and yields an energy/entropy decomposition of the excess chemical potential for all species and sites.

0 Do not calculate the temperature derivative.

1 Calculate the temperature derivative.

9.4.1.2. Grid Size

dr [0.025] Grid spacing in real space in Å.

nr [16384] Number of grid points. Should be a product of small prime factors (2, 3 and 5).

9.4.1.3. Output

outlist [] Indicates what output files to produce. Output file names use the root name of the input file with an extension listed below. This is a list of any combination of the following characters in any order, upper or lower case.

U $U^{VV}(r)$ Solvent site-site potential in real space, `inputfile.uvv` (see §9.7.3).

X $\chi^{VV}(k)$ Solvent site-site susceptibility in reciprocal space. Required input for 3D-RISM, `inputfile.xvv` (see §9.7.2).

G $G^{VV}(r)$ Solvent site-site pair distribution function in real-space, `inputfile.gvv` (see §9.7.3).

B $B^{VV}(r)$ Solvent site-site bridge correction in real space, `inputfile.bvv` (see §9.7.3).

T Thermodynamic properties of the solvent, `inputfile.therm` (see §9.7.4).

E $\text{ex}N^{VV}(r)$, $\text{ex}N^{VV}$ Solvent site-site running, `inputfile.exnvv` (see §9.7.3), and total, `inputfile.n00` (see §9.7.5), excess coordination numbers in real space.

N $N^{VV}(r)$ Solvent site-site running coordination numbers in real space, `inputfile.nvv` (see §9.7.3).

Q $\text{ex}Q^{VV}$ Solvent site-site excess total charge of site γ about α , `inputfile.q00` (see §9.7.5).

S	$S^{VV}(k)$ Solvent site-site structure factor in reciprocal space, <code>inputfile.svv</code> (see §9.7.3).
rout	[0] Largest real space separation in Å for output files. If 0 then all grid points will be output.
kout	[0] Largest reciprocal space separation in Å ⁻¹ for output files. If 0 then all grid points will be output.
ksave	[-1] Output an intermediate solution every <code>ksave</code> steps. If <code>ksave</code> ≤ 0 then no intermediate restart files are written. If any restart files are present at run time (<code>.sav</code> suffix) they are automatically used. However, such files are non-portable binary files.
progress	[1] Write the current residue to standard output every <code>progress</code> iteration. If <code>progress</code> ≤ 0 then residue is not reported.
selftest	[0] If '1', perform a self-consistency check and output the results to <code>inputfile.self.test</code> . Only tests applicable to the input parameters and system are performed. The results will depend on the input parameters (e.g., 'tolerance') used.

9.4.1.4. Species keywords

For each molecular species in the solvent mixture, a `species` name list should be provided.

density	[] (Required.) Density of the species in M. See 'units' below.
units	['M'] Units for density value. Options are 'M' (molar), 'mM' (millimolar), '1/A^3' (number per Å ³), 'g/cm^3' (g/cm ³) or 'kg/m^3' (kg/m ³).
model	[] (Required.) Relative or absolute path to and name of the <code>.mdl</code> file with the parameters for this solvent molecule.

9.4.1.5. Solution Convergence

rism1d uses MDIIS to accelerate convergence. The default parameters for this method are usually near optimal but some systems can be difficult to converge. In such cases it may be useful to use a small step size (`mdiis_del`=0.1 or 0.2). Occasionally, the target tolerance of 10⁻¹² can not be achieved. A tolerance of 10⁻¹⁰ to 10⁻¹¹ is often sufficient but it is advisable to check how sensitive your calculations are to this.

mdiis_nvec	[20] Number of MDIIS vectors to use.
mdiis_del	[0.3] MDIIS step size.
tolerance	[1e-12] Target residual tolerance for the self-consistent solution.
maxstep	[10000] Maximum number of iterations to converge to a solution.

9. Reference Interaction Site Model

extra_precision [1] Controls the use of extra precision routines at key points in the 1D-RISM solver. This can be useful for achieving low tolerances or for very large box lengths but increases computational cost. Strongly recommended for solutions with charged particles (e.g., salts).

- | | |
|---|--|
| 0 | No extra precision routines are used. |
| 1 | Sensitive matrix multiplication and addition routines are done in extra precision. A small computational cost is incurred. |

9.4.1.6. Solvent Description

temperature [298.15] Temperature in Kelvin.

dieps [] (Required.) Dielectric constant of the solvent.

nsp [] (Required.) Number of species (molecules) in the solutions. Also indicates the number of `species` name lists to follow.

9.4.1.7. Other

smear [1.0] Charge smear parameter in Å for long range asymptotics corrections.

adbcor [0.5] Numeric parameter for DRISM.

9.4.1.8. Deprecated Keywords

Several keywords from the previous release are deprecated but are still supported for backwards compatibility

closur Synonym for `closure`.

outlst Synonym for `outlist`.

routup Synonym for `rout`.

toutup Synonym for `kout`.

kshow Synonym for `progress`.

nis Synonym for `mdiis_nvec`.

delvv Synonym for `mdiis_del`.

tolvv Synonym for `tolerance`.

maxste Synonym for `maxstep`.

closureOrder When `closure`="PSEN" or "PSE" (no integers), `closureOrder` is used to specify the order of the PSE-*n* closure. E.g., you can use

```
closure='PSEN', closureOrder=3
instead of
closure='PSE3'
```

9.4.2. Example

Mixed ionic solvent.

```
&PARAMETERS
  THEORY='DRISM', CLOSURE='KH',           !Theory
  NR=16384, DR=0.025,                     !Grid size and spacing
  OUTLIST='x', ROUT=384, KOUT=0,          !Output
  MDIIS_NVEC=20, MDIIS_DEL=0.3, TOLERANCE=1.e-12, !MDIIS
  KSAVE=-1,                               !Check pointing
  PROGRESS=1,                             !Output frequency
  MAXSTEP=10000,                          !Maximum iterations
  SMEAR=1, ADCOR=0.5,                    !Electrostatics
  TEMPERATURE=310, DIEPS=78.497, NSP=3 !bulk solvent properties
/
&SPECIES
  !SPC/E water
  DENSITY=55.296d0,                      !very close to 0.0333 1/A3
  MODEL=" ../../dat/rismld/model/SPC.mdl"
/
&SPECIES
  !Sodium
  units='mM'
  DENSITY=100,
  MODEL=" ../../dat/rismld/model/Na+.mdl"
/
&SPECIES
  !Chloride
  units='g/cm^3'
  DENSITY=35.45e-4,
  MODEL=" ../../dat/rismld/model/Cl-.mdl"
/
```

9.5. 3D-RISM in NAB

3D-RISM functionality is available in NAB and is built as part of the standard install procedure. MPI functionality for 3D-RISM in NAB requires some additional information at compile time, described in Section 9.5.5. At this time, standard molecular dynamics and minimization with non-polarizable force fields are supported.

9.5.1. Solvation Box Size

The non-periodic solvation box super-cell can be defined as variable or fixed in size. When a variable box size is used, the box size will be adjusted to maintain a minimum buffer distance between the atoms of the solute and the box boundary. This has the advantage of maintaining

9. Reference Interaction Site Model

the smallest possible box size while adapting to changes of solute shape and orientation. Alternatively, the box size can be specified at run-time. This box size will be used for the duration of the calculation.

9.5.2. I/O

All 3D-RISM options, including input and output files, are specified using `mm_options()` (see Section 18.1). Generated output files can be quite large and numerous. For each type of correlation, a separate file is produced for each solvent atom type. The frequency that files are produced is controlled by the `ntwrism` parameter. For every time step that output is produced, a new set of files is written with the time step number in the file name. For example, a molecular dynamics calculation using an SPC/E water model with `ntwrism=2` and `guvfile=guv` will produce two files on time step ten: `guv.O.10.dx` and `guv.H1.10.dx`.

9.5.3. Examples

9.5.3.1. Molecular Dynamics

```
:
mm_options("ntpr=100, ntpr_md=100");
mm_options("dt=0.002");           //Large time step
mm_options("rattle=1");           //Use RATTLE
mm_options("cut=999.0");          //No solute-solute
                                   //cut off
mm_options("rism=1");             //Use 3D-RISM-KH
mm_options("xvffile=./rismld/spc/spc.xvv.save"); //1D-RISM input
:
```

9.5.3.2. Minimization

```
:
mm_options("ntpr=1, cut=999.0");  //No solute-solute
                                   //cut off
mm_options("rism=1");             //Use 3D-RISM-KH
mm_options("xvffile=./rismld/spc/spc.xvv.save"); //1D-RISM input
mm_options("tolerance=1e-11");    //Low tolerance
mm_options("solvcut=999.0");      //No solute-solvent
                                   //cut off
mm_options("centering=2");        //Center solute
                                   //using center-
                                   //of-geometry
:
```

9.5.4. Thermodynamic Output

When `nptrism` \neq 0 thermodynamic data about the solvent is output. This is presented as a table

solute_epot:	Total	LJ	Coulomb	Bond
	Angle	Dihedral	H-Bond	LJ-14
	Coulomb-14	Restraints	3D-RISM	

Solute internal energy [kcal/mol] and its components. This is written as a single line.

rism_exchem:	Total	ExChem_1	ExChem_2	...
---------------------	--------------	-----------------	-----------------	------------

Excess chemical potential (solvation free energy) [kcal/mol] for the closure used and the contribution from each solvent atom type.

rism_exchGF:	Total	ExChem_GF_1	ExChem_GF_2	...
---------------------	--------------	--------------------	--------------------	------------

Excess chemical potential (solvation free energy) [kcal/mol] using the Gaussian fluctuation approximation and the contribution from each solvent atom type.

rism_exEnUV:	Total	Energy_1	Energy_2	...
---------------------	--------------	-----------------	-----------------	------------

Average solute-solvent interaction energy [kcal/mol],

$$\Delta U_{\text{sol}} = \sum_{\alpha} \rho_{\alpha} \int d\mathbf{r} g_{\alpha}^{\text{UV}}(\mathbf{r}) u_{\alpha}^{\text{UV}}(\mathbf{r}),$$

and the contribution from each solvent atom type. Note that this is only a component of the solvation energy as it does not include changes in the solvent-solvent interaction energy[191].

rism_volume:	PMV
---------------------	------------

Partial molar volume of the solute [\AA^3].

rism_exNum:	ExNum_1	ExNum_2	...
--------------------	----------------	----------------	------------

Excess number of each atom type of solvent accumulated by the solute.

rism_exChrg:	Total	ExChg_1	ExChg_2	...
---------------------	--------------	----------------	----------------	------------

Excess charge [e] of each atom type of solvent accumulated by the solute.

rism_polar_:	Total	polar_1	polar_2	...
---------------------	--------------	----------------	----------------	------------

Solvent polarization contribution to the total excess chemical potential [kcal/mol] and the contribution from each solvent atom type. Only present when `polardecomp`=1.

rism_apolar:	Total	apolar_1	apolar_2	...
---------------------	--------------	-----------------	-----------------	------------

Cavity formation and dispersion contribution to the total excess chemical potential [kcal/mol] and the contribution from each solvent atom type. Only present when `polardecomp`=1.

9. Reference Interaction Site Model

```
rism_polGF:      Total  polarGF_1  polarGF_2      ...
```

Solvent polarization contribution to the Gaussian fluctuation total excess chemical potential [kcal/mol] and the contribution from each solvent atom type. Only present when polardecomp=1.

```
rism_apolGF:      Total  apolarGF_1  apolarGF_2      ...
```

Cavity formation and dispersion contribution to the Gaussian fluctuation total excess chemical potential [kcal/mol] and the contribution from each solvent atom type. Only present when polardecomp=1.

9.5.5. Compiling MPI 3D-RISM

Executables compiled with mpinab and 3D-RISM must link to both C and Fortran MPI libraries, which is not the default behaviour of most MPI compilers. As there are a wide variety of MPI implementations and no standards for naming Fortran libraries, 3D-RISM is not included by default when compiling mpinab. The additional steps required to include 3D-RISM in mpinab are

1. If

- a) you are using OpenMPI or MPICH2, proceed to step [2](#).
- b) you are *not* using OpenMPI or MPICH2, identify the Fortran 77 libraries corresponding to your MPI implementation. These will be found in the lib directory for your MPI implementation and will likely contain “f” or “f77” in the file name. Set the XTRA_FLIBS environment variable to contain the compiler directive to link the library.

For example, the OpenMPI and MPICH2 library files are libmpi_f77.a and libfpich.a respectively (the suffix may vary) and XTRA_FLIBS could be explicitly set as:

```
OpenMPI export XTRA_FLIBS=-lmpi_f77
MPICH2  export XTRA_FLIBS=-lfpich
```

2. Run configure and specify both -mpi and -rismmpi. For example:

```
./configure -mpi -rismmpi gnu
```

3. For dynamically linked executables (the default), set your LD_LIBRARY_PATH environment variable to the location of your MPI library:

```
export LD_LIBRARY_PATH=$MPIHOME/lib
```

\$MPIHOME is the base directory for you MPI installation.

where

9.6. rism3d.snglpnt

3D-RISM functionality is also available in the command line tools rism3d.snglpnt and rism3d.snglpnt.MPI installed at compile time. These programs perform single point 3D-RISM

calculations on trajectories and individual solute snapshots. No other processing is done to the structures so unwanted solvent molecules should be removed before hand. Except for minimization and molecular dynamics, all 3D-RISM features are available. Thermodynamic data is always output (see Section 9.5.4). Note that these executables are built by NAB so please see Section 9.5.5 on ensuring *rism3d.snglpnt.MPI* is built.

9.6.1. Usage

3D-RISM specific command line keywords generally correspond to keyword options available in NAB's *mm_options* (see Section 18.1). If run without input, *rism3d.snglpnt* prints default settings for all parameters.

- `--pdb PDB file` (Required, input.) PDB file for the solute. Coordinates are only used if a restart or trajectory file is not supplied.
- `--prmtop prmtop file` (Required, input.) Parameter topology file for the solute.
- `--rst restart file` (Optional, input.) Coordinates for the solute in restart format.
- `--nc NetCDF file` (Optional, input.) Trajectory for the solute in NetCDF format.
- `--xvv X^{VV} file` (Required, input.) Bulk solvent susceptibility file from 1D-RISM (see §9.7.2).
- `--guv G^{UV} root` (Optional, output.) Root name for 3D solvent pair distribution files.
- `--cuv C^{UV} root` (Optional, output.) Root name for 3D solvent direct correlation files.
- `--huv H^{UV} root` (Optional, output.) Root name for 3D solvent total correlation files.
- `--uuv U^{UV} root` (Optional, output.) Root name for 3D solvent potential [kT] files.
- `--asyp asymptotics root` (Optional, output.) Root name for 3D real-space long range asymptotics for total and direct correlation files. This will produce one file for each of C and H for each frame requested and does not include the solvent site charge. Multiply the distribution by the solvent site charge to obtain the long-range asymptotics for that site.
- `--quv Q^{UV} root` (Optional, output.) Root name for 3D solvent charge density distribution files. This is the charge density [$e/\text{\AA}$] at each grid point with contributions from all solvent types.
- `--chgdist charge distribution root` (Optional, output.) Root name for 3D solvent charge distribution files. This gives a point charge [e] at each grid point with contributions from all solvent types.
- `--volfmt` (Optional.) Format of volumetric data files. May be *dx* for DX files (see §9.7.6) or *xyzv* for XYZV format (see §9.7.7).
- `--closure closure name` (Optional.) One of KH, HNC or PSE_n where “ n ” is a positive integer.

9. Reference Interaction Site Model

- `--closureorder` *closure order* (Deprecated.) Specifies the order of the PSE-*n* closure if the closure name is given as “PSE” or “PSEN” (no integers).
- `--noasymptcorr` (Optional.) Turn off long range asymptotic corrections for thermodynamic output only. Long-range asymptotics are still used to calculate the solution.
- `--buffer` *distance* (Optional.) Minimum distance between the solute and the edge of the solvent box. Use this with `--grdspc`. Incompatible with `--ng` and `--solvbox`.
- `--solvcut` *distance* (Optional.) Set solute-solvent interaction cut off distance. If no value is specified then the buffer distance is used. If a buffer distance is not provided, the cut off must be explicitly set. Note that Coulomb interactions are interpolated and not truncated beyond the cut off. See [156] for details.
- `--grdspc` *3D grid spacing* (Optional.) Comma separated linear grid spacings for *x*, *y* and *z* dimensions. Use this with `--buffer`. Incompatible with `--ng` and `--solvbox`.
- `--ng` *3D grid points* (Optional.) Comma separated number of grid points for *x*, *y* and *z* dimensions. Use this with `--solvbox`. Incompatible with `--buffer` and `--grdspc`.
- `--solvbox` *3D box length* (Optional.) Comma separated solvation box side length for *x*, *y* and *z* dimensions. Use this with `--ng`. Incompatible with `--buffer` and `--grdspc`.
- `--tolerance` *residual target* (Optional.) Maximum residual value for solution convergence.
- `--mdiis_del` *step size* (Optional.) MDIIS step size.
- `--mdiis_nvec` *# of vectors* (Optional.) Number of previous iterations MDIIS uses to predict a new solution.
- `--maxstep` *step number* (Optional.) Maximum number of iterative steps per solution.
- `--npropagate` *# old solutions* (Optional.) Number of previous solutions to use in predicting a new solution.
- `--polarDecomp` (Optional.) Decomposes solvation free energy into polar and non-polar components. Note that this typically requires 80% more computation time.
- `--centering` *method* (Optional.) Select how solute is centered in the solvent box.
- 2** Center of geometry. Center on first step only.
 - 1** Center of mass. Center on first step only.
 - 0** No centering. Dangerous.
 - 1** Center of mass. Center on every step. Recommended for molecular dynamics.
 - 2** Center of geometry. Center on every step. Recommended for minimization.
- `--verbose` *level* (Optional.)
- 0** No output.
 - 1** Print the number of iterations required to converge.
 - 2** Print convergence details for each iteration.

9.7. File Formats

9.7.1. MDL

Solvent MoDeL (MDL) files use the prmtop specification. Each of the following sections may appear in the file in any order. The Fortran string format specifications can be different from the recommend values below.

```
%VERSION VERSION_STAMP = Vxxxx.yyy DATE = mm:dd:yy hh:mm:ss
```

The current version of the format is 0001.000. Date should be the date and time the file is created.

```
%FLAG TITLE
%FORMAT(20a4)
```

Optional description of the file.

```
%FLAG POINTERS
%FORMAT(10i8)
```

Defines the lengths of arrays in the file.

```
NATOM          Number of physical atoms in the model.
NSITE           Number of unique solvent sites (share common Lennard-Jones parameters and partial charges).
```

```
%FLAG ATMNAME
%FORMAT(20a4)
```

```
CHARACTER(len=4)(NSITE)  Four character name of each solvent site.
```

```
%FLAG MASS
%FORMAT(5e16.8)
```

```
REAL*8(NSITE)  Mass of each solvent site (amu).
```

```
%FLAG CHG
%FORMAT(5e16.8)
```

```
REAL*8(NSITE)  Partial charge for each solvent site,  $18.2223e (\sqrt{kT\text{\AA}})$ .
```

```
%FLAG LJEPSILON
%FORMAT(5e16.8)
```

```
REAL*8(NSITE)  Lennard-Jones  $\epsilon$  for each solvent site (kcal/mol).
```

9. Reference Interaction Site Model

%FLAG LJSIGMA

%FORMAT(5e16.8)

REAL*8 (NSITE) Lennard-Jones $r_{\min}/2$ (sometimes called $\sigma^*/2$) for each solvent site (\AA)

$$U_{\alpha\gamma}^{\text{LJ}} = \sqrt{\epsilon_{\alpha}\epsilon_{\gamma}} \left(\left(\frac{r_{\min,\alpha} + r_{\min,\gamma}}{2r} \right)^{12} - 2 \left(\frac{r_{\min,\alpha} + r_{\min,\gamma}}{2r} \right)^6 \right).$$

Note that this is related to the commonly used σ as

$$\sigma = r_{\min} 2^{-1/6}.$$

%FLAG MULTI

%FORMAT(10I8)

INTEGER*4 (NSITE) Multiplicity of each solvent site. This should sum to NATOM.

%FLAG COORD

%FORMAT(5e16.8)

REAL*8 (3*NATOM) xyz-coordinates of each atom (\AA).

9.7.2. XVV

The .xvv file provides all of the bulk-solvent information required for 3D-RISM. This includes information about the solvent model, thermodynamic state and the necessary correlation functions. .xvv files use the prmtop specification. Each of the following sections may appear in the file in any order. The format specifications can be different from the recommend values below.

1D- and 3D-RISM now use version 1.000 of the file format. Differences include

- additional information about solvent, such as mass, number of sites per species, coordinates;
- RISM's internal system of units is now used;
- temperature derivative, DELHVO_DT and XVV_DT, are included when available (see 9.4.1.1);
- and SIGV has been replaced by RMIN2V.

All 3D-RISM interfaces still support the original 0.001 version of the format. For detailed information on version 0.001, please see the AmberTools 1.5 manual.

%VERSION VERSION_STAMP = V0001.000 DATE = mm:dd:yy hh:mm:ss

The current version of the format is 0001.000. Date should be the date and time the file is created.

%FLAG POINTERS**%FORMAT(10I8)**

Defines the lengths of arrays in the file.

NR Number of 1D grid points in $\chi_{ab}^{VV}(k)$.

NV Number of total solvent sites.

NSP Number of solvent species (molecules).

%FLAG THERMO**%FORMAT(1PE24.16)**

REAL(8)(6) Temperature [K], dielectric constant, inverse Debye length (κ) [\AA], compressibility [\AA^{-3}], grid spacing [\AA], charge smear [\AA].

%FLAG ATOM_NAME**%FORMAT(20A4)**

CHARACTER(len=4)(NSITE) Four character name of each solvent site.

%FLAG MTV**%FORMAT(10I8)**

INTEGER(4)(NSITE) Multiplicity of each solvent site.

%FLAG NVSP**%FORMAT(10I8)**

INTEGER(4)(NSP) Number of sites for each solvent species.

%FLAG MASS**%FORMAT(1P5E16.8)**

REAL(8)(NSITE) Mass of each solvent site (g/mol).

%FLAG RHOV**%FORMAT(1P5E16.8)**

REAL(8)(NSITE) Number density of each solvent site (\AA^{-3}).

%FLAG QV**%FORMAT(1P5E16.8)**

REAL(8)(NSITE) Partial charge for each solvent site multiplied by the square root of the Coulomb constant, ~ 18.2223 ($\sqrt{kT\text{\AA}}$).

9. Reference Interaction Site Model

%FLAG QSPV

%FORMAT(1P5E16.8)

REAL(8) (NSPECIES) Net charge for each solvent species multiplied by the square root of the Coulomb constant, $\sim 18.2223 (\sqrt{kT\text{\AA}})$.

%FLAG EPSV

%FORMAT(1P5E16.8)

REAL(8) (NSITE) Lennard-Jones ϵ for each solvent site (kT).

%FLAG RMIN2V

%FORMAT(1P5E16.8)

REAL(8) (NSITE) Lennard-Jones $r_{\min}/2 (\sigma^*/2)$ for each solvent site (\AA).

%FLAG DELHV0

%FORMAT(1P5E16.8)

REAL(8) (NSITE) Long range Coulomb correction for each solvent site ($\sqrt{kT\text{\AA}}$).

%FLAG DELHV0_DT

%FORMAT(1P5E16.8)

REAL(8) (NSITE) (Optional) Temperature derivative long range Coulomb correction for each solvent site ($\sqrt{kT\text{\AA}}$).

%FLAG COORD

%FORMAT(1P5E16.8)

REAL(8) (3*sum(MTV)) Coordinates of all atoms (not sites) for each solvent species with the dipole moment aligned with the z-axis (\AA).

%FLAG XVV

%FORMAT(1P5E16.8)

REAL(8) (NR, NSITE, NSITE) $\chi_{ab}^{VV}(k)$. This array is stored in column major order. That is, the NR index varies fastest.

%FLAG XVV_DT

%FORMAT(1P5E16.8)

REAL(8) (NR, NSITE, NSITE) (Optional) $\delta_T \chi_{ab}^{VV}(k)$. This array is stored in column major order. That is, the NR index varies fastest.

9.7.3. Site-site functionals

All *.vv files, except .xvv (see §9.7.2), provide the separation dependence of all site-site pairings for a particular functional and use the same format. The first four lines have a “#”

in the first character column, provide a description of the contents of the file and indicate site-site pairs. The first data column is the site-site separation and the remaining columns provide the value of the functional for the site-site pair at this separation.

The following example is for the direct correlation function (.cvv) for pure water. A standard, 'two-site' water model is used, consisting of oxygen (O) and hydrogen (H1). This gives one solvent species with two atoms.

```
#RISMLD ATOM-ATOM INTERACTIONS: DIRECT CORRELATION VS. SEPARATION [A]
#S=SPECIES, A=ATOM
# SEPARATION S1A1:S1A1 S1A1:S1A2 S1A2:S1A2
# SEPARATION O:O H1:O H1:H1
0.00000000E+000 -3.81875841E+002 1.64156197E+002 -9.24562553E+001
2.50000000E-002 -3.81695327E+002 1.64139031E+002 -9.24384608E+001
```

9.7.4. Thermodynamics

Thermodynamic output is divided into global, species and site properties sections. Global properties are generally not decomposable into species or site contributions (e.g., pressure). Species properties are the values for individual molecular species, for example, the excess chemical potential of a single molecule. Some of these properties, such as the partial molar volume, may not be decomposable into individual sites. Site properties are contributions from individual sites. Values for sites from the same species will sum to give the total value for the species.

The file format is white-space delimited with the first three columns giving a description, variable name and units of the property calculated. The remaining columns contain the calculated values for the system, species or site. Descriptive lines are indicated with a leading "#".

The following example is for a standard, 'two-site' water model is used, consisting of oxygen (O) and hydrogen (H1), at standard temperature and density. In this calculation, energy/entropy free energy decomposition is also performed. I.e.,

$$\text{EXCHEM}_{\text{sp}} = \text{ESOLV}_{\text{sp}} - \text{TS}_{\text{sp}}.$$

```
#Global properties
#Description Variable Units Value
Compressibility xi [10e-4/MPa] 4.73552130E+000
Pressure_(Virial) Pvir [MPa] 2.51627507E+003
Excess_free_energy FE [kcal/mol] -1.03698038E+003
#Species properties
#Description Variable Units SPC
Excess_chemical_potential EXCHEMsp [kcal/mol] -2.79190339E+000
Solvation_energy ESOLVsp [kcal/mol] -1.16421825E+001
-Temperature*solvation_entropy -TSsp [kcal/mol] 8.85027911E+000
Partial_molar_volume PMV [A^-3] 3.00300236E+001
#Site properties
#Description Variable Units O H1
Excess_chemical_potential EXCHEMv [kcal/mol] -6.47897321E+000 3.68706981E+000
Solvation_energy ESOLVv [kcal/mol] -1.19565867E+001 3.14404192E-001
-Temperature*solvation_entropy -TSv [kcal/mol] 5.47761350E+000 3.37266562E+000
```

9.7.5. Total excess values

.n00 and .q00 files provide the total excess coordination number and charge about each solvent site. The total excess of site γ around site α is

$$n_{\alpha\gamma}^{\text{extot}} = \rho_{\gamma} \int_0^{\infty} h_{\alpha\gamma}(r) dr,$$

while the total excess charge is

$$q_{\alpha\gamma}^{\text{extot}} = q_{\gamma} n_{\alpha\gamma}^{\text{extot}}.$$

These values are presented in their respective files as $n_{\text{site}} \times n_{\text{site}}$ arrays. Any asymmetry in these arrays is due to numerical error. .q00 files additionally provided the total excess charge from all sites.

The following example gives the total excess charge for a standard, ‘two-site’ water model is used, consisting of oxygen (O) and hydrogen (H1), at standard temperature and density.

```
#Total excess coordinated charge [e] of column site about row site
      O          H1          Total charge
O      7.92607232E-001 -7.92607230E-001  1.67181313E-009
H1     7.92607231E-001 -7.92607229E-001  2.44386922E-009
```

9.7.6. DX volumetric data

By default, 3D correlation functions from 3D-RISM calculations use the ASCII version of the Data Explorer (DX) file format for volumetric data on regular grids as defined in the DX user manual: <http://opendx.informatics.jax.org/docs/html/pages/usrgu068.htm#HDREDF>.

Header

```
object 1 class gridpositions counts Nx Ny Nz

Nx          INTEGER*4. Number of grid points in the x dimension.

Ny          INTEGER*4. Number of grid points in the y dimension.

Nz          INTEGER*4. Number of grid points in the z dimension.

origin Ox Oy Oz

Ox          REAL*8. x coordinate of grid origin in Cartesian space.

Oy          REAL*8. y coordinate of grid origin in Cartesian space.

Oz          REAL*8. z coordinate of grid origin in Cartesian space.
```



```

delta dx 0 0
delta 0 dy 0
delta 0 0 dz

```

dx REAL*8. Linear grid size between in the x dimension.

dy REAL*8. Linear grid size between in the y dimension.

dz REAL*8. Linear grid size between in the z dimension.

```

object 2 class gridconnections counts Nx Ny Nz
object 3 class array type double rank 0 items N data follows

```

N INTEGER*4. $N = N_x \times N_y \times N_z$.

Data

```

data(i,j,k) data(i,j,k+1) data(i,j,k+2)

```

data(i,j,k) REAL*8. Three data values per line with the last (z) index varying fastest for a total of N values.

Footer

```

object "Untitled" call field

```

9.7.7. XYZV volumetric data

An alternate format for volumetric data is the simple ASCII x - y - z -value (XYZV) format. The x -, y - and z -coordinates each grid point is written on a line followed by the value of the grid point. There is no header or footer. For example,

```

:
:
-7.10789855E+000 -1.12570084E+001 -1.61284113E+001 1.35771922E-006
-2.10789855E+000 -1.12570084E+001 -1.61284113E+001 -5.32279347E-006
2.89210145E+000 -1.12570084E+001 -1.61284113E+001 -1.58802759E-005
:
:

```


10. MMPBSA.py

Note: Amber now has three(!) scripts to carry out MM-PBSA-like calculations. The one described here (the “python” version) is more recent, generally simpler to use, and has a more active support community for answering questions. The *amberlite* code (described in Chapter 5) is more limited, and focussed on protein-ligand interactions; it is a great place for users new to AmberTools to begin. The version described in the Amber12 manual (the “perl” version) continues to be updated, and has some specialized features. Most new users should try the python or amberlite versions first.

None of these should be considered as a “black-box”, and users should be familiar with Amber before attempting these sorts of calculations. These scripts automate a series of calculations, and cannot trap all the types of errors that might occur. You should be sure that you know how to carry out an MM-PBSA calculation “by hand” (*i.e.*, without using the scripts); if you don’t understand in detail what is going on, you will have no good reason to trust the results.

10.1. Introduction

This section describes the use of the python script MMPBSA.py to perform Molecular Mechanics / Poisson Boltzmann (or Generalized Born) Surface Area (MM/PB(GB)SA) calculations. This is a post-processing method in which representative snapshots from an ensemble of conformations are used to calculate the free energy change between two states (typically a bound and free state of a receptor and ligand). Free energy differences are calculated by combining the so-called gas phase energy contributions that are independent of the chosen solvent model as well as solvation free energy components (both polar and non-polar) calculated from an implicit solvent model for each species. Entropy contributions to the total free energy may be added as a further refinement. The entropy calculations can be done in either a HCT Generalized Born solvation model [192, 193] or in the gas phase using a *mmpbsa_py_nabnmode* program written in the *nab* programming language, or via the quasi-harmonic approximation in *ptraj*.

The gas phase free energy contributions are calculated by *sander* within the Amber program suite according to the force field with which the topology files were created. The solvation free energy contributions may be further decomposed into an electrostatic and hydrophobic contribution. The electrostatic portion is calculated using either the Poisson Boltzmann (PB) equation or by the Generalized Born method. The PB equation is solved numerically by either the *pbsa* program included with AmberTools or by the Adaptive Poisson Boltzmann Solver (APBS) program through the iAPBS interface with Amber (for more information, see <http://www.poissonboltzmann.org/apbs>). The hydrophobic contribution is approximated by the LCPO method [129] implemented within *sander*.

MM/PB(GB)SA typically employs the approximation that the configurational space explored

by the systems are very similar between the bound and unbound states, so every snapshot for each species is extracted from the same trajectory file, although MMPBSA.py will accept separate trajectory files for each species. Furthermore, explicit solvent and ions are stripped from the trajectory file(s) to hasten convergence by preventing solvent-solvent interactions from dominating the energy terms. A more detailed explanation of the theory can be found in Srinivasan, et. al.[80] You may also wish to refer to reviews summarizing many of the applications of this model,[82, 84] as well as to papers describing some of its applications.[85–89]

Many popular types of MM/PBSA calculations can be performed using just AmberTools, while some of the more advanced functionality requires the *sander* program from Amber.

10.2. Preparing for an MM/PB(GB)SA calculation

MM/PB(GB)SA is often a very useful tool for obtaining relative free energies of binding when comparing ligands. Perhaps its biggest advantage is that it is very computationally inexpensive compared to other free energy calculations, such as TI or FEP. Following the advice given below before any MD simulations are run will make running MMPBSA.py successfully much easier.

10.2.1. Building Topology Files

MMPBSA.py requires at least three, usually four, compatible topology files. If you plan on running MD in explicit water, you will need a solvated topology file of the entire complex, and you will always need a topology for the entire complex, one for just the receptor, and a final one for just the ligand. Moreover, they must be compatible with one another (i.e., each must have the same charges for the same atoms, the same force field must be used for all three of the required prmtops, and they must have the same PBRadii set, see LEaP for description of pbradii). Thus, it is strongly advised that all prmtop files are created with the same script. We run through a typical example here, though leave some of the details to other sections and other tutorials. We will start with a system that is a large protein binding a small, one-residue ligand. We will assume that a docked structure has already been obtained as a PDB and that two separate PDBs have been constructed, receptor.pdb and LIG.pdb. We will also assume that a MOL2 file was created from LIG.pdb, residue name 'LIG', was built with charges already derived (either through antechamber or some other method), and an frcmod file for 'LIG' that contains all missing parameters have already been created. Furthermore, we will use the FF99SB force field for this example. A sample script file called, for instance, mmpbsa_leap.in, is shown below

```
source leaprc.ff99SB
loadAmberParams LIG.frcmod
LIG = loadMol2 LIG.mol2
receptor = loadPDB receptor.pdb
complex = combine {receptor LIG}
set default PBRadii mbondi2

saveAmberParm LIG lig.top lig.crd
```

```

saveAmberParm receptor rec.top rec.crd
saveAmberParm complex com.top com.crd

solvateOct complex TIP3PBOX 15.0
saveAmberParm complex com_solvated.top com_solvated.crd
quit

```

The above script, run with the command

```
tleap -f mmpbsa_leap.in
```

should produce four prmtop files, lig.top, rec.top, com.top, and com_solvated.top. Topology files created in this manner will make running MMPBSA.py far easier. This is, of course, the simplest case, but we briefly describe some other examples. MMPBSA.py will guess the mask for both the receptor and ligand inside the complex topology file as long as the ligand residues appear continuously in the complex topology file. Therefore, if you're adding two ligands, combine them consecutively in the complex (rather than one residue at the beginning and one at the end, for instance). If you have done this, you should allow MMPBSA.py to guess the masks since it provides a good error check.

10.2.2. Using ante-MMPBSA.py

ante-MMPBSA.py is a python utility that allows you to create compatible complex, receptor, and ligand topology files from a solvated topology file, or compatible receptor and ligand topology files from a complex topology file. The usage statement for ante-MMPBSA.py is

```

Usage: ante-MMPBSA.py [options]
-h, --help          show this help message and exit
-p PRMTOP, --prmtop=PRMTOP
                    Input "dry" complex topology or solvated complex
                    topology
-c COMPLEX, --complex-prmtop=COMPLEX
                    Complex topology file created by stripping PRMTOP
                    of solvent
-r RECEPTOR, --receptor-prmtop=RECEPTOR
                    Receptor topology file created by stripping
                    COMPLEX of ligand
-l LIGAND, --ligand-prmtop=LIGAND
                    Ligand topology file created by stripping COMPLEX
                    of receptor
-s STRIP_MASK, --strip-mask=STRIP_MASK
                    Amber mask of atoms needed to be stripped from
                    PRMTOP to make the COMPLEX topology file
-m RECEPTOR_MASK, --receptor-mask=RECEPTOR_MASK
                    Amber mask of atoms needed to be stripped from
                    COMPLEX to create RECEPTOR. Cannot specify with
                    -n/--ligand-mask

```

10. MMPBSA.py

```
-n LIGAND_MASK, --ligand-mask=LIGAND_MASK  
    Amber mask of atoms needed to be stripped from  
    COMPLEX to create LIGAND. Cannot specify with  
    -m/--receptor-mask
```

The input prmtop is required. It can either be a solvated, complex topology file or a complex topology file with no solvent present. If a strip_mask is given, you must also provide a complex topology file, and that complex topology file will be created by stripping strip_mask from the input prmtop. If you wish to create receptor and ligand topology files (you must create both or neither), provide BOTH a -receptor-prmtop and a -ligand-prmtop file name, as well as only ONE of either -receptor-mask or -ligand-mask. Whichever mask you do NOT define will be defined as the negated mask that you DID provide.

In short, you can use ante-MMPBSA.py to strip solvent from your prmtop for 3 applications.

1. Strip solvent from a solvated topology file and write out a non-solvated topology file.
2. Create ligand and receptor topologies from a complex topology by removing a given ligand or receptor mask.
3. A combination of 1 and 2 in the same command.

10.2.3. Running Molecular Dynamics

Not many details will be given here because MM/PB(GB)SA is a post-processing trajectory analysis technique. Molecular dynamics are run to generate an ensemble of snapshots upon which to calculate the binding energy. This technique is most effective when the structures are not correlated, which means that the simulated time between extracted snapshots should be sufficiently large to avoid such correlation.

There are two techniques that can be employed when running these simulations with respect to MMPBSA.py. The first is what's called the "single trajectory protocol" and the second of which is called the "multiple trajectory protocol". The first method will extract the snapshots for the complex, receptor, and ligand from the same trajectory. This is a faster method because it requires the simulation of only a single system, but makes the assumption that the configurational space explored by the receptor and ligand is unchanged between the bound and unbound states. The latter method eliminates this assumption at the cost of more simulations. MMPBSA.py requires a complex trajectory, but will accept a receptor and/or ligand trajectory as well. Any trajectory not given to the script will be extracted from the complex trajectory.

10.3. Running MMPBSA.py

10.3.1. The input file

The input file was designed to be as syntactically similar to other programs in Amber as possible. The input file has the same namelist structure as both sander and pmemd. The allowed namelists are &general, &gb, &pb, &crism, &alanine_scanning, &nmode, and &decomp. The input variables recognized in each namelist are described below, but those in &general are

typically variables that apply to all aspects of the calculation. The `&gb` namelist is unique to Generalized Born calculations, `&pb` is unique to Poisson Boltzmann calculations, `&rism` is unique to 3D-RISM calculations, `&alanine_scanning` is unique to alanine scanning calculations, `&nmode` is unique to the normal mode calculations used to approximate vibrational entropies, and `&decomp` is unique to the decomposition scheme. All of the input variables are described below according to their respective namelists. Integers and floating point variables should be typed as-is while strings should be put in either single- or double-quotes. All variables should be set with “variable = value” and separated by commas. See the examples below. Variables will be matched to the minimum number of characters greater than or equal to 4 required to uniquely identify that variable within that namelist. For example, “star” in `&general` will match “startframe”. However, “stare” will match nothing.

&general namelist variables

debug_printlevel MMPBSA.py prints errors by raising exceptions, and not catching fatal errors. If `debug_printlevel` is set to 0, then detailed tracebacks (effectively the call stack showing exactly where in the program the error occurred) is suppressed, so only the error message is printed. If `debug_printlevel` is set to 1 or higher, all tracebacks are printed, which aids in debugging of issues. Default: 0. (Advanced Option)

endframe The frame from which to stop extracting snapshots from the full, concatenated trajectory comprised of every trajectory file supplied on the command-line. (Default = 1000000000)

entropy Specifies whether or not a quasi-harmonic entropy approximation is made with ptraj. Allowed values are 0: Don't. 1: Do (Default = 0)

interval The offset from which to choose frames from each trajectory file. For example, an interval of 2 will pull every 2nd frame beginning at startframe and ending less than or equal to endframe. (Default = 1)

keep_files The variable that specifies which temporary files are kept. All temporary files have the prefix “_MMPBSA_” prepended to them. Allowed values are 0, 1, and 2. 0: Keep no temporary files, 1: Keep all generated trajectory files and mdout files created by sander simulations, 2: Keep all temporary files. Temporary files are only deleted if MMPBSA.py completes successfully. (Default = 1) A verbose level of 1 is sufficient to use -rewrite-output and recreate the output file without rerunning any simulations.

ligand_mask The mask that specifies the ligand residues within the complex prmtop (NOT the solvated prmtop if there is one). The default guess is generally sufficient and will only fail when stated above. You should use the default mask assignment if possible because it provides a good error catch. This follows the same description as the receptor_mask above.

netcdf Specifies whether or not to use netcdf trajectories internally rather than writing temporary ASCII trajectory files. NOTE: netcdf trajectories can be used as input for MMPBSA.py regardless of what this variable is set to, but netcdf trajectories are faster

10. MMPBSA.py

to write and read. For very large trajectories, this could offer significant speedups, and requires less temporary space. However, this option is incompatible with alanine scanning. Allowed values are 0: Do NOT use temporary netcdf trajectories, 1: Use temporary netcdf trajectories. (Default = 0)

receptor_mask The mask that specifies the receptor residues within the complex prmtop (NOT the solvated prmtop if there is one). The default guess is generally sufficient and will only fail if the ligand residues are not found in succession within the complex prmtop. You should use the default mask assignment if possible because it provides a good error catch. It uses the “Amber mask” syntax described elsewhere in this manual. This will be replaced with the default receptor_mask if ligand_mask (below) is not also set.

search_path Advanced option. By default, MMPBSA.py will only search for executables in \$AMBERHOME/bin. To enable it to search for binaries in your full PATH if they can't be found in \$AMBERHOME/bin, set search_path to 1. Default 0 (do not search through the PATH).

startframe The frame from which to begin extracting snapshots from the full, concatenated trajectory comprised of every trajectory file placed on the command-line. This is always the first frame read. (Default = 1)

strip_mask The variable that specifies which atoms are stripped from the trajectory file if a *solvated_prmtop* is provided on the command-line. See 10.3.2. (Default = “:WAT:Cl:CIO:Cs+:IB:K+:Li+:MG2:Na+:Rb+”)

use_sander Forces MMPBSA.py to use *sander* for energy calculations, even when *mmpbsa_py_energy* will suffice. 0 - Use *mmpbsa_py_energy* when possible. 1 - Always use *sander*. (Default 0)

verbose The variable that specifies how much output is printed in the output file. There are three allowed values: 0, 1, and 2. A value of 0 will simply print difference terms, 1 will print all complex, receptor, and ligand terms, and 2 will also print bonded terms if one trajectory is used. (Default = 1)

&gb namelist variables (More thorough descriptions of each can be found in the Amber manual)

ifqnt Specifies whether a part of the system is treated with quantum mechanics. 1: Use QM/MM, 0: Potential function is strictly classical (Default = 0). This functionality requires *sander*

igb Generalized Born method to use. See the description in the Amber manual. Allowed values are 1, 2, 5, 7 and 8. (Default = 5) Only models 1, 2, and 5 are available through *mmpbsa_py_energy*. Selecting 7 or 8 will require *sander*

qm_residues Comma- or semicolon-delimited list of complex residues to treat with quantum mechanics. All whitespace is ignored. All residues treated with quantum mechanics in the complex must be treated with quantum mechanics in the receptor or ligand to

obtain meaningful results. If the default masks are used, then MMPBSA.py will figure out which residues should be treated with QM in the receptor and ligand. Otherwise, skeleton mdin files will be created and you will have to manually enter qmmask in the ligand and receptor topology files. There is no default, this must be specified.

qm_theory Which semi-empirical Hamiltonian should be used for the quantum calculation. No default, this must be specified. See its description in the QM/MM section of the manual for options.

qmcharge_com The charge of the quantum section for the complex. See the description of *qmcharge* in the AmberTools manual. (Default = 0)

qmcharge_lig The charge of the quantum section of the ligand. (Default = 0)

qmcharge_rec The charge of the quantum section for the receptor. (Default = 0)

qmcut The cutoff for the qm/mm charge interactions. See the description in the AmberTools manual. (Default = 9999.0)

saltcon Salt concentration in Molarity. (Default = 0.0)

surfoff Offset to correct (by addition) the value of the non-polar contribution to the solvation free energy term (Default *sander* value, which depends on GB model used)

surften Surface tension value (Default = 0.0072)

&pb namelist variables (More thorough descriptions of each can be found in the AmberTools manual)

cavity_offset Offset value used to correct non-polar free energy contribution (Default = -0.5692) This is not used for APBS.

cavity_surften Surface tension. (Default = 0.00378 kcal/mol Angstrom²). Unit conversion to *kJ* done automatically for APBS.

exdi External dielectric constant (Default = 80.0)

fillratio The ratio between the longest dimension of the rectangular finite-difference grid and that of the solute (Default = 4.0)

indi Internal dielectric constant (Default = 1.0)

inp Nonpolar optimization method (Default = 2)

istrng Ionic strength in Molarity. It is converted to mM for PBSA and kept as M for APBS. (Default = 0.0)

linit Maximum number of iterations of the linear Poisson Boltzmann equation to try (Default = 1000)

prbrad Solvent probe radius in Angstroms. Allowed values are 1.4 and 1.6 (Default = 1.4)

10. MMPBSA.py

radiopt The option to set up atomic radii according to 0: the prmtop, or 1: pre-computed values (see Amber manual for more complete description). (Default = 1)

sander_apbs Option to use APBS for PB calculation instead of the built-in PBSA solver. This will work only through the iAPBS interface built into sander.APBS. Instructions for this can be found online at the iAPBS/APBS websites. Allowed values are 0: Don't use APBS, or 1: Use sander.APBS. (Default = 0)

scale Resolution of the Poisson Boltzmann grid. It is equal to the reciprocal of the grid spacing. (Default = 2.0)

&alanine_scanning namelist variables

mutant_only Option to perform specified calculations only for the mutants. Allowed values are 0: Do mutant and original or 1: Do mutant only (Default = 0)

Note that all calculation details are controlled in the other namelists, though for alanine scanning to be performed, the namelist must be included (blank if desired)

&nmode namelist variables

dielc Distance-dependent dielectric constant (Default = 1.0)

drms Convergence criteria for minimized energy gradient. (Default = 0.001)

maxcyc Maximum number of minimization cycles to use per snapshot in sander. (Default = 10000)

nminterval* Offset from which to choose frames to perform nmode calculations on (Default = 1)

nmendframe* Frame number to stop performing nmode calculations on (Default = 1000000000)

nmode_igb Value for Generalized Born model to be used in calculations. Options are 0: Vacuum, 1: HCT GB model [192, 193] (Default 1)

nmode_istrng Ionic strength to use in nmode calculations. Units are Molarity. Non-zero values are ignored if *nmode_igb* is 0 above.

nmstartframe* Frame number to begin performing nmode calculations on (Default = 1)

* These variables will choose a subset of the frames chosen from the variables in the &general namelist. Thus, the “trajectory” from which snapshots will be chosen for nmode calculations will be the collection of snapshots upon which the other calculations were performed.

&decomp namelist variables

csv_format Print the decomposition output in a Comma-Separated-Variable (CSV) file. CSV files open natively in most spreadsheets. If set to 1, this variable will cause the data to be written out in a CSV file, and standard error of the mean will be calculated and included for all data. If set to 0, the standard, ASCII format will be used for the output file. Default is 1 (CSV-formatted output file)

dec_verbose Set the level of output to print in the decomp_output file.

- 0 - DELTA energy, total contribution only
- 1 - DELTA energy, total, sidechain, and backbone contributions
- 2 - Complex, Receptor, Ligand, and DELTA energies, total contribution only
- 3 - Complex, Receptor, Ligand, and DELTA energies, total, sidechain, and backbone contributions

Note: If the values 0 or 2 are chosen, only the Total contributions are required, so only those will be printed to the mdout files to cut down on the size of the mdout files and the time required to parse them. However, this means that -rewrite-output cannot be used to change the default verbosity to print out sidechain and/or backbone energies, but it can be used to reduce the amount of information printed to the final output. The parser will extract as much information from the mdout files as it can, but will complain and quit if it cannot find everything it's being asked for.

Default = 0

idecomp Energy decomposition scheme to use:

- 1 - Per-residue decomp with 1-4 terms added to internal potential terms
- 2 - Per-residue decomp with 1-4 EEL added to EEL and 1-4 VDW added to VDW potential terms.
- 3 - Pairwise decomp with 1-4 terms added to internal potential terms
- 4 - Pairwise decomp with 1-4 EEL added to EEL and 1-4 VDW added to VDW potential terms

(No default. This must be specified!) This functionality requires *sander*.

print_res Select residues from the complex prmtop to print. The receptor/ligand residues will be automatically figured out if the default mask assignments are used. If you specify your own masks, you will need to modify the mdin files created by MMPBSA.py and rerun MMPBSA.py with the -use-mdins flag. Note that the DELTAs will not be computed in this case. This variable accepts a sequence of individual residues and/or ranges. The different fields must be either comma- or semicolon-delimited. For example: print_res = "1, 3-10, 15, 100", or print_res = "1; 3-10; 15; 100". Both of these will print residues 1, 3 through 10, 15, and 100 from the complex prmtop and the corresponding residues in either the ligand and/or receptor prmtops. (Default: print all residues)*

* Please note: Using idecomp=3 or 4 (pairwise) with a very large number of printed residues and a large number of frames can quickly create very, very large temporary mdout files. Large print selections also demand a large amount of memory to parse the mdout files and write decomposition output file (~500 MB for just 250 residues, since that's 62500 pairs!) It is not

unusual for the output file to take a significant amount of time to print if you have a lot of data. This is most applicable to pairwise decomp, since the amount of data scales as $O(N^2)$.

&rism namelist variables*

buffer Minimum distance between solute and edge of solvation box. Specify this with `grdspc` below. Mutually exclusive with `ng` and `solvbox`. Set `buffer < 0` if you wish to use `ng` and `solvbox`. (Default = 14 Å)

closure The approximation to the closure relation. Allowed choices are *kh* (Kovalenko-Hirata), *hnc* (Hypernetted-chain), or *pse_n* (Partial Series Expansion of order-*n*) where “*n*” is a positive integer (e.g., “*pse3*”). (Default = ‘kh’)

closureorder (Deprecated) The order at which the PSE-*n* closure is truncated if closure is specified as “*pse*” or “*pse_n*” (no integers). (Default = 1)

grdspc Grid spacing of the solvation box. Specify this with `buffer` above. Mutually exclusive with `ng` and `solvbox`. (Default = 0.5 Å)

ng Number of grid points to use in the x, y, and z directions. Used only if `buffer < 0`. Mutually exclusive with `buffer` and `grdspc` above, and paired with `solvbox` below. No default, this must be set if `buffer < 0`. Define like “`ng=1000,1000,1000`”

polardecomp Decompose the solvation free energy into polar and non-polar contributions. Note that this will increase computation time by roughly 80%. 0: Don’t decompose solvation free energy. 1: Decompose solvation free energy. (Default = 0)

rism_verbose Level of output in temporary RISM output files. May be helpful for debugging or following convergence. Allowed values are 0 (just print the final result), 1 (additionally prints the total number of iterations for each solution), and 2 (additionally prints the residual for each iteration and details of the MDIIS solver). (Default = 0)

solvbox Length of the solvation box in the x, y, and z dimensions. Used only if `buffer < 0`. Mutually exclusive with `buffer` and `grdspc` above, and paired with `ng` above. No default, this must be set if `buffer < 0`. Define like “`solvbox=20,20,20`”

solvcut Cutoff used for solute-solvent interactions. The default is the value of `buffer`. Therefore, if you set `buffer < 0` and specify `ng` and `solvbox` instead, you must set `solvcut` to a non-zero value or the program will quit in error. (Default = `buffer`)

thermo Which thermodynamic equation you want to use to calculate solvation properties. Options are “std”, “gf”, or “both” (case-INsensitive). “std” uses the standard closure relation, “gf” uses the Gaussian Fluctuation approximation, and “both” will print out separate sections for both. (Default = “std”). Note that all data are printed out for each RISM simulation, so no choice is any more computationally demanding than another. Also, you can change this option and use the `-rewrite-output` flag to obtain a different printout after-the-fact.

tolerance Upper bound of the precision requirement used to determine convergence of the self-consistent solution. This has a strong effect on the cost of 3D-RISM calculations. (Default = 1e-5).

* 3D-RISM calculations are performed with the `rism3d.snglpnt` program built with AmberTools, written by Tyler Luchko. It is the most expensive, yet most statistically mechanically rigorous, solvation model available in MMPBSA.py. See the section about RISM in the AmberTools manual for a more thorough description of options and theory. A list of references can be found there, too. One advantage of 3D-RISM is that an arbitrary solvent can be chosen; you just need to change the `xvfile` specified on the command line (see [10.3.2](#)).

Sample input files

Sample input file for GB and PB calculation

```
&general
    startframe=5, endframe=100, interval=5,
    verbose=2, keep_files=0,
/
&gb
    igb=5, saltcon=0.150,
/
&pb
    istrng=0.15, fillratio=4.0
/
```

Sample input file for Alanine scanning

```
&general
    verbose=2,
/
&gb
    igb=2, saltcon=0.10
/
&alanine_scanning
/
```

Sample input file with nmode analysis

```
&general
    startframe=5, endframe=100, interval=5,
    verbose=2, keep_files=2,
/
&gb
    igb=5, saltcon=0.150,
/
&nmode
    nmstartframe=2, nmendframe=20, nminterval=2,
```

10. MMPBSA.py

```
maxcyc=50000, drms=0.0001,  
/  
-----  
Sample input file with decomposition analysis  
&general  
    startframe=5, endframe=100, interval=5,  
/  
&gb  
    igb=5, saltcon=0.150,  
/  
&decomp  
    idecomp=2, dec_verbose=3,  
    print_res="20, 40-80, 200"  
/  
-----  
Sample input file for QM/MMGBSA  
&general  
    startframe=5, endframe=100, interval=5,  
    ifqnt=1, qmcharge=0, qm_residues="100-105, 200"  
    qm_theory="PM3"  
/  
&gb  
    igb=5, saltcon=0.100,  
/  
-----  
Sample input file for MM/3D-RISM  
&general  
    startframe=5, endframe=100, interval=5,  
/  
&rism  
    polardecomp=1, thermo='gf'  
/  
-----
```

A few important notes about input files. Comments are allowed by placing a # at the beginning of the line (whitespace is ignored). Variable initialization may span multiple lines. In-line comments (i.e., putting a # for a comment after a variable is initialized in the same line) is not allowed and will result in an input error. Variable declarations must be comma-delimited, though all whitespace is ignored. Finally, all lines between namelists are ignored, so comments may be put before each namelist without using #.

10.3.2. Calling MMPBSA.py from the command-line

MMPBSA.py is invoked through the command line as follows:

```
Usage: MMPBSA.py [Options]  
Options:
```

```

--help, -h, --h, -H
    show this help message and exit
-O
    Overwrite existing output files
-i
    input_file
    MM/PBSA input file
-o
    output_file
    Final MM/PBSA statistics file. Default
    FINAL_RESULTS_MMPBSA.dat
-sp
    solvated_prmtop
    Solvated complex topology file
-cp
    complex_prmtop
    Complex topology file. Default "complex_prmtop"
-rp
    receptor_prmtop
    Receptor topology file
-lp
    ligand_prmtop
    Ligand topology file
-y
    mdcrd1,mdcrd2,...,mdcrdN
    Input trajectories to analyze. Default mdcrd
-do
    decompout
    Decomposition statistics summary file. Default
    FINAL_DECOMP_MMPBSA.dat
-eo
    energyout
    CSV-format output of all energy terms for every frame in
    every calculation. File name forced to end in .csv
-deo
    dec_energies
    CSV-format output of all decomposition energy terms for
    every frame. File name forced to end in .csv
-yr
    receptor_mdcrd1,receptor_mdcrd2,...,receptor_mdcrdN
    Receptor trajectory file for multiple trajectory approach
-yl
    ligand_mdcrd1,ligand_mdcrd2,...,ligand_mdcrdN
    Ligand trajectory file for multiple trajectory approach
-mc
    mutant_complex_prmtop
    Alanine scanning mutant complex topology file
-ml
    mutant_ligand_prmtop
    Alanine scanning mutant ligand topology file
-mr
    mutant_receptor_prmtop
    Alanine scanning mutant receptor topology file
-slp
    solvated_ligand_prmtop
    Solvated ligand topology file
-srp
    solvated_receptor_prmtop
    Solvated receptor topology file
-xvffile xvffile
    XVV file for 3D-RISM. Default
    $AMBERHOME/dat/mmpbsa/spc.xvv

```

10. MMPBSA.py

```
-make-mdins
    Create the Input files for each calculation and quit
-use-mdins
    Use existing input files for each calculation
-rewrite-output
    Don't rerun any calculations, just parse existing output
    files
--clean
    Clean temporary files from previous run
```

-make-mdins and -use-mdins are created to give added flexibility to user input. If the MM/PBSA input file does not expose a variable you require, you may use the -make-mdins flag to generate the MDIN files and then quit. Then, edit those MDIN files, changing the variables you need to, then running MMPBSA.py with -use-mdins to use those modified files.

-clean will remove all temporary files created by MMPBSA.py in a previous calculation.

-version will display the program version and exit.

10.3.3. Running MMPBSA.py

10.3.3.1. Serial version

This version is installed with Amber during the serial install of AmberTools. AMBERHOME must be set, or it will quit on error. If any changes are made to the modules, MMPBSA.py must be remade so the updated modules are found by MMPBSA.py. Note a change with the previous version of MMPBSA.py released with AmberTools 1.5: MMPBSA.py is now called directly. An example command-line call is shown below:

```
MMPBSA.py -O -i mmpbsa.in -cp com.top -rp rec.top -lp lig.top -y traj.crd
```

The tests, found in \${AMBERHOME}/test/mmpbsa_py provide good examples for running MMPBSA.py calculations.

10.3.3.2. Parallel (MPI) version

This version is installed with Amber during the parallel install. The python package mpi4py is included with the MMPBSA.py source code and must be successfully installed in order to run the MPI version of MMPBSA. It is run in the same way that the serial version is above, except MPI directions must be given on the command line as well. There are two common issues that may break parallel execution. The first is that mpi4py is not properly installed. One note: at a certain level, running RISM in parallel may actually hurt performance, since previous solutions are used as an initial guess for the next frame, hastening convergence. Running in parallel loses this advantage. Also, due to the overhead involved in which each thread is required to load every topology file when calculating energies, parallel scaling will begin to fall off as the number of threads reaches the number of frames. A usage example is shown below:


```
mpirun -np 2 MMPBSA.py.MPI -O -i mmpbsa.in -cp com.top -rp rec.top \
      -lp lig.top -y traj.crd
```

10.3.4. Types of calculations you can do

There are many different options for running MMPBSA.py. Among the types of calculations you can do are:

1. Normal binding free energies, with either PB or GB implicit solvent models. Each can be done with either 1, 2, or 3 different trajectories, but the complex, receptor, and ligand topology files must all be defined. The complex `mcdcrd` must always be provided. Whichever trajectories of the receptor and/or ligand that are NOT specified will be extracted from the complex trajectory. This allows a 1-, 2-, or 3-trajectory analysis. All PB calculations and GB models `igb = 1, 2, and 5` can be performed with just AmberTools via the `mmpbsa_py_energy` program installed with MMPBSA.py. GB models 7 and 8 require *sander* from the Amber package.
2. Stability calculations with any calculation type. If you only specify the complex `prmtop` (and leave receptor and ligand `prmtop` options blank), then a “stability” calculation will be performed, and you will get statistics based on only a single system. Any additional receptor or ligand information given will be ignored, but note that if receptor and/or ligand topologies are given, it will no longer be considered a stability calculation. The previous statement refers principally to mutated receptor/ligand files or extra ligand/receptor trajectory files.
3. Alanine scanning with either PB or GB implicit solvent models. All trajectories will be mutated to match the mutated topology files, and whichever calculations that would be carried out for the normal systems are also carried out for the mutated systems. Note that only 1 mutation is allowed per simulation, and it must be to an alanine. If `mutant_only` is not set to 1, differences resulting from the mutations are calculated. This option is incompatible with intermediate NetCDF trajectories (see the `netcdf = 1` option above). This has the same program requirements as option 1 above.
4. Entropy corrections. An entropy term can be added to the free energies calculated above using either the quasi-harmonic approximation or the normal mode approximation. Calculations will be done for the normal and mutated systems (alanine scanning) as requested. Normal mode calculations are done with the `mmpbsa_py_nabnmode` program included with AmberTools.
5. Decomposition schemes. The energy terms will be decomposed according to the decomposition scheme outlined in the `idecomp` variable description. This should work with all of the above, though entropy terms cannot be decomposed. APBS energies cannot be decomposed, either. Neither can PBSA surface area terms. This functionality requires *sander* from the Amber package.

10. MMPBSA.py

6. QM/MMGBSA. This is a binding free energy (or stability calculation) using the Generalized Born solvent model allowing you to treat part of your system with a quantum mechanical Hamiltonian. See “Advanced Options” for tips about optimizing this option. This functionality requires *sander* from the Amber package.
7. MM/3D-RISM. This is a binding free energy (or stability calculation) using the 3D-RISM solvation model. This functionality is performed with *rism3d.snglpnt* built with Amber-Tools.

10.3.5. The Output File

The header of the output file will contain information about the calculation. It will show a copy of the input file as well as all files that were used in the calculation (topology files and coordinate file(s)). If the masks were not specified, it prints its best guess so that you can verify its accuracy, along with the residue name of the ligand (if it is only a single residue).

The energy and entropy contributions are broken up into their components as they are in *sander* and *nmode* or *ptraj*. The contributions are further broken into G_{gas} and G_{solv} . The polar and non-polar contributions are EGB (or EPB) and ESURF (or ECAVITY / ENPOLAR), respectively for GB (or PB) calculations.

By default, bonded terms are not printed for any one-trajectory simulation. They are computed and their differences calculated, however. They are not shown (nor included in the total) unless specifically asked for because they should cancel completely. A single trajectory does not produce any differences between bond lengths, angles, or dihedrals between the complex and receptor/ligand structures. Thus, when subtracted they cancel completely. This includes the BOND, ANGLE, DIHED, and 1-4 interactions. If inconsistencies are found, these values are displayed and inconsistency warnings are printed. When this occurs the results are generally useless. Of course this does not hold for the multiple trajectory protocol, and so all energy components are printed in this case.

Finally, all warnings generated during the calculation that do not result in fatal errors are printed after calculation details but before any results.

10.3.6. Temporary Files

MMPBSA.py creates working files during the execution of the script beginning with the prefix `_MMPBSA_`. The variable “keep_files” controls how many of these files are kept after the script finishes successfully. If the script quits in error, all files will be kept. You can clean all temporary files from a directory by running `MMPBSA -clean` described above.

If MMPBSA.py does not finish successfully, several of these files may be helpful in diagnosing the problem. For that reason, every temporary file is described below. Note that not every temporary file is generated in every simulation. At the end of each description, the lowest value of “keep_files” that will retain this file will be shown in parentheses.

`_MMPBSA_gb.mdin` Input file that controls the GB calculation done in *sander*. (2)

`_MMPBSA_pb.mdin` Input file that controls the PB calculation done in *sander*. (2)

- `_MMPBSA_gb_decomp_com.mdin` Input file that controls the GB decomp calculation for the complex done in *sander*. (2)
- `_MMPBSA_gb_decomp_rec.mdin` Input file that controls the GB decomp calculation for the receptor done in *sander*. (2)
- `_MMPBSA_gb_decomp_lig.mdin` Input file that controls the GB decomp calculation for the ligand done in *sander*. (2)
- `_MMPBSA_pb_decomp_com.mdin` Input file that controls the PB decomp calculation for the complex done in *sander*. (2)
- `_MMPBSA_pb_decomp_rec.mdin` Input file that controls the PB decomp calculation for the receptor done in *sander*. (2)
- `_MMPBSA_pb_decomp_lig.mdin` Input file that controls the PB decomp calculation for the ligand done in *sander*. (2)
- `_MMPBSA_gb_qmmm_com.mdin` Input file that controls the GB QM/MM calculation for the complex done in *sander*. (2)
- `_MMPBSA_gb_qmmm_rec.mdin` Input file that controls the GB QM/MM calculation for the receptor done in *sander*. (2)
- `_MMPBSA_gb_qmmm_lig.mdin` Input file that controls the GB QM/MM calculation for the ligand done in *sander*. (2)
- `_MMPBSA_complex.mdcrd` Trajectory file printed out by `_MMPBSA_cenptraj.in` that contains only those snapshots that will be processed by MMPBSA.py. Only created if `full_traj` is set in `&general` or if a quasi-harmonic calculation is requested. (1)
- `_MMPBSA_ligand.mdcrd` Trajectory file printed out by `_MMPBSA_ligtraj.in` that contains only those snapshots that will be processed by MMPBSA.py. Only created in the same instances that `_MMPBSA_complex.mdcrd` is created. (1)
- `_MMPBSA_receptor.mdcrd` Trajectory file printed out by `_MMPBSA_rectraj.in` that contains only those snapshots that will be processed by MMPBSA.py. Only created in the same instance that `_MMPBSA_ligand.mdcrd` and `_MMPBSA_complex.mdcrd` are created. (1)
- `_MMPBSA_dummycomplex.inpcrd.1` Dummy inpcrd file generated by `_MMPBSA_complexinpcrd.in` for use with `imin=5` functionality in *sander*. (1)
- `_MMPBSA_dummyreceptor.inpcrd.1` Same as above, but for the receptor. (1)
- `_MMPBSA_dummyligand.inpcrd.1` Same as above, but for the ligand. (1)
- `_MMPBSA_complex.pdb` Dummy PDB file of the complex required to set molecule up in nab programs

10. MMPBSA.py

`_MMPBSA_receptor.pdb` Dummy PDB file of the receptor required to set molecule up in nab programs

`_MMPBSA_ligand.pdb` Dummy PDB file of the ligand required to set molecule up in nab programs

`_MMPBSA_complex_nm.mdcrd` Trajectory file printed out by `_MMPBSA_comtraj_nm.in`. (1)

`_MMPBSA_receptor_nm.mdcrd` Trajectory file printed out by `_MMPBSA_rectraj_nm.in`. (1)

`_MMPBSA_ligand_nm.mdcrd` Trajectory file printed out by `_MMPBSA_ligtraj_nm.in`. (1)

`_MMPBSA_ptrajentropy.in` Input file that calculates the entropy via the quasi-harmonic approximation. This file is processed by *ptraj*. (2)

`_MMPBSA_avgcomplex.pdb` PDB file containing the average positions of all complex conformations processed by `_MMPBSA_cenptraj.in`. It is used as the reference for the `_MMPBSA_ptrajentropy.in` file above. (1)

`_MMPBSA_complex_entropy.out` File into which the entropy results from `_MMPBSA_ptrajentropy.in` analysis on the complex are dumped. (1)

`_MMPBSA_receptor_entropy.out` Same as above, but for the receptor. (1)

`_MMPBSA_ligand_entropy.out` Same as above, but for the ligand. (1)

`_MMPBSA_ptraj_entropy.out` Output from running *ptraj* using `_MMPBSA_ptrajentropy.in`. (1)

`_MMPBSA_complex_gb.mdout.# sander` output file containing energy components of all complex snapshots done in GB. (1)

`_MMPBSA_receptor_gb.mdout.# sander` output file containing energy components of all receptor snapshots done in GB. (1)

`_MMPBSA_ligand_gb.mdout.# sander` output file containing energy components of all ligand snapshots done in GB. (1)

`_MMPBSA_complex_pb.mdout.# sander` output file containing energy components of all complex snapshots done in PB. (1)

`_MMPBSA_receptor_pb.mdout.# sander` output file containing energy components of all receptor snapshots done in PB. (1)

`_MMPBSA_ligand_pb.mdout.# sander` output file containing energy components of all ligand snapshots done in PB. (1)

`_MMPBSA_complex_rism.out.# rism3d.snglpnt` output file containing energy components of all complex snapshots done with 3D-RISM (1)

- `_MMPBSA_receptor_rism.out.#` *rism3d.snglpnt* output file containing energy components of all receptor snapshots done with 3D-RISM (1)
- `_MMPBSA_ligand_rism.out.#` *rism3d.snglpnt* output file containing energy components of all ligand snapshots done with 3D-RISM (1)
- `_MMPBSA_pbsanderoutput.junk.#` File containing the information dumped by sander.APBS to STDOUT. (1)
- `_MMPBSA_ligand_nm.out.#` Output file from *mmpbsa_py_nabnmode* that contains the entropy data for the ligand for all snapshots. (1)
- `_MMPBSA_receptor_nm.out.#` Output file from *mmpbsa_py_nabnmode* that contains the entropy data for the receptor for all snapshots. (1)
- `_MMPBSA_complex_nm.out.#` Output file from *mmpbsa_py_nabnmode* that contains the entropy data for the complex for all snapshots. (1)
- `_MMPBSA_mutant_...` These files are analogs of the files that only start with `_MMPBSA_` described above, but instead refer to the mutant system.
- `_MMPBSA_*out.#` These files are thread-specific files. For serial simulations, only `#=0` files are created. For parallel, `#=0` through `NUM_PROC - 1` are created.

10.3.7. Advanced Options

The default values for the various parameters as well as the inclusion of some variables over others in the general MMPBSA.py input file were chosen to cover the majority of all MM/PB(GB)SA calculations that would be attempted while maintaining maximum simplicity. However, there are situations in which MMPBSA.py may appear to be restrictive and ill-equipped to address. Attempts were made to maintain the simplicity described above while easily providing users with the ability to modify most aspects of the calculation easily and without editing the source code.

- use-mdins** This flag will prevent MMPBSA.py from creating the input files that control the various calculations (`_MMPBSA_gb.mdin`, `_MMPBSA_pb.mdin`, `_MMPBSA_sander_nm_min.mdin`, and `_MMPBSA_nmode.in`). It will instead attempt to use existing input files (though they must have those names above!) in their place. In this way, the user has full control over the calculations performed, however care must be taken. The mdin files created by MMPBSA.py have been tested and are (generally) known to be consistent. Modifying certain variables (such as `imin=5`) may prevent the script from working, so this should only be done with care. It is recommended that users start with the existing mdin files (generated by the flag below), and add and/or modify parameters from there.
- make-mdins** This flag will create all of the mdin and input files used by sander and nmode so that additional control can be granted to the user beyond the variables detailed in the input file section above. The files created are `_MMPBSA_gb.mdin`

10. MMPBSA.py

which controls GB calculation; `_MMPBSA_pb.mdin` which controls the PB calculation; `_MMPBSA_sander_nm_min.mdin` which controls the sander minimization of snapshots to be prepared for nmode calculations; and `_MMPBSA_nmode.in` which controls the nmode calculation. If no input file is specified, all files above are created with default values, and `_MMPBSA_pb.mdin` is created for AmberTools's *pbsa*. If you wish to create a file for sander.APBS, you must include an input file with "sander_apbs=1" specified to generate the desired input file. Note that if an input file is specified, only those mdin files pertinent to the calculation described therein will be created!

strip_mask This input variable allows users to control which atoms are stripped from the trajectory files associated with `solvated_prmtop`. In general, counterions and water molecules are stripped, and the complex is centered and imaged (so that if `iwrap` caused the ligand to "jump" to the other side of the periodic box, it is replaced inside the active site). If there is a specific metal ion that you wish to include in the calculation, you can prevent ptraj from stripping this ion by NOT specifying it in *strip_mask*. Note that *strip_mask* does nothing if no *solvated_prmtop* is provided.

QM/MMGBSA There are a lot of options for QM/MM calculations in *sander*, but not all of those options were made available via options in the MMPBSA.py input file. In order to take advantage of these other options, you'll have to make use of the `-make-mdins` and `-use-mdins` flags as detailed above and change the resulting `_MMPBSA_gb_qmmm_com/rec/lig.mdin` files to fit your desired calculation. Additionally, MMPBSA.py suffers all shortcomings of *sander*, one of those being that PB and QM/MM are incompatible. Therefore, only QM/MMGBSA is a valid option right now.

Please send any bug reports, comments, suggestions, or questions to amber@ambermd.org. Thanks!

11. mdgx: A Developmental Molecular Simulation Engine in Simple C Code

David S. Cerutti

The *mdgx* simulations package is a molecular dynamics engine with functionality that mimics some of *sander* and *pmemd*, but featuring simple C code and an atom sorting routine that simplifies the flow of information during force calculations. With the availability of *pmemd* and its GPU-compatible variant for efficient, long-timescale simulations, and the extensive development of thermodynamic integration, free energy calculations, and enhanced sampling methods that has taken place in *sander*, the principal purpose of *mdgx* is to provide a tool for radical redesign of the basic molecular dynamics algorithms and models. Currently, *mdgx* supports modest parallel capabilities, but the limiting factor is load-balancing; the molecular dynamics routines are designed for much higher parallelism. The first application of *mdgx* was to demonstrate the feasibility of multiple reciprocal space meshes spanning different regions of the simulation cell at different resolutions.[194] Future applications, discussed in more detail later in this chapter, pertain to new charge distributions with significant numbers of off-atom “virtual” force centers.

While it is capable of performing molecular dynamics based on standard **prmtop** topology, **inpcrd** starting coordinates files, and input files in a format very much like the **mdin** files, it should be emphasized that *mdgx* is really a program for experts with knowledge of classical dynamics algorithms. There is currently no minimization algorithm in place, so *mdgx* cannot yet be used as a standalone program for converting coordinates from an experiment into a trajectory. The program does not have all of the thermostats available in the *sander* and *pmemd* software packages, and the compatibility of some barostats and constraint algorithms is not yet verified and therefore forbidden in the release version. However, *mdgx* is starting to incorporate some of the more advanced features of *sander* and *pmemd*, and with continued development is on a course to become a production molecular dynamics code for general use.

11.1. Input and Output

Input command files for *mdgx* may be similar to the **mdin** format used by *sander* and *pmemd*. One requirement of *mdgx* that is not found in *sander* is that each of the &namelist segments of the input file must begin with the identifier of the &namelist on its own line and end with the keyword &end on its own separate line. However, the &namelist format is not strictly enforced in *mdgx*, not all *sander* input variables are available in *mdgx*, and some new input variables have been added. All *mdgx* input variables can also be identified by aliases that may be lengthier than their *sander* counterparts but may make the input easier for a human to parse.

11. *mdgx*: A Developmental Molecular Simulation Engine in Simple C Code

All *mdgx* &namelists and their associated variables may be browsed by running the *mdgx* program itself; running the program with no command line arguments will produce basic instructions for usage and a list of command-line arguments to display each &namelist. Certain directives to *mdgx* may be supplied as either part of the input file or on the command line; in particular, the names of the topology, input coordinates, and output files may be specified in either manner. Also, in the new release, the random number generator seed and thermodynamic integration coupling parameter λ may be specified on the command line. This duality makes it possible to pack information into an input file and run *mdgx* with much shorter command-line input. However, if the same variable is declared both on the command line and in the input file, the command-line input will take precedence. This predominance may make it possible to run multiple related *mdgx* runs based on a single input file. Units of input variables follow the *sander* and *pmemd* conventions.

The *mdgx* program will read standard AMBER **prmtop** files using its own routines and perform basic tests of the topology to identify common problems such as omitted disulfide bonds or “D” to “L” chirality flips in the standard amino acids; any potential problems are reported in the **mdout** output diagnostics files, but do not immediately lead the program to halt. In addition to the standard information contained in an AMBER topology file, *mdgx* is being developed to also be able to read other sorts of information given certain directives in the input command (**mdin**) file. As will be discussed later, *mdgx* is able to read auxiliary information that modifies the topology specified by a **prmtop** file, adding virtual sites or changing the nonbonded parameters of specified atoms. (These changes are not written back into the original **prmtop**.)

Output files produced by *mdgx* follow the AMBER .crd and NetCDF formats for coordinates and velocities. Forces on all atoms can also be printed over the course of a trajectory. Separate suffixes may be applied to the **mdout** output diagnostic information, trajectory files, energy, and restart files, as specified by the user. *mdgx* also has the capability to print outputs from a single trajectory into multiple segments if the user specifies a value of the *nfistep* variable that is a factor of the *sander*-related *nstlim* variable. In such a case, *mdgx* will print files of the format [base name]###.[suffix], where [base name] is a base file name supplied by the user, ### is a number of the segment beginning at zero, and [suffix] is a file extension supplied by the user (for instance, “rst” for restart files, “out” for **mdout** output diagnostics). The number of segments is determined by the ratio of *nstlim* to *nfistep*; the former indicates the total number of dynamics steps, the latter the number of steps in each segment. At the start of dynamics *mdgx* will check for the existence of complete output diagnostics and restart files (as indicated by a special three-line ASCII mark) starting at segment 0 and continuing until a missing output diagnostics file is encountered, even if file overwriting (the *sander*-related “-O” command-line option) has been specified. (Not allowing file overwriting will only cause the program to abort if, on a subsequent segment for which complete output diagnostics and restart files do not exist, some other output such as a trajectory coordinates file does exist.) The intention of this elaborate scheme is to permit one long run to be broken into many segments without halting the program, and to provide an internal means of checkpointing a run if the program must be restarted. Because the changes to the output format are potentially dramatic, the *nfistep* variable must be set deliberately; any value that is not a factor of *nstlim* will result in *nfistep* being set to zero and outputs will be printed to files named [base name].[suffix].

The *mdgx* program also provides its own output format for force diagnostics. In *sander*,

information relating the bond, angle, torsion, and nonbonded direct and reciprocal space forces is only available by running in “debugging” mode as specified by the `&debugf` namelist block. In *mdgx*, such output is available by setting the *sander*-related *imin* variable to 2; the output is produced in ASCII format with numerous comments to make the results comprehensible to a human.

11.2. Installation

mdgx is installed as part of the AmberTools package. The program relies on the FFTW 3.2.2 and NetCDF libraries already distributed as part of AmberTools.

11.3. Special Algorithmic Features of *mdgx*

While it does not currently support the breadth of molecular dynamics algorithms offered by the *pmemd* or *sander* programs, *mdgx* does have capabilities that set it apart from other simulators in the AMBER software package.

First, *mdgx* can perform molecular simulations at constant volume with the “Multi-Level Ewald” implementation of Particle Mesh Ewald [194] electrostatics. This algorithm breaks the one reciprocal space mesh used in most Particle:Particle / Particle:Mesh techniques into multiple slabs spanning subdomains of the simulations cell and a much coarser variant of the global mesh for reuniting the subdomains. The intention of this algorithm is to provide a means for distributing and mitigating the communications required for solving the system’s long-ranged electrostatics. *mdgx* also provides an implementation of standard Smooth Particle Mesh Ewald electrostatics [195], with the added generality of independent interpolation orders in each of the three mesh dimensions. These features may be accessed through control variables in the `&ewald` namelist.

Another feature of *mdgx* currently not found in any other AMBER molecular dynamics engine is a Monte-Carlo barostat, available by specifying *ntt* = 5. This remarkably simple barostat makes volume moves, rescales system coordinates to match the new unit cell dimensions, and uses the Metropolis criterion to compare the energies of the original and trial configurations:

$$\chi_{\text{acc}} = \min \left[1, \left(\frac{V^{\text{new}}}{V^{\text{old}}} \right)^N \exp \left(-\frac{1}{kT} (U^{\text{new}} - U^{\text{old}} + P(V^{\text{new}} - V^{\text{old}})) \right) \right]$$

In the above formula, the probability of accepting a move χ is determined by the product of two factors. The first factor is the ratio of volumes V in the trial (new) and initial (old) configurations taken to the power of the number of particles in the system N . (Note that in the presence of rigid constraints, each rigidly constrained group of atoms counts as only one particle.) The second factor is a Boltzmann-weighted probability based on the sum of the potential energy of the system U and the pressure-volume work that the system does on its surroundings. In the above formula, the pressure P and temperature T are arbitrary parameters of the barostat: specifically, P is the external pressure (*pres0* in *sander* input), and T is set to match the external temperature of the thermostat in use. In this barostat, the system kinetic energy does not directly play a role in determining the system volume. However, in a condensed system of real particles the kinetic and potential energies quickly exchange, and even in the case

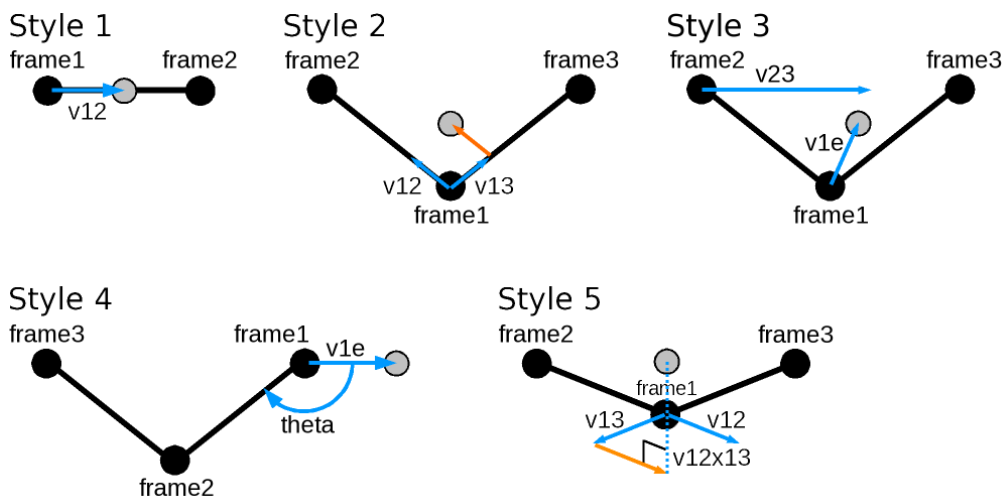
of an ideal gas the two factors balance out such that the familiar ideal gas law is recovered so long as the temperature T given to the barostat and the actual temperature of the gas particles match. Currently, this barostat is set up to rescale the volume isotropically, but in principle anisotropic volume changes and even alterations of the unit cell angles are feasible.

A principal advantage of the Monte-Carlo barostat is that no computation of the virial is necessary: the barostat can be applied at whatever frequency the user requires, and will maintain the proper system volume so long as moves are accepted at a frequency much greater than the rate at which the system might spontaneously move to configurations which change its equilibrium volume under the applied external pressure P . The default of attempting one barostat move by increasing or decreasing the system volume by up to one-tenth of a percent of its initial volume appears to result in a good acceptance ratio. The default of attempting the moves every 100 steps of dynamics, about every 100 to 200 femtoseconds, should be sufficient to accommodate most processes of interest and produce sound equilibrium statistics on the timescale of nanoseconds. Because the moves only require recalculation of the energy (which is done with merely a few additions and multiplications in special cases), the Monte-Carlo barostat may also have a speed advantage over the methods currently implemented in *sander* and *pmemd*.

11.4. Customizable Virtual Site Support in *mdgx*

It is not completely feasible to perform molecular dynamics with massless particles. However, for many useful cases in which the locations of massless particles are determined by the locations of two or more atoms with mass, it is possible to perform dynamics by using the chain rule to transfer forces from the “virtual sites” to the massive particles. These constructions, enumerated below, provide a means for breaking out of the “one atom, one site” paradigm that has dominated classical molecular dynamics, but the **prmtop** format utilized by the *sander* and *pmemd* programs does not always provide a straightforward means of expressing the relationships between virtual sites and their parent (or “frame”) atoms and the *sander* and *pmemd* programs only support the most widely used cases of virtual sites (e.g. TIP4P and TIP5P water).

The *mdgx* program provides a means for adding any number of virtual sites to an existing force field, with custom charges and even Lennard-Jones properties. The only limitations with the virtual sites are that no new bonded terms may be added, that the virtual sites carry zero mass, and that each virtual site location be determined by two or three frame atoms which do have mass. The constructions below follow those outlined in the GROMACS manual; a four-point frame construction devised by the GROMACS team is not yet implemented, but a “zeroth” frame type is available in *mdgx* which allows, without changing the **prmtop**, run-time modification of existing atomic non-bonded parameters.



In the figure above, the `&rule` namelist variables for specifying each virtual site constructor are superimposed on atoms, vectors, and angles. In Style 1, the virtual site lies along the line determined by two atoms; v_{12} denotes the fraction of the distance between the two atoms at which to place the virtual site. In Style 2, the virtual site lies in the plane determined by three atoms at a point determined by a combination of the displacements between atoms 1 and 2 and atoms 2 and 3. Virtual sites of Styles 1 and 2 are located by linear combinations of the positions of their frame atoms. In Style 3, the virtual site is located along the line described by frame atom 1 and a point between frame atoms 2 and 3 (v_{23} denoting the fraction of this distance), at a fixed distance v_{1e} from frame atom 1. Style 4, perhaps the most mathematically challenging frame type to define but very useful and intuitively comprehensible, places a virtual site at a fixed distance v_{1e} from frame atom 1 such that the angle illustrated has the value θ (specified in radians in the `&rule` namelist). The virtual site remains in the plane of the frame atoms, and frame atom 3, which must not be colinear with the other frame atoms, orients the sign of θ . Virtual sites of Style 5 are defined as sites of Style 2, but projected normal to the plane according to a multiple $v_{12 \times 13}$ of the cross product of the vectors between frame atoms 1 and 2 and frame atoms 1 and 3. Note that virtual sites of Styles 1, 2, and 5 will stretch with their frames, whereas 3 and 4 will not. The stretching will be minor if the frame atoms are bonded as shown in the figure. Due to the manner in which virtual sites are positioned in *mdgx*, frame atoms 2 and 3, and the virtual site when placed, must lie within half the van-der Waals non-bonded cutoff of frame atom 1. This should seldom if ever be a problem. A complete list of `&rule` namelist variables follows in the table.

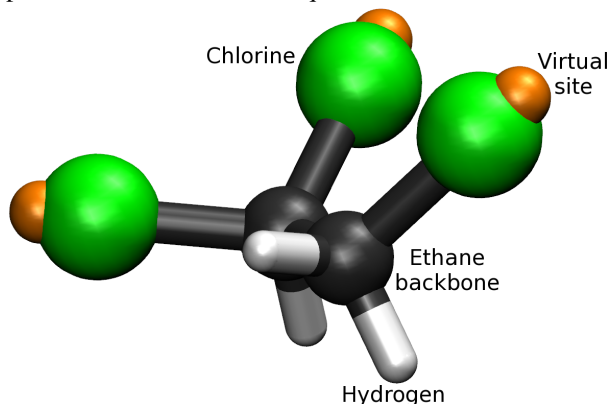
11. *mdgx: A Developmental Molecular Simulation Engine in Simple C Code*

Name	Alias	Description
frame?	FrameAtom?	When ? is 1, 2, or 3, this specifies the frame atoms needed for virtual site construction
epname	ExtraPoint	The name of the virtual site
atom	AtomName	The name of the virtual site (alternate specifications)
style	FrameStyle	The frame style to use (see descriptions in the preceding figure); acceptable values are 0 through 5
excl?	Exclude?	The virtual site is definitively 1:1 bound to frame atom 1 and thereby inherits all 1:2, 1:3, and 1:4 neighbors of frame atom 1, but if ? is 2 or 3 then the virtual site will also be considered 1:1 to frame atoms 2 or 3 and inherit their bonded neighbors as well. This will not affect the 1:2, 1:3, and 1:4 neighbor lists of the frame atoms themselves.
v12	Vector12	Defined according to frame type; see preceding paragraph and illustration.
v1e	Vector1E	
v13	Vector13	
theta	Theta	
v23	Vector23	
v12x13	Vector12x13	
q	Charge	Charge of the virtual site
sig	Sigma	Lennard-Jones σ and ϵ parameters of the virtual site
eps	Epsilon	
residue	ResidueName	The residue to which extra points will be added. Because it is specified according to the four-character name, there is some possibility for ambiguity as terminal residues often have the same names as residues in the middle of a chain. Therefore, in order to add a virtual site to an the amino terminus of N-terminal alanine but skip over alanines within a polypeptide, the N-terminal alanine would have to be given a new name within the prmtop .

The purpose of the zeroth frame type is to round out a temporary solution to the problem of testing virtual sites configurations in Amber; ultimately, the best solution is to incorporate all virtual site constructions into LEaP and expand the **prmtop** format to accommodate them. However, for experimentation and validation the *mdgx* approach of adding particles to an existing topology is straightforward, faster than creating new topologies starting with antechamber, and will remain available as part of the program for the foreseeable future. It is possible in *mdgx* (noting that the rigid geometry of the massive atoms is the same throughout all TIP water models) to simulate TIP4P(author?) 46 or TIP5P(author?) 46 water starting from a **prmtop** containing TIP3P water, although it is more convenient and perhaps marginally faster to simulate beginning with a **prmtop** specifying the more complex water model.

Virtual sites added in this manner follow the neighbor conventions described in the accompanying AMBER manual: virtual sites are counted as “1:1” neighbors of their first parent atoms and then inherit all 1:2 (bond), 1:3 (angle), and 1:4 nonbonded neighbors of the first parent atom. It is also possible to endow virtual sites with neighbors of other parent atoms, effec-

tively declaring the virtual sites to be 1:1 neighbors of more than one atom. The neighbor list updates implied by adding virtual sites do not get applied retroactively, however, so multiple frame atoms do not become 1:1 neighbors of each other. Because of the exclusions implied by different frame constructions, care should be taken when defining parent atoms. For instance, in the chlorinated ethane derivative below virtual sites of frame type 1 ($v12 = -0.3$, with chlorines being frame atom 1 and the bonded carbons being frame atom 2) can be shown to significantly improve the electrostatic fit to quantum-mechanical MP2 calculations.



In principle, the frame atom 1 may be defined as the carbon, with the chlorine (which is actually closest to the virtual site) merely defining the direction of the virtual site projection. However, this construction omits interactions between virtual sites on opposite ends of the molecule, and as a result the torsional conformations of the molecule are drastically altered (so much so that the hydration free energy in explicit solvent simulations changes by more than 3 kcal/mol). If the chlorines themselves are made frame atom 1 in each virtual site frame, the virtual sites become 1:4 neighbors to one another and interact by a slightly screened electrostatic potential. The effects on the torsional distribution and resulting hydration free energy are then much more modest. This trichloroethane represents an extreme case, but more subtle examples abound. In general, virtual sites can change the charge distribution of a molecule to roughly the same degree that refitting an atom-centered charge model to new quantum data does. Ideally, torsional parameters would be refitted in all cases to accommodate the new electrostatic model.

11.5. Restrained Electrostatic Potential Fitting in *mdgx*

Because of the extensive capabilities for adding virtual sites, *mdgx* also contains an internal means of assigning charges to them. The Restrained Electrostatic Potential (RESP) methodology is the basis for charge assignment based on quantum-mechanical electrostatic potential data, but the details differ somewhat from the implementation in *antechamber*.

The basic concept of fitting charges to reproduce the electrostatic potential of a molecule, by finding the solution with least squared error in the presence of restraints, is carried over from the original Kollmann RESP. However, instead of Lagrangian constraints, equivalent charges are unified as single variables in the fit, and penalty functions are added to the fitting matrix to enforce total charge constraints. This is not an ideal implementation, but it does return high-

11. *mdgx: A Developmental Molecular Simulation Engine in Simple C Code*

quality fits. Residual inequalities left over by the inexact constraint methods are eliminated post-hoc by routines that strictly equalize charges which are meant to be equal and set the total system charge to the specified value. Where *mdgx* does excel is in the control it gives the user over what fitting data will be used. Rather than relying on a quantum-chemistry package to select a particular surface around a molecule, *mdgx* will read the electrostatic potential due to that molecule on a regular grid and select points from that grid based on a solvent-accessible region determined by the actual Lennard-Jones parameters of the model. Because most solvent models make use of hydrogen atoms with modest or non-existent steric properties, *mdgx* also considers points which may not be accessible to the solvent probe but might be accessible to a hydrogen atom connected to that probe. *mdgx* will read a **prmtop** describing the system and also, if required, a Virtual Sites rule file, so that partial charges may be fitted for any virtual sites that the user wishes to add. Once fitting is complete, *mdgx* can return a new Virtual Sites rule file that will apply the fitted charges to the original **prmtop** in future simulations.

Fitting is called by its own separate &fit namelist, and triggers a distinct run mode in the sense that the program will terminate after the fit is complete. The options available in the &fit namelist include:

11.5. Restrained Electrostatic Potential Fitting in mdgx

Name	Alias	Description
phi#	QMPhi#	Names of additional electrostatic potentials to use in fitting. The files are read as formatted <i>Gaussian</i> cubegen output, containing electrostatic potentials sampled on a regular grid and a list of molecular coordinates which is expected to match the atoms found in the prmtop .
auxphi#	AuxPhi#	An auxiliary electrostatic potential to use in fitting, also in formatted <i>Gaussian</i> cubegen output, corresponding to phi#. The effect of specifying an auxiliary potential is to have a single set of charges fit to reproduce the average of the two potentials. This feature supports development of fixed-charge force fields if one posits that the correct charges of a non-polarizable model would sit halfway between the charges of a fully polarized molecule in some solvent reaction field and the charges of an unpolarized molecule in the gas phase.
eprules	EPRules	If specified, <i>mdgx</i> will output all fitted charges in the form of a Virtual Sites rule file, which can be given as input to subsequent simulations to modify the original prmtop and apply the fitted charge model.
conf	ConfFile	If specified, <i>mdgx</i> will output the first molecular conformation, complete with any added virtual sites, in PDB format for inspection. This is useful for understanding exactly what model is being fitted.
qtot	TotalQ	The total charge constraint in units of the proton charge; the sum of all fitted charges is required to equal this value. Default 0.0.
minq#	MinimizeQ#	Restrain the charges of a group of atoms to zero by the weight given in minqwt. The groups are specified in ambmask format.
equalq#	EqualizeQ#	Restrain the charges of a group of atoms to have the same values. Groups are specified in ambmask format.

11. *mdgx*: A Developmental Molecular Simulation Engine in Simple C Code

Name	Alias	Description
minqwt	MinQWeight	Weight used for restraining values of charges to zero; as more and more fitting data is included (either through a higher sampling density of the electrostatic potential due to each molecular conformation or additional molecular conformations) higher values of minqwt may be needed to keep the fitted charges small. However, with more data the need to restrain charges may diminish as well.
phiwt#	PhiWeight#	The weights assigned to electrostatic potentials specified by phi#. This modulates the importance of one molecular configuration, and the electrostatic potential it implies, in the fit. Default 1.0 for all files phi#.
nfpt	FitPoints	The number of fitting points to select from each electrostatic potential grid. The points nearest the molecule, which satisfy the limits set by the solvent probe and point-to-point distances as defined below, will be selected for the fit. Default 1000.
psig	ProbeSig	The Lennard-Jones σ parameter of the solvent probe. Default 3.16435 (TIP4P oxygen).
peps	ProbeEps	The Lennard-Jones ϵ parameter of the solvent probe. Default 0.16275 (TIP4P oxygen).
parm	ProbeArm	The probe arm; points on the electrostatic potential grid that would be inaccessible to the solvent probe may still be included in the fit if they are within the probe arm's reach. Default 0.9572 Å (TIP oxygen-hydrogen bond distance).
pnrj	StericLimit	The maximum Lennard-Jones energy of the solvent probe at which a point will qualify for inclusion in the fit. Default 3.0 kcal/mol.
flim	Proximity	The minimum proximity of any two points to be included in the fit. Default 0.4 Å.
hbin	HistogramBin	If hist is specified, <i>mdgx</i> will print a histogram reporting the number of fitting points falling within any particular distance of some atom of the molecule. This parameter controls the discretization of the histogram.
maxmem	MaxMemory	Because fitting matrices can become very large in some cases (in particular, those involving multiple systems with correlated partial charges), <i>mdgx</i> offers this parameter as a safeguard against creating a matrix that may inadvertently take up too much memory. Values for this argument may be integers, or integers followed immediately (no spaces) with terms such as "GB," "Mb," or "kB" (case-insensitive) for giga/mega/kilo bytes. Default 1GB.
verbose	Verbose	Unless set to zero by the user, <i>mdgx</i> will print periodic updates and record milestones from the fitting run in terminal output.

Many options in the &fit namelist may be specified with numbers, denoted by # in the table

above. The # represents any number from 1 to 256, but declining to state a number simply implies the first member of the series. Skips in the series are forbidden. An example of a &fit namelist is given below. In this particular problem, EC12 and EC13 were the names of virtual sites not in the original topology file but specified by a Virtual Sites rule file.

```
&fit
  QMPhi1    Conf12/pcm12.cube,
  QMPhi2    Conf13/pcm13.cube,
  QMPhi3    Conf14/pcm14.cube,
  pnrg      2.0,
  nfpt      15000,
  minqwt    175.0,
  EqualizeQ1 '@H1,H2'
  EqualizeQ2 '@C12,C13'
  EqualizeQ3 '@EC12,EC13'
  MinimizeQ = '@E*'
  EPRules    frag.xpt
  ConfFile   f6xp.pdb
&end
```

Virtual site constructions have strong support in *mdgx* to rapidly translate between an imagined model and a practical simulation.

11.6. Thermodynamic Integration

A rudimentary implementation of thermodynamic integration is available in *mdgx*. This facility is not fully developed, but does permit users to test changes in hydration free energy or other consequences of new charge models, such as those that include virtual sites. The only significant similarity to *sander* is that there are two trajectories propagated simultaneously using a mixture of the forces obtained at each endpoint; otherwise the implementation is very different. In *mdgx*, both trajectories are propagated by the same processor, so it is feasible to run TI in serial mode, without a parallel build. A single input file carries all the necessary information for a thermodynamic integration run, including the names of the topologies describing the initial and final states of the system and the path for changing between them. A single output file contains all the relevant information concerning the energies of the system at each endpoint and the derivative of the potential with respect to the coupling parameter λ .

Parameters specific for thermodynamic integration in *mdgx* include:

11. *mdgx*: A Developmental Molecular Simulation Engine in Simple C Code

Name	Alias	Description
icfe	RunTI	Flag to turn on thermodynamic integration (default 0, set to 1 to activate)
klambda	MixOrder	The exponent on terms involving the coupling parameter λ (see the AMBER manual; default 1)
clambda	MixFactor	The value of the coupling parameter λ (default 0.0)
nsynch	SynchTI	Frequency at which to explicitly synchronize the two trajectories. In principle, this should never be necessary but to prevent some corner case from occurring the nsynch is set to 1000 steps by default. A brief report of activity required to synchronize coordinates appears in the output file every time this routine is called.

The *mdgx* program can accept up to two topologies, for the initial and final states of the system. The topologies are specified by the argument “-p#” on the command line or the arguments “Topology#” or “-p#” in the **mdin** input file, where # is blank, 1, or 2. A blank value of # corresponds to 1. Different Virtual Site rules files may be specified for each topology with the “-xpt#” option on the command line and the “-xpt#” or “EPRules#” options in the **mdin** file. Assigning the same topology file to both -p1 and -p2 parameters but assigning a Virtual Sites rule file to one of them is a way to test the energetic consequences of changing the charge model found in some standard force field to one that includes new virtual sites.

Thermodynamic integration routines in *mdgx* can handle topologies of different numbers of atoms. Unique atoms at each endpoint are considered points with mass but no other properties at the other endpoint. This functionality is not yet mature, so “experts only!” With the future addition of soft-core repulsive potentials for smooth growth and removal of atoms, this functionality will become more robust and accessible to end users.

11.7. Future Directions and Goals of the *mdgx* Project

While it does draw on adaptations of some code found in the *sander* program, *mdgx* is not a re-implementation of a subset of *sander*’s functionality. Many of the algorithms used by *mdgx* differ from those used by *sander* and *pmemd*, including the velocity version of the Verlet integrator, the domain decomposition for nonbonded interactions, and an atomic, as opposed to a molecular, virial calculation (if the virial is computed at all). This independence of the *mdgx* program may create some difficulties when trying to compare *mdgx* results from simulations to *sander* and *pmemd*. However, the program’s simple C implementation should be adaptable and expandable, and added features for conveniently printing forces and energies for specific snapshots or over entire trajectories should make the program a useful tool for testing and comparing new algorithms.

With this release, *mdgx* offer modest parallel scaling to 8 or more CPUs, for a roughly five- to six-fold speed advantage over its serial implementation. Currently the lack of a real load balancer reduces the efficiency of parallel implementation and the strategy of merging all reciprocal space mesh pieces onto a single processor for three-dimensional FFT operations limits the overall scaling. However, other aspects of the code are designed to support much higher levels of parallelism. The collection of reciprocal space mesh pieces on multiple processors for par-

allel FFT operations is a straightforward extension of the existing code. While the direct space decomposition presently requires that cells be at least the length of the direct space cutoff in all dimensions, a typical 10Å cutoff would still permit 27 processors to be devoted to direct-space tasks for a system of 1,000 water molecules, and with simple extensions the direct space work could be divided cleanly between two processors: in such a state, the code would theroretically scale to more than one processor per 50 atoms. Also planned for the next release is a parallel implementation that takes full advantage of the Multi-Level Ewald method for decomposition of the reciprocal space electrostatic sum to make *mdgx* scale well across multicore nodes.

Aside from providing a platform for testing new algorithms, another major goal of the *mdgx* project is to support force field development. This is why *mdgx* contains its own charge-fitting implementation as well as arbitrary virtual site support. In order to further support force field development, future releases of *mdgx* will include optimizations for thermodynamic integration that avoid redundant computation of identical quantities in force calculations for the two alchemical endpoints, as well as soft-core potentials and support for other free energy methods and enhanced sampling.

12. paramfit

Robin Betz

The *paramfit* program allows specific forcefield parameters to be optimized or created by fitting to quantum energy data. *Paramfit* can be used when parameters are missing in the default forcefields and *antechamber* cannot find a replacement, or when existing parameters do not describe the system to the desired level of accuracy, such as for dihedral constants on protein backbones.

Paramfit attempts to make the following statement true: **The quantum energy and the energy that AMBER predicts should be the same over many conformations of a structure.**

Paramfit attempts to fit the AMBER energy to the quantum energy for a variety of conformations of the input structure, minimizing the equation $E_{MD} - E_{QM} + K = 0$ where K is a constant intrinsic difference between the QM and MD calculations. The program works by altering the parameters that AMBER uses to describe the molecule, which alter the elements in the AMBER sum that is used to calculate the energy. It is necessary to evaluate over many conformations of the molecule because the parameters predict how the molecule will behave dynamically rather than statically. To get a good idea of the forces on a dihedral, for example, the energy needs to be evaluated for multiple conformations of the dihedral to see how it changes each time. *Paramfit* will fit so that the energy changes that AMBER predicts will happen when the dihedral twists match the changes predicted with quantum methods.

Paramfit provides functionality for the majority of steps in the fitting process, including writing input files for quantum packages, specifying which parameters are to be fit, determining the value of K for the system, and finally conducting the fit and saving it in a force field modification file that can be used by other programs. An external quantum program is needed to generate the energies needed for *paramfit* to conduct a fitting. Currently, the program is capable of writing input files for ADF, GAMESS, and Gaussian, although if you write your own input files instead of using *paramfit*'s functionality, any quantum package will work.

Paramfit has OpenMP support for parallelization of the AMBER function evaluation over the input conformations, where each core will evaluate the energy for a subset of the conformations. Enable this by adding the *-openmp* option to configure and rebuilding *paramfit*. By default all available cores will be used. To change this, set the OMP_NUM_THREADS environment variable to the number of threads to be executed.

12.1. Usage

Paramfit is called from the command line:

```
paramfit -i Job_Control.in -p prmtop -c mdcrd -q QM_data.dat \
-v MEDIUM --random-seed seed
```

12. *paramfit*

Only input files are specified from the command line, and will default to the following file names unless otherwise specified.

Job_Control.in The job control file for the program. See [12.2](#) for a description of the options and format for this file.

prmtop The molecular topology file for the structure.

mdcrd A coordinate file containing many conformations of the input structure. These may be generated by running a short simulation in solution, or by manually specifying coordinates for each atom. It is important that there be a good representation of the solution space for any parameters that are to be optimized- for example, if you want a bond force constant it would be a good idea to have input structures with a good range of values for the length of the that bond type. See [12.2.6](#)

QM_data.dat A file containing the quantum energies of the structures in the coordinate file, in order, one per line. You will have to extract the energies from the output files that the quantum package produces. An example script to do this for Gaussian formatted output files can be found in \$AMBERHOME/AmberTools/src/paramfit/scripts.

MEDIUM The verbosity level to run the program at, either LOW, MEDIUM, or HIGH.

seed The integer seed for the random number generator. Only specify this parameter when exactly reproducible results are needed for debugging.

12.2. The Job Control File

Similarly to *sander* and other programs, *paramfit* requires a job control file that specifies individual options for each run. The format consists of variable assignments, in the format variable=value, with one assignment per line. Pound signs (#) will comment out lines. See the following sections for a description of what to put in the job control file for various tasks. There is a template job control file in \$AMBERHOME/AmberTools/src/paramfit/example_config_files that lists all possible options.

12.2.1. General options

paramfit requires several options be set for every run. These variables should usually appear in your job control file.

RUNTYPE Specifies whether this run will be creating quantum input files, setting parameters, or conducting a fit.

= CREATE_INPUT The structures in the coordinate file will be written out as individual input files for a quantum package. See [12.2.2](#).

= SET_PARAMS Provides an interactive prompt allowing you to specify which parameters will be fit for this molecule. See [12.2.3](#).

= **FIT** Conducts a fitting using one of the two minimization algorithms. See 12.2.4 for other options that need to be specified.

NSTRUCTURES Specifies how many structures are in the input coordinate file. If this value is less than the total number of structures in the file, only the first **n** will be read.

12.2.2. Creating quantum input files

Given a trajectory, *paramfit* can write input files for a variety of quantum packages. This is necessary to generate the energy values for each input conformation that *paramfit* will fit to. You do not necessarily need to do this step and can write your own input files if desired. Currently Gaussian, ADF, and GAMESS formats are supported.

By default, the files will be named Job.**n**.in, where **n** is the nth structure in the coordinate file. Once all the input files are written, you must run the quantum package yourself and then extract the energies from the output files into the format that *paramfit* requires, with one energy per line in the same order as the input structures. An example script to do this for Gaussian format output files is in the \$AMBERHOME/AmberTools/src/paramfit/scripts directory.

To enter this mode, set RUNTYPE=CREATE_INPUT and specify the following options in your job control file:

QMHEADER File that will be prepended to all created input files for the quantum program. This specifies things on a per-system basis, such as choice of basis set, amount of memory to use, etc. These parameters will vary depending on which quantum package you are using. Sample header files for all supported quantum packages are included in example_config_files in *paramfit*'s source directory.

QMFILEFORMAT Specifies which quantum package the created input files should be formatted for.

= **ADF** Use the Amsterdam Density Functional Theory package.

= **GAMESS** Use the General Atomic and Molecular Electronic Structure System (GAMESS).

= **GAUSSIAN** Use Gaussian.

QM_SYSTEM_CHARGE The integral charge of the system. Defaults to 0. Note that some quantum packages may require this to also be specified in your header file.

QM_SYSTEM_MULTIPPLICITY The integral multiplicity of the system. Defaults to 1 (singlet).

QMFILEOUTSTART The prefix for each of the created input files. Defaults to 'Job.' The structure number and then the suffix will be appended to this value.

QMFILEOUTEND The suffix for each of the created input files. Defaults to '.in'. With both default options, the file will be named Job.**n**.in.

12.2.3. Specifying parameters

In order to facilitate batch runs as well as simplify the process of running *paramfit* on larger systems, the parameters to be fit are saved and then loaded in during actual fitting so that they do not have to be specified every time. The parameter setting runtime accomplishes this by prompting whether you would like to fit bond, angle and/or dihedral parameters and then displaying a list of the specific atom types for each so that you can pick exactly what *paramfit* should optimize. This saved file does not specify a value for any of the parameters, but simply indicates which ones are to be changed during fitting.

If you do not wish to save a parameter file, you may instead fit a default set of parameters or be prompted every time. See [12.2.4](#).

To enter this mode, set RUNTYPE=SET_PARAMS and the following options:

PARAMETER_FILE_NAME Specifies the name of a file in which to store the parameters. When loading these parameters in during a fitting, this line will stay the same. Do not modify this file by hand: *paramfit* numbers each bond, angle, and dihedral in a manner that is consistent but not human-readable.

12.2.4. Fitting options

The fitting function accomplishes the actual parameter modification. It does this by minimizing the least squares difference between the quantum energy and the energy calculated with the AMBER equation over all of the input conformations. For a perfect fit, this means that over all structures, $E_{MD} - E_{QM} + K = 0$.

K is the intrinsic difference between the quantum and the classical energies, which is represented as a parameter that is also fit. The value of K depends on the system, and should be fit once as the only parameter before fitting any other parameters.

To enter this mode, set RUNTYPE=FIT and set the following additional variables:

FITTING_FUNCTION The minimization algorithm to use. *paramfit* currently implements a genetic algorithm and a simplex algorithm for conduction minimization. Each algorithm requires several parameters and is suited to different problems. Please see [12.2.5](#) for descriptions of these options and a guide on choosing the appropriate algorithm.

= GENETIC

= SIMPLEX

K The intrinsic difference between the quantum and classical energies. This value needs to be determined once for each system so that the algorithm can minimize to zero instead of to a constant. See [12.3.2](#) for an example.

PARAMETERS_TO_FIT Sets how *paramfit* determines which parameters are to be fit. *paramfit* does not fit electrostatics, but is capable of fitting every other element of the AMBER sum, which include bond harmonic force constant and equilibrium length, angle harmonic force constant and equilibrium angle, and proper and improper dihedral barrier height, phase shift, and periodicity. As a general rule, the fewer parameters there are to fit, the faster and more accurate the results will be. Avoid fitting more parameters than necessary.

- = **DEFAULT** Fit all bond force constants and lengths, angle force constants and sizes, and dihedral force constants. This option will usually fit a very large number of parameters, and is rarely necessary. For most cases, only a few parameters are desired, and they should be fit individually.
- = **K_ONLY** Do not fit any force field parameters. Only fit the value of K (the difference between quantum and classical energies for the system). This needs to be done once per system in order to determine K before any other parameters are fit, as attempting to fit it at the same time results in inaccurate results. Since small changes in K produce a great change in the overall least squares sum, the algorithm will tend to focus on changing the value of K and will neglect the parameters.
- = **LOAD** The list of parameters to be fit is contained in a file that was previously created with the parameter setting runtime. Set `PARAMETER_FILE_NAME` to the location of this file.
- = **PROMPT** Interactively asks the user which parameters are to be fit for this run.

SCEE The value by which to scale 1-4 electrostatics for the AMBER sum. Defaults to 1.2

SCNB The value by which to scale 1-4 van der Waals for the AMBER sum. Defaults to 2.0.

QM_ENERGY_UNITS The unit of energy in the quantum data file. This will depend on your quantum package and settings used for the single point calculations.

= **HARTREE** Default

= **KCALMOL**

= **KJMOL**

WRITE_ENERGY Saves the final AMBER energy and the quantum data for each structure to a file. Plotting these data is useful in verifying the results of the fitting and identifying any problem structures. See [12.3.4](#) for more on how to verify the accuracy of results.

WRITE_FRCMOD When the fitting is complete, the parameters will be saved in a force field modification file at this location in addition to displaying them in standard output. This file may be used with leap to create a new *prmtop*. If no value is specified the file will not be created.

12.2.5. Algorithm options

Paramfit implements two minimization algorithms: a simplex and a genetic algorithm (GA). Each algorithm has its own strengths and weaknesses, and choosing the correct algorithm for a given problem is important for achieving a good fit within a reasonable amount of time.

The genetic algorithm starts with a randomly generated solution set, which it recombines and alters in ways similar to evolution, converging to an optimum after a number of “generations” have passed without improvement. Currently, this algorithm requires many more evaluations of the AMBER sum than the simplex algorithm, making it slower for some problems. However,

12. paramfit

the GA excels on sample sets that are not as well defined, and outperforms the simplex algorithm when a large number of parameters are to be changed, especially in systems where some parameters are interdependent, such as molecules with multiple dihedrals.

The GA will start with many initial randomly generated sets of parameters. It will then determine which are the best by evaluating the AMBER sum, select them for recombination to produce a new set of parameters, randomly alter a few parameters slightly to prevent premature convergence, and iterate until convergence has been reached.

Choose the genetic algorithm if you wish to optimize more than three parameters or do not have a very good sampling of the parameters to be fit. The GA also requires the following options:

OPTIMIZATIONS The integer number of possible optimizations the algorithm will use. Analogous to the population size in evolution; larger values require more function evaluations and are slower but produce better results, and smaller ones will delay convergence. In general, choose the largest value your hardware and/or patience will tolerate. Defaults to 20.

SEARCH_SPACE If positive, the algorithm will search for new parameters for everything except dihedral phases within this percentage of the original value, where 1.0 will search within $\pm 100\%$ of the value found in the input *prmtop*. See 12.3 for examples of how to use this variable. Defaults to searching over the entire range of valid values and ignoring the original value in the *prmtop*.

MAX_GENERATIONS The maximum number of iterations the algorithm is allowed to run before it returns the best non-converged optimization. Defaults to 10000.

GENERATIONS_TO_CONV The number of iterations in a row that must pass without improvement in the best parameter set for the algorithm to be considered converged. The value will not be checked until 100 generations have passed, to prevent premature convergence. Set to a larger value for a longer but potentially more accurate run. Defaults to 1000, which may be too large for many systems.

The simplex algorithm starts at an initial set of parameters and moves “downhill” iteratively, converging when the improvement from one step to another becomes negligible. The simplex algorithm is generally faster than the GA, and excels at well-defined systems with a small number of dimensions. This algorithm requires a very well-defined sample space, and the input structures should contain a good range over all the bonds, angles, and dihedrals that are to be optimized. Otherwise, the algorithm tends to wander and will converge in badly defined areas of the sample set. In smaller, well-defined systems with only a few parameters, this algorithm will outperform the GA.

Choose the simplex algorithm if you wish to fit only a few parameters and have a large number of input conformations, and specify the following options:

BONDFC_dx Intrinsic length of parameter space for minimization. Used to determine the size of the steps to construct the initial simplex. Should be large enough that the steps sample a sufficiently large area but small enough to not move outside of normal parameter range. Bond force constant step size defaults to 5.0.

BONDEQ_dx Bond equilibrium length step size. Defaults to 0.02.

ANGLEFC_dx Angle force constant step size. Defaults to 1.0.

ANGLEEQ_dx Angle equilibrium step size. Defaults to 0.05.

DIHEDRALBH_dx Dihedral force constant step size. Defaults to 0.2.

DIHEDRALN_dx Dihedral periodicity step size. Defaults to 0.01.

DIHEDRALG_dx Dihedral phase step size. Defaults to 0.05.

K_dx Step size for intrinsic difference constant. Defaults to 10.0.

CONV_LIMIT Floating point number that details the convergence limit for the minimization. The smaller the number, the longer the algorithm will take to converge but the results may be more accurate. Defaults to 1.0E-15.

12.2.6. Bounds Checking

In order to ensure that the algorithms can return meaningful results, bounds checking routines are included in *paramfit*. The bounds checking functionality ensures that the algorithm's results are reasonable given the initial sample set, and also makes sure that the sample set is well-defined.

Since bonds and angles are approximately harmonic, the algorithm's result is reasonable if it lies within a well-defined area of the sample set. Bonds and angle values are therefore checked after the algorithm has finished running. In order to properly fit dihedrals, sample structures should span the entire range of phases for each dihedral that is to be fit. Dihedral checking is therefore accomplished before the algorithm begins to conduct the fit.

Bounds checking defaults to halting execution of the program upon reaching a failing condition. It is not recommended that this behavior be disabled, since the results of the fit are most likely inaccurate. Using the fitted parameters anyway will probably result in an inaccurate depiction of the molecule. Properly represented parameters in the input structures are crucial for a valid fit. Instead of using the parameters, fix the input structures so that data are provided in the missing ranges, which will be stated in the error message, and rerun the program twice: first in **CREATE_INPUT** mode to obtain quantum energies for the added structures and then in **FIT** mode to redo the fit.

If you **know** that your input structures describe the parameters to be fit quite well, the selectivity of the bounds checking can be altered by the specifying the following options in the job control file. Use these options with caution, and verify the generated parameters carefully.

CHECK_BOUNDS

- = **ON** The recommended and default option. This will halt execution when the bounds check fails.
- = **WARN** Continue upon reaching a bounds failure condition, but output a warning. Do not use the parameters generated by this fit! Use the error message and other results to determine if they are reasonable.

12. *paramfit*

BOND_LIMIT Fitting results for bond lengths that are this many Angstroms away from the closest approximation in the input structures will result in a failing condition. Defaults to 0.1.

ANGLE_LIMIT Fitting results for angles that are more than this many radians away from the closes approximation in the input structures will result in a failing condition. Defaults to 0.05π .

DIHEDRAL_SPAN The entire range of valid dihedral angles, 0 to π , for each dihedral that is to be fit should be spanned by this many input structure values, otherwise a failing condition will result. Defaults to 12, meaning that there needs to be a dihedral in every $\frac{\pi}{12}$ radian interval of the valid range.

12.3. Examples

12.3.1. Setting up to fit

The fitting process with *paramfit* follows a specific order. Example job control files for each step and a description of the step follow.

First, write a job control file to create the input structures and run *paramfit*:

```
RUNTYPE=CREATE_INPUT
NSTRUCTURES=50
QMFILEFORMAT=GAUSSIAN
QMHEADER=Gaussian.header
$AMBERHOME/bin/paramfit -i Job_Control.in -p prmtop -c mdcrd
```

After all 50 input files have been created, run the quantum program on them. Once it's finished, extract the quantum energies from the output files using the provided script. Since the example used Gaussian:

```
$AMBERHOME/AmberTools/src/paramfit/scripts/extract_gaussian.x \
output_directory energies.dat
```

Now, or while the quantum jobs are running since neither the energies nor the structures are needed yet, determine which parameters are to be fit and save them.

```
RUNTYPE=SET_PARAMS
PARAMETER_FILE_NAME=saved_params
$AMBERHOME/bin/paramfit -i Job_Control.in -p prmtop
```

Now the quantum energies to fit have been obtained and the parameters to fit have been set, and the fitting process may begin.

12.3.2. Fitting K

The first step in fitting is determining the value of K for a system. A job control file that will only fit K follows:

```

RUNTYPE=FIT
PARAMETERS_TO_FIT=K_ONLY
FITTING_FUNCTION=SIMPLEX

```

Then,

```
$AMBERHOME/bin/paramfit -i Job_Control.in -p prmtop -c mdcrd -q energies.dat
```

Take this value of K and put it back in the job control file when conducting the actual fit.

```

RUNTYPE=FIT
PARAMETERS_TO_FIT=LOAD
PARAMETER_FILE_NAME=saved_params
FITTING_FUNCTION=GENETIC
OPTIMIZATIONS=500
GENERATIONS_TO_CONV=150
WRITE_FRCMOD=fitted_params.frcmod

```

And call *paramfit* just as before. This example fit will create a force field modification file that can later be read into *leap* to create a new *prmtop* with the modified parameters for the molecule.

12.3.3. Improving a fit iteratively

The genetic algorithm does a good job improving on the initial values that it is given, but can converge away from the actual minimum. If you suspect that this is happening, *paramfit* can be run several times to produce better results by varying the `SEARCH_SPACE` parameter of the genetic algorithm. This parameter determines how far away from the initial values the algorithm will start to look for better answers. Setting it to a large value can help the algorithm escape local minima, but it may be slower to converge.

Start with `SEARCH_SPACE=-1` to have the algorithm search within the entire valid range for each parameter. If you want to search only around the existing values in the input *prmtop*, set it to a positive value. The algorithm will then search within plus or minus that fractional difference of the original value for the new results. For example, `SEARCH_SPACE=0.14` will search within $\pm 14\%$ of the original value.

If you have some values that you want to search the entire space for but not others, set those values to zero in your *prmtop*. This can be done with *xleap* and/or a force field modification file, or by (carefully) editing the *prmtop* yourself if you understand which values you need to change. Make sure to carefully check the output to see what *paramfit* thinks the initial parameters are!

12.3.4. Evaluating Results

When using *paramfit*, it is important to verify the accuracy of the fitted parameters for your input structures. The `WRITE_ENERGY` option in the Job Control file is useful for this. Set it to a filename and *paramfit* will write the final AMBER energy of each structure next to the quantum energy for the same structure in a file that can be easily graphed.

12. *paramfit*

If you have gnuplot, a script has been provided to quickly show each structure's energies. Assuming your energy file is named `energy.dat`:

```
$AMBERHOME/AmberTools/src/paramfit/scripts/plot_energy.x energy.dat
```

The resulting graph makes the identification of problem structures much easier, and gives a good visualization of the fit. In general, carefully validate parameters generated by *paramfit* against other data before conducting large simulations.

13. Miscellaneous utilities

13.1. ambpdb

NAME ambpdb - convert amber-format coordinate files to pdb format

SYNOPSIS

```
ambpdb [ -p prmtop-file ][ -tit title ] [ -pqr|-bnd|-atm ]  
      [ -aatm ] [-bres ] [-noter] [-ext] [-offset #] [-bin] [-first]
```

ambpdb is a filter to take a coordinate "restart" file from an AMBER dynamics or minimization run (on STDIN) and prepare a pdb-format file (on STDOUT). The program assumes that a *prmtop* file is available, from which it gets atom and residue names.

OPTIONS

- help* Print a usage summary to the screen.
- tit* The title, if given, will be output as a REMARK at the top of the file. It should be protected by quotes or double quotes if it contains spaces or special characters.
- pqr* If *-pqr* is set, output will be in the format needed for the electrostatics programs that need charge and radius information.
- atm* creates files used by Mike Connolly's surface area/volume programs.
- bnd* creates a file that lists the bonds in the molecule, one per line.
- aatm* This switch controls whether the output atom names follow Amber or Brookhaven (PDB) formats. With the default (when this switch is not set), atom names will be placed into four columns following the rules used by the Protein Data Base in Version 3.
- bin* If *-bin* is set, an unformatted (binary) "restart" file is read instead of a formatted one (default). Please note that no detection of the byte ordering happens, so binary files should be read on the machine they were created on.
- bres* If *-bres* (Brookhaven-residue-names) is not set (the default), Amber-specific atom names (like CYX, HIE, RG5, etc.) will be kept in the pdb file; otherwise, these will be converted to PDB-standard names (CYS, HIS, G, in the above example). Note

13. Miscellaneous utilities

that setting *-bres* creates a naming ambiguity between protonated and unprotonated forms of amino acids.

If you plan to re-read the pdb file back into Amber programs, you should use the default behavior; for programs that demand stricter conformance to Brookhaven standards, set *-bres*.

- first* If *-first* is set, a pdb file augmented by additional information about hydrogen bonds, salt bridges, and hydrophobic tethers is generated, which can serve as input to the stand alone version of the FIRST software by D. J. Jacobs, L. A. Kuhn, and M. F. Thorpe to analyze the rigidity / flexibility of protein and nucleic acid structures.[196, 197] The criteria to include hydrophobic tethers differ for protein and nucleic acid structures. Note that currently not all modified RNA nucleosides are explicitly considered and that DNA structures are treated according to a parametrization derived for RNA structures. Details about the RNA parametrization can be found in ref.[198] .
- noter* If *-noter* is set, the output PDB file not include TER cards between molecules. Otherwise, TER cards will be added whenever there is not bond between adjacent residues. Note that this means there will be a TER card between each water molecule, for example, unless *-noter is set*. The PDB is idiosyncratic about TER cards: they are generally present between separate protein chains, but generally not present between cofactors or solvent molecules. This behavior is not mimicked by *ambpdb*.
- ext* Use the “extended” pdb information in the *prmtop* file to recover the chain IDs and residue numbers that were present in the original pdb file used to make the *prmtop* file.
- offset* If a number is given here, it will be added to all residue numbers in the output pdb file. This is useful if you want the first residue (which is always "1" in an Amber *prmtop* file, to be a larger number, (say to more closely match a file from Brookhaven, where initial residues may be missing). Note that the number you provide is one less than what you want the first residue to have.
- Residue numbers greater than 9999 will not "fit" into the Brookhaven format; *ambpdb* actually prints $\text{mod}(\text{resno}, 10000)$; that is, after 9999, the residue number re-cycles to 0.

FILES Assumes that a *prmtop* file (with that name, or the one given in the *-p* option) exists in the current directory; reads AMBER coordinates from STDIN, and writes pdb-file to STDOUT.

BUGS Inevitably, various niceties of the Brookhaven format are not as well supported as they should be. The *protonate* program can be used to fix up hydrogen atom names, but that functionality should really be integrated here. There is no good solution to the PDB problem of using the same residue name for different chemical species; depending on how the output file is to be used, the two options supported (setting or not setting *-bres*)

may or may not suffice. Radii used for the *-pqr* option are hardwired into the code, requiring a recompilation if they are to be changed. Atom name output may be incorrect for atoms with two-character atomic symbols, like calcium or iron. The *-offset* flag is a very limited start toward more flexible handling of residue numbers; in the future (we hope!) Amber *prmtop* files will keep track of the "original" residue identifiers from input *pdb* files, so that this information would be available on output.

13.2. *reduce*

Reduce is a program for adding hydrogens to a Protein DataBank (PDB) molecular structure file. It was developed by J. Michael Word at Duke University in the lab of David and Jane Richardson. *Reduce* is described in: Word, et. al. (1999) Asparagine and Glutamine: Using Hydrogen Atom Contacts in the Choice of Side-chain Amide Orientation, *J. Mol. Biol.* **285**, 1733-1747.

Both proteins and nucleic acids can have hydrogens added. HET groups can also be processed as long as the atom connectivity is provided. A slightly modified version of the connectivity table provided by the PDB is included. The latest version of *reduce* is available at <http://kinemage.biochem.duke.edu/>. The version bundled with AmberTools 1.4 is *reduce.3.14.080821*. See the files in \$AMBERHOME/AmberTools/src/reduce for more information. The information below is taken from the *README.usingReduce.txt* file.

13.2.1. Running *reduce*

In most circumstances, the recommended command when using *reduce* to add hydrogens to a PDB file and standardize the bond lengths of existing hydrogens is

```
reduce -build coordfile.pdb > coordfileH.pdb
```

which includes the optimization of adjustable groups (OH, SH, NH₃⁺, Met-CH₃, and Asn, Gln and His sidechain orientation). When speed is important, the *-build* option can be dropped; hydrogens will still be added, but not His side-chain NH hydrogens, and side-chains will not be flipped. For even greater speed, but even less accuracy, adding *-nooh* and *-noadj* will skip the OH and SH hydrogens and eliminate optimization altogether. Input is from the specified PDB format coordinate file and the new, updated PDB coordinates are written to "standard output", here redirected to a file with the *'>'* symbol.

Disulfides, covalent modifications, and connection of the ribose-phosphate nucleic acid backbone, are recognized and any hydrogens eliminated by bonding are skipped. When an amino acid main-chain nitrogen is not connected to the preceding residue or some other group, *reduce* treats it as the N-terminus and constructs an NH₃⁺ only if the residue number is less than or equal to an adjustable limit (1, by default). Otherwise, it considers the residue to be the observable beginning of an actually-connected fragment and does not protonate the nitrogen. *Reduce* does not protonate carboxylates (including the C-terminus) because it does not specifically consider pH, instead modeling a neutral environment.

Hydrogens are positioned with respect to the covalently bonded neighbors and these are identified by name. Nonstandard atom names are the primary cause of missing or misplaced

13. Miscellaneous utilities

hydrogens. If reduce tries to process a file which contains hydrogens with nonstandard names, the existing hydrogens may not be recognized and may interfere with the generation of new hydrogens. The solution may be to remove existing hydrogens before further processing.

Hydrogens can be removed from a pdb format file with reduce.

```
reduce -trim labcH > labc
```

This can be used, for example, to update the orientation of Asn/Gln/His side chains where the H atoms are not wanted; first build the hydrogens and then trim them back out. Trimming can occasionally be fooled if a hydrogen has been given a non-standard name. The most common example of this comes from left-justified atom names: gamma hydrogens masquerade as mercury atoms! In this case, manual editing may be required.

13.2.2. General input flags

The following brief description of the command line flags is displayed with the -h flag:

```
$ reduce -h
reduce: version 2.20 6/03/03, Copyright 1997-2003, J. Michael Word
arguments: [-flags] filename or -
Adds hydrogens to a PDB format file and writes to standard output.
(note: By default, HIS sidechain NH protons are not added. See -BUILD)
Flags:
-Trim remove (rather than add) hydrogens
-NOOH remove hydrogens on OH and SH groups
-OH add hydrogens on OH and SH groups (default)
-HIS create NH hydrogens on HIS rings
-FLIPs allow complete ASN, GLN and HIS sidechains to flip
      (usually used with -HIS)
-NOHETH do not attempt to add NH proton on Het groups
-ROTNH3 allow lysine NH3 to rotate (default)
-NOROTNH3 do not allow lysine NH3 to rotate
-ROTEXist allow existing rotatable groups (OH, SH, Met-CH3) to rotate
-ROTEXOH allow existing OH & SH groups to rotate
-ALLMethyls allow all methyl groups to rotate
-ONLYA only adjust 'A' conformations (default)
-ALLALT process adjustments for all conformations
-NOROTMET do not rotate methionine methyl groups
-NOADJust do not process any rot or flip adjustments
-BUILD add H, including His sc NH, then rotate and flip groups
      (except for pre-existing methionine methyl hydrogens)
      (same as: -OH -ROTEXOH -HIS -FLIP)
-Keep keep bond lengths as found
-NBonds# remove dots if cause within n bonds (default=3)
-Model# which model to process (default=1)
-Nterm# max number of nterm residue (default=1)
```

```

-DENSity#.# dot density (in dots/A^2) for VDW calculations (Real)
-RADius#.# probe radius (in A) for VDW calculations (Real, default=0)
-OCCcutoff#.# occupancy cutoff for adjustments (default=0.01)
-H2OCCcutoff#.# occupancy cutoff for water atoms (default=0.66)
-H2OBcutoff# B-factor cutoff for water atoms (Integer, default=40)
-PENalty#.# fraction of std. bias towards original orientation
-HBREGcutoff#.# over this gap regular HBonds bump (default=0.6)
-HBChargedcut#.# over this gap charged HBonds bump (default=0.8)
-BADBumpcut#.# at this gap a bump is 'bad' (default=0.4)
-METALBump#.# H 'bumps' metals at radius plus this (default=0.865)
-NONMETALBump#.# 'bumps' nonmetal at radius plus this (default=0.125)
-SEGIDmap "seg,c..." assign chainID based on segment identifier field
-Xplor use Xplor conventions for naming polar hydrogens
-NOCon drop conect records
-LIMIT# max num iter. for exhaustive search (default=100000)
-NOTICKs do not display the set orientation ticker during processing
-SHOWScore display scores for each orientation considered during proce
-FIX "filename" if given, file specifies orientations for adjustable groups
-DB "filename" file to search for het info
    note: can also redirect with unix environment variable: REDUCE_HET_DICT
-Quiet do not write extra info to the console
-REFeRence display citation reference
-Help more extensive description of command line arguments

```

13.2.3. Fixing an orientation

At times it is useful to control the flip state or rotation angle of an adjustable group when adding hydrogens, either because the correct orientation has already been established, allowing the optimization time to be reduced, or because a non-optimal orientation is sought.

One of the command line flags (-fix myfile.txt) takes a file containing information about which conformation to set for one or more adjustable groups. The colon delimited format is similar to the orientation data that reduce prints in the header file

```
action:residueID:comment
```

(one line for each group to be fixed) and because spacing matters in the residue identifier string, the easiest way to produce this file is to copy and edit USER MOD records from reduce output. The action can be one of three kinds, depending on residue type: O to leave in the original orientation, F to flip the orientation, and R# to rotate a dihedral to an angle of #deg. Using either O or F with His sidechains allows the protonation state to vary; to specify a particular orientation and protonation state use F# where # is the number of the state (1, 2 or 3 for the original orientation with H (1) only on NE2, (2) only on ND1, or (3) doubly protonated; 4-6 for the corresponding three flipped states).

13.2.4. Cliques

The current version of reduce uses brute-force enumeration to optimize the conformations of adjustable groups. If a 'clique' of adjustable groups is too large ($> \sim 7$) this sort of search technique is inadequate—the enumeration will be abandoned and these groups will be left in their original conformations. The cutoff point is based on the total number of permutations, which the user can control with the -limit# option. Although we are considering more powerful search techniques for these situations, some work-around strategies have been developed.

First check to see if distinct chainIDs are provided for each chain. Reduce does not support files which specify chain information only in the segID field and can get confused.

Examination of the clique may reveal that the orientations of one or more groups are obvious; for instance, they may interact with obligate H-bond donors or acceptors. By fixing the orientation of these groups (as described above), the total number of permutations is reduced. This is especially effective if it breaks the clique into smaller sum-cliques or singletons.

An alternative way to break up cliques is to rotate all the methionine CH₃s and lysine/N-terminus NH₃+s in an initial pass, then keep them fixed in a second pass.

```
reduce -nooh inputfile | reduce -build -norotmet -norotnh3 - > outputfileH
```

The single dash towards the end of the command line tells reduce to read data piped ('|') from the first pass rather than from a file. A few NH₃⁺ H-bonds may have inferior geometry with this two pass approach but the result is otherwise comparable to using -build alone and can be combined with the previous approach, if necessary. With this technique, unusual cliques requiring many hours to process have been converted into several smaller problems which were all solved in a matter of minutes.

13.2.5. Contact

If you use reduce, I would appreciate any comments you send my way.

J. Michael Word

e-mail: mike.word@duke.edu voice: (919)483-3522

Richardson Lab, Biochemistry Department, Duke University, Durham, NC USA 27710

13.3. elsize

NAME

```
elsize - Given the structure, estimates its effective electrostatic size
         (parameter Arad ) need by the ALPB model.
```

SYNOPSIS

```
Usage: elsize input-pqr-file [-options]
-det an estimate based on structural invariants. DEFAULT.
```

```

-ell an estimate via elliptic integral (numerical).
-elf same as above, but via elementary functions.
-abc prints semi-axes of the effective ellipsoid.
-tab prints all of the above into a table without header.
-hea prints same table as -tab but with a header.
-deb prints same as -tab with some debugging information.
-xyz uses a file containing only XYZ coordinates.

```

DESCRIPTION

elsize is a program originally written by G. Sigalov to estimate the effective electrostatic size of a structure via a quick, analytical method. The algorithm is presented in detail in Ref. [199] You will need your structure in a pqr format as input, which can be easily obtained from the prmtop and inpcrd files using *ambpdb* utility described above:

```
ambpdb -p prmtop -pqr < inpcrd > input-file-pqr
```

After that you can simply do: *elsize input-file-pqr*, the value of electrostatic size in Angstroms will be output on stdout. The source code is in the src/etc/ directory, its comments contain more extensive description of the options and give an outline of the algorithm. A somewhat less accurate estimate uses just the XYZ coordinates of the molecule and assumes the default radius size of for all atoms:

```
elsize input-file-xyz
```

This option is not recommended for very small compounds. The code should not be used on structures made up of two or more completely disjoint" compounds – while the code will still produce a finite value of *Arad*, it is not very meaningful. Instead, one should obtain estimates for each compound separately.

13.4. Utilities for Molecular Crystal Simulations

David S. Cerutti

Simulations of biomolecular crystals are in principle no different than any of the simulations that AMBER does in periodic boundary conditions. However, the setup of these systems is not trivial and probably cannot be accomplished with the LEaP software. Of principal importance are the construction of the solvent conditions (packing precise amounts of multiple solvent species into the simulation cell), and tailoring the unit cell dimensions to accommodate the inherently periodic nature of the system. The LEaP software, designed to construct simulations of molecules in solution, will overlay a pre-equilibrated solvent mask over the (biomolecular) solute, tile that mask throughout the simulation cell, and then prune solvent residues which clash with the solute. The result of this procedure is a system which will likely contract under constant pressure dynamics as the pruning process has left vacuum bubbles at the solute:solvent interface. Simulations of biomolecular crystals require that the simulation cell begin at a size corresponding to the crystallographic unit cell, and deviate very little from that size over the

13. Miscellaneous utilities

course of equilibration and onset of constant pressure dynamics. This demands a different strategy for placing solvent in the simulation cell. Four programs in the *AmberTools* release are designed to accomplish this. An example of their use is given in a web-based tutorial at <http://ambermd.org/tutorials/advanced/tutorial13/XtalTutor1.html>.

For brevity, only basic descriptions of the programs are given in this manual. All of the programs may be run with command line input; the input options to each program may be listed by running each program with no arguments.

13.4.1. UnitCell

A macromolecular crystal contains many repeating unit cells which stack like blocks in three dimensional space just as simulation cells do in periodic boundary conditions. Each unit cell, in turn, may contain multiple symmetry-related clusters of atoms. A PDB file contains one set of coordinates for the irreducible unit of the crystal, the “asymmetric unit,” and also information about the crystal space group and unit cell dimensions. The *UnitCell* program reads PDB files, seeking the SMTRY records within the REMARKs to enumerate the rotation and translation operations which may be applied to the coordinates given in the PDB file to reconstruct one complete unit cell.

13.4.2. PropPDB

Simulations in periodic boundary conditions require a minimum unit cell size: the simulation cell must be able to enclose a sphere of at least the nonbonded direct space cutoff radius plus a small buffer region for nonbonded pairlist updates. Many biomolecular crystal unit cells come in “shoebox” dimensions that may have one very short side; many unit cells are also not rectangular but triclinic, meaning that the size of the largest sphere they can enclose is further reduced. For these reasons, and perhaps to ensure that the rigid symmetry imposed by periodic boundary conditions does not create artifacts (crystallographic unit cells are equivalent when averaged over all time and space, but not necessarily identical at any given moment), it may be necessary to include multiple unit cells within the simulation cell. This is the purpose of the *PropPDB* program: to propagate a unit cell in one or more directions so that the complete simulation cell meets minimum size requirements.

13.4.3. AddToBox

The *AddToBox* program handles placement of solvent within a crystal unit cell or supercell (as may be created by *PropPDB*). As described in the introduction, the basic strategy is to place solvent such that added solvent molecules do not clash with biomolecule solutes, but *may* clash with one another initially. This compromise is necessary because enough solvent must be added to the system to ensure that the correct unit cell dimensions are maintained in the long run, but it is not acceptable to place solvent within the interior of a biomolecule where it might not belong and never escape.

The *AddToBox* program takes a PDB file providing the coordinates of a complete biomolecular unit cell or supercell (argument -c), the dimensions by which that supercell repeats in space (-X, -Y, -Z for the three box edge lengths, and -a1, -b1, and -g1 for the three unit cell angles), a

PDB file describing the solvent residue to add (argument -a), and the number of copies of that solvent molecule to add (argument -na). *AddToBox* inherently assumes that the biomolecular unit cell it is initially presented may contain some amount of solvent already, and according to the AMBER convention of listing macromolecular solute atoms first and solvent last assumes that the first -P atoms in the file are the protein (or biomolecule). *AddToBox* will then color a very fine grid “black” if the grid point is within a certain distance of a biomolecular atom (argument -RP) or other solvent atom (argument -RW); the grid is “white” otherwise (the grid is stored in binary for memory efficiency). *AddToBox* will then make a copy of the solvent residue and randomly rotate and translate it somewhere within the unit cell. If all atoms of the solvent residue land on “white” grid voxels, the solvent molecule will become part of the system and the grid around the newly added solvent will be blacked out accordingly. If the solvent molecule cannot be placed, this process will be repeated until a million consecutive failures are encountered, at which point the program will terminate. If *AddToBox* has not placed the requested number of solvent molecules by the time it terminates, the -V option can be used to order the program to recursively call itself with progressively smaller solvent buffer distances until all the requested solvent can be placed. The output of the *AddToBox* program is another PDB named by the -o option.

Successful operation of *AddToBox* may take practice. If multiple solvent species are required, as is the case with heterogeneous crystallization solutions, *AddToBox* may be called repeatedly with each input molecular cell being the previous call’s output. When considering crystal solvation, the order of addition is important! It is recommended that rare species, such as trace buffer reagents, be added first, with large -RW argument to ensure that they are dispersed throughout the available crystal void zones. Large solvent species such as MPD (an isohexane diol commonly used in crystallization conditions) or should be added second, and with a sufficiently large -RW argument that methyl groups and ring systems cannot become interlocked (which will likely lead to SHAKE / vlimit errors). Small and abundant species such as water should be added last, as they can go anywhere that space remains.

It is likely that the unobservable “void” regions between biomolecules in most crystals *do not* contain solvent species in proportion to their abundance in the crystallization solution—the vast majority of these regions are within a few Ångströms of some biomolecular surface, and different biomolecular functional groups will preferentially interact with some types of solvent over others. Also, in many crystals some solvent molecules *are* observed; in many of these, the amount of solvent observed is such that it would be impossible to pack other species into the unit cell in proportion to their abundances in the crystallization fluid. In these cases, we recommend estimating the amount of volume that must be filled with solvent *apart from solvent which has already been observed in the crystal*, and filling this void with solvent in proportion to the composition of the crystallization fluid. For example, if a crystal were grown in a 1:1 mole-to-mole water/ethanol mixture, and the crystal coordinates as deposited in the PDB contained 500 water molecules and 3 ethanol molecules, we would use *AddToBox* to add water and ethanol in a 1:1 ratio until the system contained enough solvent to maintain the correct volume during equilibrium dynamics at constant pressure.

Finally, it is difficult to estimate exactly how much solvent will be needed to maintain the correct equilibrium volume; the advisable approach is simply to make an initial guess and script the setup so that, over multiple runs and reconstructions, the correct system composition can be found. We recommend matching the equilibrium unit cell volume to within 0.3% to keep this

13. Miscellaneous utilities

simulation parameter within the error of most crystallographic measurements. While errors of 0.5-1% will show up quickly after constant pressure dynamics begin, a 10 to 20ns simulation may be needed to ensure that the correct equilibrium volume has been achieved.

13.4.4. ChBox

After the complex process of adding solvent, the LEaP program may be used to produce a topology and initial set of coordinates based on the PDB file produced by *AddToBox*. By using the *SetBox* command, LEaP will create a periodic system without adding any more solvent on its own. The only problem with using LEaP at this point is that the program will fail to realize that the system *does* tile in three dimensions if only the box dimensions are set properly. If visualized, the output of UnitCell / PropPDB will likely look jagged, but the output of *AddToBox*, containing lots of added water, will make it obvious how parts of biomolecules jutting out one face of the box fit neatly into open spaces on an opposite face. The topology produced by LEaP needs no editing; only the last line of the coordinates does. This can be done manually, but the *ChBox* program automates the process, taking the same coordinates supplied to *AddToBox* and grafting them into the input coordinates file.

13.5. ParmEd

ParmEd (*parmed.py*) is a topology file editor written in Python that enables high level control of the primary force field file in Amber: the prmtop file. ParmEd will modify the topology file and produce a new topology file that will work with *sander*, *pmemd*, and NAB programs, and provides options unavailable otherwise.

The principle capabilities unavailable through other methods are applying the mbondi3 GB radii optimized for the igb=8 GB model in *sander* and *pmemd*, and assigning specific van der Waals parameters for a pair of atoms without affecting the other pairs of those specific types.

13.5.1. Running parmed.py

parmed.py is used in a manner very similarly to *ptraj* and *cpptraj*.

```
Usage: parmed.py [Options] <prmtop> <input_file>
-h, --help          show this help message and exit
-d, --debug          Print how each action is parsed and show
                     which line of which file an error occurred on
                     (verbose tracebacks). OFF by default.
-e, --enable-interpreter
                     Allow the use of ! and !! to permit the user to
                     drop into a limited Python interpreter with access
                     to the topology file API. This is an advanced option,
                     use with care.
-q, --quiet          Disable verbose tracebacks. Reverses -d/--debug
-p PROMPT, --prompt=PROMPT
                     Which character/string to use as a command prompt
```


-n, --no-splash Prevent printing the greeting logo

Like with ptraj and cpptraj, if you do not supply the prmtop or the input_file, it will read the commands from STDIN as you type them. The keyword “go” launches the specified actions from STDIN. The debugging option enables the printing of detailed tracebacks. This adds the exact line number on the exact file that the error occurred on, which can be useful for some debugging if the error message is too confusing.

13.5.2. ParmEd commands (they are all case-insensitive)

13.5.2.1. addAtomicNumber

Usage: *addAtomicNumber*

Adds a section in the topology file with the flag ATOMIC_NUMBER in order to identify specific elements. Elements are matched based on their atomic masses in the MASS section of the topology file. An atom is assigned an element by matching it with the element on the periodic table whose atomic mass is closest to the atom in question. This approach should work for any atom whose mass is either unchanged from the LEaP output or if that atom’s mass has only been changed to one of its isotopes.

13.5.2.2. addDihedral

Usage: *addDihedral* <mask1> <mask2> <mask3> <mask4> <phi_k> <per> <phase> <scce> <scnb> [<type>]

Adds a dihedral term (will NOT replace an existing dihedral) between atoms in mask1, mask2, mask3, and mask4. The dihedral is defined around the bond between the atoms in mask2 and mask3. Each mask must define the same number of atoms. For mask1 defines atoms 1,2,3; mask2 defines atoms 11,12,13; mask3 defines atoms 21,22,23; and mask4 defines atoms 31,32,33, then 3 new dihedrals will be added. One between atoms 1, 11, 21, and 31, another between atoms 2, 12, 22, and 32, and a third between atoms 3, 13, 23, and 33. The dihedrals will be set with force constant *phi_k*, periodicity *per*, phase angle *phase*, 1-4 electrostatic scaling factor *scce* (this must be specified – the default Amber value is 1.2 and the default GLYCAM value is 1.0), the 1-4 van der Waals scaling factor *scnb* (this must be specified – the default Amber value is 2.0 and the default GLYCAM value is 1.0). The *type* is either “normal”, “multiterm”, or “improper”. “multiterm” simply means that 1-4 factors for that dihedral are not calculated, which should be true for all but the last term in a multiterm dihedral (so 1-4 interactions are only counted once) and in some ring systems with at most 6 atoms, since that could also lead to double-counting specific 1-4 interactions.

13.5.2.3. addExclusions

Usage: *addExclusions* <mask1> <mask2>

Allows you to add arbitrary exclusions to the exclusion list. Every atom in <mask2> is added to the exclusion list for each atom in <mask1> so that non-bonded interactions between those atom pairs will not be computed. NOTE that this ONLY applies to direct-space (short-range) non-bonded potentials. For PME simulations, long-range electrostatics between these atom pairs are still computed (in different unit cells).

13. Miscellaneous utilities

13.5.2.4. addLJType

Usage: *addLJType* <mask> [new_radius] [new_depth] [new_radius_14] [new_depth_14]

This command will assign all atoms specified in the given mask to a new van der Waals (VDW) atom type. Note that several different Amber atom types may in fact be the same VDW type, so this command is designed to give you control over changing just a single atom's (or single Amber atom type's) VDW parameters. Every atom specified in the mask will be given the SAME type (but different from every other atom in the topology file), even if their original VDW types are different. The parameters [new_radius] and [new_depth] are optional parameters that specify that atom's radius and well depth, which are combined with every other type's radius and depth via the canonical Amber combining rules. They default to the original value of the FIRST atom that is matched by the mask.

Note that for *chamber*-created topology files (ONLY), each atom type has separate 1-4 parameters that may be specified as well. Unspecified values will be taken from the default parameters of the first atom type as described above. Any attempt to supply the 1-4 parameters on a normal topology created with leap will result in an error.

See the command *printLJTypes* for additional information here. You can use this command to see if *addLJType* may be necessary for what you're trying to do.

13.5.2.5. change

Usage: *change* <property> <atom_mask> <new_value>

This command allows you to change the value of an atom's property for every atom in a given mask to a new value. The allowed atomic properties you can modify are the CHARGE (given in units of elementary atomic charges), MASS (in g/mol), RADII (in Angstroms, these are the GB radii), SCREEN (the GB screening parameters), ATOM_NAME, and AMBER_ATOM_TYPE (this is NOT the van der Waals type). Every atom in the mask will be given the same new_value.

NOTE: The *prmtop* utility used here stores the partial CHARGE array in terms of elementary atomic charges. All charges are multiplied by 18.2223 prior to being written to any new topology file (and is divided by that number when read in from a topology file). Therefore, if you are changing specific atomic charges in this case, specify new charges in elementary atomic charges.

NOTE: This command gives you access to specific atoms. If you want to change all of the GB radii to be compatible with a specific GB model, see the *changeRadii* command.

13.5.2.6. changeLJPair

Usage: *changeLJPair* <mask1> <mask2> Rmin Depth

This command changes a specific pairwise interaction between the atom type of the atoms in mask1 (these must all be the same type) and the atoms in mask2 (these must all be the same type as well). Rmin and Depth are the pre-combined values of these variables, which allows you to define your own combining rules for a specific pair of atoms.

If you want to see which atoms this command will affect, you can use the *printLJTypes* with either of the given masks to get a list of atoms that share the same type as the atoms in that mask.

This command is similar to NBFIX available through CHARMM.

13.5.2.7. changeLJ14Pair

Usage: *changeLJ14Pair* <mask1> <mask2> <Rmin_14> <Depth_14>

This command is similar to *changeLJPair* above, except it alters the 1-4 Lennard Jones terms only. Note that this command is only available for *chamber*-created topology files, and will result in an error if applied to a normal topology created with leap.

13.5.2.8. changeLJSingleType

Usage: *changeLJSingleType* <mask> <Rmin> <Depth>

This command allows you to change the radius and well depth of particular nonbonded atom types. It will set new values for each interaction the selected type has with every other atom type (irrespective if *changeLJPair* altered one of these terms before).

13.5.2.9. changeProtState

Usage: *changeProtState* <mask> <state #>

Changes the protonation state of a residue that is titratable via constant pH simulations in Amber. <mask> must match all atoms of one, and only one, titratable residue as defined in \$AMBERHOME/AmberTools/src/etc/cpin_data.py. As of Amber 11, current titratable residues include AS4, GL4, HIP, LYS, CYS, TYR, and the basic nucleic acid residues DA, DC, DG, DT, A, C, G, and U. See comments in cpin_data.py for descriptions of which state numbers correspond to which protonation charge state.

13.5.2.10. changeRadii

Usage: *changeRadii* <parameter_set>

Parameter set is one of the following: bondi, mbondi, mbondi2, mbondi3, amber6. This command will reset all of the intrinsic GB radii to the specified set without having to recreate a topology file through leap. (Also, mbondi3 is only available here)

13.5.2.11. checkValidity

Usage: *checkValidity*

Checks through the topology file to catch various errors that I have seen pop up from time to time. If you get a weird error using sander or pmemd, it may be worth checking that the topology file is valid here (or if you did substantial modifications yourself)

13.5.2.12. combineMolecules

Usage: *combineMolecules* <mol_id1> [mol_id2]

This command will combine the two adjacent molecule numbers mol_id1 and mol_id2. The molecule sequences begin at 1 (that is, the first molecule is 1, not 0). This is useful if you want to image a couple molecules together, even if they have no covalent bonds defined between them. For instance, if you have 2 strands of DNA that you don't want to be imaged separately, this command will combine them and force sander/pmemd to think of them as a single molecule. Likewise for a protein system with a bound ligand, or anything else like this.

13. Miscellaneous utilities

13.5.2.13. defineSolvent

Usage: *defineSolvent* <residue_list>

This command will allow you to define custom solvent residues. The residue_list must be a comma-separated list with no whitespace separating the residue names. This is important for the proper determination of the SOLVENT_POINTERS and ATOMS_PER_MOLECULE sections of the topology file. By default, HOH and WAT residues are recognized as solvent.

13.5.2.14. go

Usage: *go*

Stop reading commands and execute every command that has come before. This has exactly the same effect as the End Of File (EOF) character. All commands in a script after “go” will be ignored. Placing “go” as the last line of a script is the same as not including it at all (since the next line contains EOF, which executes the same behavior). Thus, you can get the same behavior from the interactive session by either typing “go” or sending the EOF character (which on unix is CTRL-D)

13.5.2.15. help

Usage: *help* [action]

This command does one of two things. If *action* is not specified, a list of available commands along with their short usage statement is displayed in a nicely formatted table. If *action* is provided and that action exists, a usage statement along with a short description is printed. This is a useful reference for quick interactive sessions.

13.5.2.16. loadRestrt

Usage: *loadRestrt* <restart_filename>

This command takes an inpcrd or a restart file to assign coordinates to each of the atoms. This is currently only needed for the writeOff function, as those files require coordinates.

13.5.2.17. netCharge

Usage: *netCharge* [mask]

This command will calculate the net charge of all atoms belonging to a specific mask. If no mask is provided, it returns the net charge of all atoms in the topology file.

13.5.2.18. outparm

Usage: *outparm* <prmtop_name> [restrt_name>]

This command is just like parmout, except it can occur as many times as you want it to, and that topology file is written in the order in which that command is placed in the input file or read from STDIN (similar to outtraj in cpptraj). If you provide a file name for restrt_name, parmed.py will also write a valid restart file from the provided initial coordinates and velocities (if present) from the restart file added via the loadRestrt command. It will include velocities if

they were present in the initial restart file. Note this is most useful when used in conjunction with the “strip” command. If all solvent is stripped, the box information will be discarded. If you do not strip all solvent molecules, the box info will remain unchanged from the original (even if you strip a large number of solvent molecules). If you removed a large number of solvent molecules, take care to re-equilibrate the density before continuing with production dynamics.

13.5.2.19. parmout

Usage: *parmout* <prmtop_name> [<restrt_name>]

This command is similar to trajout in cpptraj and ptraj. It is ALWAYS the last command executed, and only the last one is executed. It writes a topology file with all of the modifications made to it during the course of the whole ParmEd session. If you provide a file name for restrt_name, parmed.py will also write a valid restart file from the provided initial coordinates and velocities (if present) from the restart file added via the loadRestrt command. It will include velocities if they were present in the initial restart file. Note this is most useful when used in conjunction with the “strip” command. If all solvent is stripped, the box information will be discarded. If you do not strip all solvent molecules, the box info will remain unchanged from the original (even if you strip a large number of solvent molecules). If you removed a large number of solvent molecules, take care to re-equilibrate the density before continuing with production dynamics.

13.5.2.20. printAngles

Usage: *printAngles* <mask>

This will print out every angle that involves at least one atom specified by <mask>.

13.5.2.21. printBonds

Usage: *printBonds* <mask>

This will print out every bond that involves at least one atom specified by <mask>.

13.5.2.22. printDihedrals

Usage: *printDihedrals* <mask>

This will print out every dihedral that involves at least one atom specified by <mask>. It labels multiterm dihedrals with an M and improper dihedrals with an I in the output.

13.5.2.23. printDetails

Usage: *printDetails* <mask>

This command prints atomic details of every atom matching a given mask (atom number, residue number, residue name, atom name, atom type, van der Waals radius, van der Waals well depth, mass, and charge) in standard Amber units. This is a useful command to make sure that every atom you think belongs in a mask actually does belong in the mask (and that no atoms were missed). The mask parser implemented in Python here is (mostly) a copy of

13. Miscellaneous utilities

ptraj's mask parser implemented in C, but some parts had to be rewritten slightly to adjust for different syntaxes of the two languages. Note, distance-based criteria is not yet implemented in this parser.

13.5.2.24. printFlags

Usage: *printFlags*

This command prints every %FLAG present in the topology file (see <http://ambermd.org/formats.html> for a description of what each section labelled with these FLAGS means).

13.5.2.25. printInfo

Usage: *printInfo* <flag>

This command just prints out all of the data in a given prmtop %FLAG (see <http://ambermd.org/formats.html> for details)

13.5.2.26. printLJTypes

Usage: *printLJTypes* [mask]

This command prints out each atom's van der Waals, or Lennard-Jones type in the mask, as well as every other atom that shares the same atom type as any type in the mask. If no mask is provided, it prints out that information for every atom. This is particularly useful if you want to see if changing a particular pair interaction will affect more atoms than you expect. If it turns out that you wish to treat some of the atoms that share the same VDW type differently from one another, you will have to "separate" them by using the *addLJType* command before modifying them.

13.5.2.27. printPointers

Usage: *printPointers*

This command will print every pointer along with its name and a short description in the topology file. Solvated topology files will also have their SOLVENT_POINTERS printed in the same manner.

13.5.2.28. quit

Usage: *quit*

This command will halt *parmed.py* in its tracks. It is effectively the same as *go* except it will NOT execute any *parmout* command (although any *outparm* command used prior to quitting has already been executed)

13.5.2.29. scee

Usage: *scee* <value>

Allows the user to set/change the value of the electrostatic scaling constant that will be used to scale 1-4 electrostatic interactions. This needs to be set in the prmtop since it was removed from the *sander/pmemd* input file in Amber 11. This will apply <value> to all dihedral terms.

13.5.2.30. **scnb**

Usage: *scnb* <value>

Allows the user to set/change the value of the VDW scaling constant that will be used to scale 1-4 VDW interactions. This needs to be set in the prmtop since it was removed from the *sander/pmemd* input file in Amber 11. This will apply <value> to all dihedral terms.

13.5.2.31. **setAngle**

Usage: *setAngle* <mask1> <mask2> <mask3> <k> <THETeq>

Changes (or adds a non-existent) angle in the topology file. Each mask must select the same number of atoms, and an angle will be placed between the atoms in mask1, mask2, and mask3 (one angle between atom1 from mask1, atom1 from mask2, and atom1 from mask3, another angle between atom2 from mask1, atom2 from mask2, and atom2 from mask3, etc.)

13.5.2.32. **setBond**

Usage: *setBond* <mask1> <mask2> <k> <Req>

Changes (or adds a non-existent) bond in the topology file. Each mask must select the same number of atoms, and a bond will be placed between the atoms in mask1 and mask2 (one bond between atom1 from mask1 and atom1 from mask2 and another bond between atom2 from mask1 and atom2 from mask2, etc.)

13.5.2.33. **setMolecules**

Usage: *setMolecules* [*solute_ions*=True|False]

This command uses its own algorithm to determine system molecularity (which resets SOLVENT_POINTERS and ATOMS_PER_MOLECULE to what they *should* have been set to by *leap*). It will also determine if there are any errors in which molecules are not represented as consecutive atoms within a topology file (which won't happen unless you modify it yourself or there is a bug in tleap that prevents it from reordering atoms properly). However, in some unusual systems, tleap has been known to set the molecularity incorrectly, leading to strange segfaults and errors in sander and pmemd. Errors of this type can be caught with *checkValidity* and corrected using this command. It will also allow you to choose whether free ions are treated as part of the solute or part of the solvent.

13.5.2.34. **setOverwrite**

Usage: *setOverwrite* [True|False]

Allows the original topology file to be overwritten. By default, the original prmtop file is protected, and you cannot overwrite it. If you provide no value on this line, then it defaults to

13. Miscellaneous utilities

True. Note that no check is made if you are overwriting any other existing file (just the original topology).

13.5.2.35. strip

Usage: *strip* <mask>

This will strip every atom that corresponds to the given atom mask out of the topology file altogether. Any bond, angle, or dihedral that it is a part of will be deleted as well. The bond, angle, and dihedral types that are no longer referenced after the atoms are stripped out are deleted from the topology file. All Lennard Jones parameters are kept, however, even if they are no longer used. In this way, any LJ modifications you did before the strip command will remain intact. If all solvent residues and atoms are deleted, then the IFBOX pointer is set to 0 and the SOLVENT_POINTERS, ATOMS_PER_MOLECULE, and BOX_DIMENSIONS (unused section of the topology file) are deleted. NOTE that if you only remove a couple solvent molecules, any combineMolecules or setMolecules commands issued previously will be reset! You will have to run them again. Finally, pointer order could not be preserved for remaining atoms for efficiency considerations. For this reason, all pointers are recalculated before a new topology file is written out, so even stripping just a small ligand molecule will appear to change the topology file significantly if comparing via diff or a similar program. However, these differences are caused by a simple rearrangement of pointers and should yield correct energies.

13.5.2.36. writeFrcmod

Usage: *writeFrcmod* <frcmod_name>

This command will dump a complete frcmod file containing every parameter in your topology file. (Note that because LEaP cannot produce pair-specific VDW parameters, the effects of a changeLJPair will NOT be reflected in the topology file unless the pair you choose is between two atoms with the same VDW type). It assumes the canonical Amber combining rules for VDW terms, and uses each type's interaction with itself to extract the well depths and VDW radii.

13.5.2.37. writeOFF

Usage: *writeOFF* <OFF_File>

Writes an Amber OFF (library) file containing every residue, including terminal residues, found in a given topology file.

13.5.3. Examples

This section outlines a couple of example input files for *parmed.py* with comments describing what each command does. You can try these examples on the test parameter files in \$AMBERHOME/AmberTools/test/parmed (either the normal_prmtop/trx.prmtop or the chamber_prmtop/dhfr_gas.prmtop).

Example 1

```

# This file generates a topology file with the new mbondi3 radii
# optimized for the igb = 8 GB model and changes the charge set
# of LYS 3 (trx.prmtop) to set up for a FEP-like calculation.
# In practice you would need more than just the protonated and
# deprotonated state (you would have to interpolate), but this
# is just a demonstration.

# Change to mbondi3
changeRadii mbondi3

# Output the first topology file
outparm trx_mbondi3_state0.parm7

# Change the charges of the LYS
change charge :3@N -0.3479
change charge :3@H 0.2747
change charge :3@CA -0.24
change charge :3@HA 0.1426
change charge :3@CB -0.10961
change charge :3@HB2,HB3 0.034
change charge :3@CG 0.06612
change charge :3@HG2,HG3 0.01041
change charge :3@CD -0.03768
change charge :3@HD2,HD3 0.01155
change charge :3@CE 0.32604
change charge :3@HE2,HE3 -0.03358
change charge :3@NZ -1.03581
change charge :3@HZ1 0
change charge :3@HZ2,HZ3 0.38604
change charge :3@C 0.7341
change charge :3@O -0.5894

# Output the second topology file
outparm trx_mbondi3_state1.parm7

```

Example 2

```

# This file generates a topology file in which the L-J
# interactions between atoms 10 and 28 have been removed,
# and the L-J interactions between atoms 40, 41, 42, and
# 57 with everybody else has been removed.

# Make atoms 10 and 28 new LJ types, but keep their original

```

13. Miscellaneous utilities

```
# well depths and radii
addLJType @10
addLJType @28

# Zero the interaction between them
changeLJPair @10 @28 0.0 0.0

# Make atoms 40, 41, 42, and 57 a new LJ type with 0s for
# their parameters to remove all of their LJ interactions
# with every other atom
addLJType @40-42,57 0.0 0.0

# Write the final topology file. This statement could have
# been put anywhere
parmout altered_LJ.parm7
```

13.5.4. xparmed.py

To aid in simple tasks and make single- (or few-) prmtop file changes easier, a GUI version of ParmEd is available. It uses the Tk/Python graphical toolkit interface (called *Tkinter*). Tkinter is part of the standard Python library, but not all operating systems provide it with their system Python. The package names recognized by different package managers (e.g. *apt-get*, *port*, and *yum*) vary from system to system, and are detailed in the section below separated by common operating systems that have been tested by developers.

The GUI is very basic with a number of limitations. For instance, windows cannot be resized (but should fit on most standard terminals and should be sized appropriately). Furthermore, if an information window is present, the application will not end with the “Exit *xParmEd*” button until all information windows are closed. For scripting purposes, the text-based version, *parmed.py*, should be used instead.

13.5.4.1. Tkinter on Ubuntu (Debian)

To install Tkinter on Ubuntu (the package name on other Debians may differ), use the following command: *sudo apt-get install python-tk*

13.5.4.2. Tkinter on Red Hat

To install Tkinter on Red Hat (and CentOS and Fedora, probably), use the following command: *sudo yum install tkinter*

13.5.4.3. Tkinter on Mac OS X

The default Python installation on Mac OS X has Tkinter installed by default. In fact, it's a much 'prettier' version because it is built on top of Apple's GUI toolkits, which makes it look like a native Mac application. You can force Amber programs to use the Mac system Python by specifying */usr/bin/python* as the default python to configure. If you wish to use a

Python installed via MacPorts, you will need to also install the corresponding tkinter port. For instance, if you installed Python 2.7 from MacPorts and wish to use that, you will also need to install *py27-tkinter*.

13.5.4.4. Tkinter on Everything Else

If your system does not already have Tkinter installed, and none of the above helps you, you should consult a search engine or online forums. If it doesn't exist, you may have to stick with *parmed.py*.

13.5.5. Advanced Options

This section describes some of the advanced options in *parmed.py*. Note these are not generally available to *xparmed.py*

13.5.5.1. Interactive Python Shell

To increase ParmEd's flexibility, you can activate an limited, interactive Python interpreter to inject your own custom Python code into *parmed.py*'s normal execution. This brings with it the risk that custom code can be malicious if untrusted, so custom code evaluation is disallowed by default. To enable it, use the “-e” or “-enable-interpreter” command-line flag when executing *parmed.py*. To improve security, import statements are disallowed, although the math module has been imported for basic mathematical operations. To execute a single instruction, begin the command with a “!”. In this case, leading whitespace is eliminated (so leading tabs/spaces are ignored here). For example,

```
bash $ parmed.py -e -n trx.prmtop
Loaded Amber topology file trx.prmtop

Reading input from STDIN...
> !print amber_prmtop.parm_data['ATOM_NAME'][0:10]
['N', 'H1', 'H2', 'H3', 'CA', 'HA', 'CB', 'HB2', 'HB3', 'OG']
```

To execute a formatted block of code that requires more than one line, use “!!” to indicate to ParmEd that you wish to drop to interpreter mode. Terminate that block of code with another “!!” line. The prompt in STDIN-mode changes to “py >>>”. For example:

```
bash$ parmed.py -e -n trx.prmtop
Loaded Amber topology file trx.prmtop

Reading input from STDIN...
> !!
py >>> def formatted_print(items):
py >>>     i = 0
py >>>     for item in items:
py >>>         print '%10.4f ' % item,
py >>>         i += 1
```

13. Miscellaneous utilities

```
py >>>         if i % 5 == 0: print "
py >>>     print "
py >>>
py >>> formatted_print(amber_prmtop.parm_data['CHARGE'][0:10])
py >>> !!
      0.1849      0.1898      0.1898      0.1898      0.0567
      0.0782      0.2596      0.0273      0.0273     -0.6714

> quit
Quitting.
```

The main topology class instance being worked on is called `amber_prmtop`. See the API documentation below if you are interested in making custom modifications. Note that it is VERY easy to break a topology file with this approach, so consider this an advanced option. A description of the topology file format can be found on <http://ambermd.org/formats.html>.

WARNING: Variable declarations you make here drop onto the top-level namespace in ParmEd's normal operating environment. That is, any variable you declare here MIGHT override a critical one for ParmEd. Variable names to avoid using include any of the Python built-in functions and types as well as *line*, *code*, *debug*, *ParmedActions*, *ParmError*, *LineToCmd*, *amberParm*, *output_parm*, and *input*.

13.5.5.2. Python Amber Topology API documentation

`class amberParm`: The main topology file class. Its constructor takes a topology file name (required) and a restart file name (optional). Certain instance attributes are accessible only if a restart file is loaded (these are indicated below). It is accessible through the module `chemistry.amber.readparm`. Instantiate `amberParm` objects via commands like:

```
from chemistry.amber.readparm import AmberParm
my_topology = AmberParm('my_file.prmtop')
```

or

```
import chemistry
my_topology = chemistry.amber.readparm.AmberParm('my_file.prmtop', 'my_file.inpcrd')
```

Class methods:

`__init__(prmtop_name,[inpcrd_name])` Constructor. Sets up the instance variables, parses the topology file, and loads the coordinates for each atom if an `inpcrd` file name is given.

`__str__()` Returns the topology file name as the string representation of an `amberParm` class. Called via "typecasting" an `amberParm` to a str-type or invoking the `__str__` method directly. Use like: `str(my_topology)` –or– `my_topology.__str__()` –or– `'%s' % my_topology`

`LoadPointers()` Reloads the "pointers" instance attributes from the POINTERS section of the topology file data. You should use this if you make any changes to the data in the POINTERS section of the topology file. Use like: `my_topology.LoadPointers()`

ptr(pointer) Returns the value of the given pointer from the pointers dictionary (NOT from the topology file). It is case-insensitive. See <http://ambermd.org/formats.html> for a list of pointer names. Use like: `num_atoms = my_topology.ptr('natom')`

rdparm() Parses the topology file and stores all of the data in the arrays and dictionaries detailed below. This is called automatically in the constructor (`__init__`) method, so you should not call this method directly unless you really have to.

rdparm_old() Parses old-style topology files. This is called automatically inside `rdparm()` if it's determine

writeParm(name) Writes a new topology file with the given name (required) using all data present in the `parm_data` and `formats` dictionaries.

frcmmod(name) Writes an `frcmmod` file with a given name with every parameter present in the topology file. It supports variable 1-4 scaling factors present in the `prmtop`. It does NOT work for 10-12 `prmtops` or chamber topologies (there are no `frcmmods` for chamber `prmtops`).

writeOFF(name) Writes an OFF file to a given filename (defaults to "off.lib")

fill_LJ() Calculates the LJ radii and LJ depths for each atom type by analyzing each type's self-interaction (the `ACOEf` and `BCOEf` for each atom type interacting with another atom of the same type) by reversing the combining rules. This fills `LJ_radius`, `LJ_depth`, and `LJ_types` arrays/dictionary.

fill_14_LJ() Calculates the LJ radii and LJ depths for each atom type's 1-4 interactions (CHAMBER `prmtops` only!) the same way that it's done in `fill_LJ()` (but it fills the `LJ_14_radius` and `LJ_14_depth` arrays).

recalculate_LJ() Repopulates the `LENNARD_JONES_ACEOF` and `LENNARD_JONES_BCOEF` arrays by using the normal Amber combining rules on the well depths and radii found in `LJ_depth` and `LJ_radius`.

recalculate_14_LJ() Same as `recalculate_LJ()`, but it does it for CHAMBER `prmtops` for the 1-4 nonbonded parameters using the `LJ_14_radius` and `LJ_14_depth` arrays.

LoadRst7(filename) Loads a restart file and its coordinates and/or velocities. This is called automatically in the constructor if a restart filename is given.

addFlag(options)** Options are (`flag_name`, `flag_format`, `num_items` | `data`, `comments`). This will add a `%FLAG` to the topology file data dictionary, it will add the appropriate Fortran format statement (it must be a simple statement like `10I8`, `5E16.8`, etc.) to the `formats` dictionary, and it will either add an array of size `num_items` filled with 0s OR it will use the provided `data` array. If you do not give a data array (which MUST be an iterable, and it is converted to a Python list), then you have to give the number of 0s to put in a list under that `FLAG` name. It will also add any `prmtop` comments if you supply them.

13. Miscellaneous utilities

delete_mask(mask) This takes an AmberMask instance (but checks that the AmberMask's topology is the same as itself) or it takes a string mask and converts it to an AmberMask object, removing all atoms from atom_list. This should only be called **once** for each instance, as not all internal variables and settings are reset properly to enable a second delete_mask. The coord and vels arrays are updated to reflect only the coordinates and velocities of the remaining atoms. remake_parm() is called at the end of delete_mask.

remake_parm() This recalculates the topology parameters from the given atom_list and lists of bonds, angles, and dihedrals. So far, it only works with normal topology files (not chamber-created topology files, LES topology files, or Amoeba topology files). This only needs to be called if any of the above variables have been changed (and is called automatically by writeParm if it detects any of the arrays have been modified in any way).

Instance variables (or attributes). Note that Python dictionaries are like hash tables and Python lists index starting from 0:

parm_data Dictionary that pairs a prmtop %FLAG name with a Python list containing all of the data corresponding to that FLAG.

parm_comments Dictionary that pairs a prmtop %FLAG name with a Python list containing all of the comments associated with that FLAG.

formats Dictionary that pairs a prmtop %FLAG name with its Fortran format string specified in the topology file.

chamber Boolean value that indicates whether a topology file was written by CHAMBER or not (if it has a %FLAG CTITLE instead of TITLE)

version Version string found at the top of the prmtop file (str type)

prm_name Name of the original topology file (str type)

overwrite Boolean (True or False) that determines if we are allowed to overwrite prm_name in the writeParm method described above.

valid Boolean that indicates whether there were any problems parsing the topology file or any glaring issue with it (like it was lacking a POINTERS section)

exists Boolean that indicates whether or not the prmtop file exists.

LJ_types Dictionary that maps AMBER_ATOM_TYPE to the type index from the flag ATOM_TYPE_INDEX. Useful if you only have the Amber atom type and not the atom number (in which case, just use the ATOM_TYPE_INDEX list from parm_data)

LJ_radius Python list of ordered Lennard Jones radii corresponding to ATOM_TYPE_INDEX values.

LJ_depth Python list of ordered Lennard Jones well depths corresponding to ATOM_TYPE_INDEX values.

LJ_14_radius Same as LJ_radius above for 1-4 non-bonded parameters. ONLY present in CHAMBER prmtops!

LJ_14_depth Same as LJ_depth above for 1-4 non-bonded parameters. ONLY present in CHAMBER prmtops!

coords Python list with coordinates of each atom in the format [x1,y1,z1,x2,y2,z2, ..., xN,yN,zN]. Only exists if a restart file was loaded via the LoadRst7() above.

hasvels Boolean value that indicates whether velocities were loaded from the parsed restart file. Only present if a restart file was loaded.

vels If has_vels is True, this stores all of the velocities parsed from the restart file in a Python list. Only present if has_vels is True.

hasbox Boolean value that indicates whether box information was loaded from the parsed restart file. Only present if a restart file was loaded.

box Python list containing all box information found in the restart file. Only present if hasbox is True.

atom_list List of _Atom classes that describe each atom in the system. Each atom has instance variables bond_partners, angle_partners, dihedral_partners, xx, xy, xz (cartesian coordinates if a restart file is loaded), vx, vy, vz (velocities if a restart file is loaded), starting_index, and idx. The partners arrays are used to define which atoms that atom defines a bond, angle, or dihedral with (each atom appears only once and only in one of those arrays). These are used to define the exclusion list. starting_index is a pointer into all of the atomic data arrays (like ATOM_NAME, ATOM_TYPE_INDEX, etc.), and is updated every time remake_parm() is called. idx is never set until writeParm is called to write the topology file (and is reset to -1 after starting_index is updated at the end of the routine).

bonds_inc_h List of all bonds including hydrogen listed in the original topology file. This array is NOT modified by delete_mask. The only bonds from this list that are added to the prmtop in remake_parm are the ones whose atoms still exist in the atom_list array at the time remake_parm is called. Each bond has associated with it a bond type that is in the bond_type_list array described below.

bonds_without_h List of all bonds without hydrogen. See description for bonds_inc_h

bond_type_list List of all bond types defined in original topology file. The only ones assigned indexes are the ones found in bonds between remaining atoms defined in bonds_inc_h and bonds_without_h.

angles_inc_h, angles_without_h, angle_type_list Same as bond counterparts, but for angles

dihedrals_inc_h, dihedrals_without_h, dihedral_type_list Same as bond/angle counterparts, but for dihedrals

13. Miscellaneous utilities

angles A Python list of Python lists for each atom in the system. Each atom index (beginning from 0 to NATOM - 1) points to a Python list that contains every atom index (beginning from 0) in the system that that atom creates an angle with, but is NOT in the *bonds* list corresponding to that atom (above).

residue_container A Python list in which each atom's index (starting from 0) contains the residue number (CAREFUL: starting from 1) that that atom belongs to.

13.5.5.3. Extending ParmEd

This section describes what is necessary to add a new action to ParmEd. When testing additions, it is useful to use the `-d/-debug` flags, which will print detailed information (such as offending file lines, etc.) about syntax errors and other exceptions.

All actions are parsed from the `ParmEdActions.py` file in `$AMBERHOME/AmberTools/src/parmed/` directory. Each action must be its own class that inherits from `Action` and takes the `AmberParm` instance `amber_prmtop` as its first argument in its constructor. All arguments in the constructor *after* the topology file class must appear in the order that you want the user to place them in the command. See existing methods as examples. You also need to take care to write the class doc-strings (the string immediately following every class declaration) to be as helpful as possible, because they are used in the help function. You must also add your command's usage statement in the "usages" dictionary found at the top of `ParmEdActions.py`, or it will be invisible to the help function.

No further action is necessary to add your functionality to ParmEd (and you should never have to edit `parmed.py` directly – any class put in `ParmEdActions.py` is immediately accessible by `parmed.py`). Existing actions provide helpful examples if you choose to expand ParmEd.

Extending xParmEd: Any action that is added to `ParmEdActions.py` will be visible as buttons in `xparmed.py`, but will be disabled by default unless you implement that action directly. There is no well-defined standard for implementing actions in the GUI version like there is in the text-based version. GUI actions are defined in `$AMBERHOME/AmberTools/src/parmed/-ParmEdTools/gui/_guiactions.py`, and all additional actions must be defined there. You should only have to modify `_guiactions.py`, since the GUI is automatically sized and filled based on classes in `ParmEdActions.py`. The best advice I can give if you want to expand `xParmEd` is to copy the class that does a similar task and modify it for your class. The related examples are fairly consistent in their style of implementation, so hopefully it is easy enough to add actions quickly.

14. NAB: Introduction

Nucleic acid builder (nab) is a high-level language that facilitates manipulations of macromolecules and their fragments. nab uses a C-like syntax for variables, expressions and control structures (if, for, while) and has extensions for operating on molecules (new types and a large number of builtins for providing the necessary operations). We expect nab to be useful in model building and coordinate manipulation of proteins and nucleic acids, ranging in size from fairly small systems to the largest systems for which an atomic level of description makes good computational sense. As a programming language, it is not a solution or program in itself, but rather provides an environment that eases many of the bookkeeping tasks involved in writing programs that manipulate three-dimensional structural models.

The current implementation is version 6.0, and incorporates the following main features:

1. Objects such as points, atoms, residues, strands and molecules can be referenced and manipulated as named objects. The internal manipulations involved in operations like merging several strands into a single molecule are carried out automatically; in most cases the programmer need not be concerned about the internal data structures involved.
2. Rigid body transformations of molecules or parts of molecules can be specified with a fairly high-level set of routines. This functionality includes rotations and translations about particular axis systems, least-squares atomic superposition, and manipulations of coordinate frames that can be attached to particular atomic fragments.
3. Additional coordinate manipulation is achieved by a tight interface to distance geometry methods. This allows relationships that can be defined in terms of internal distance constraints to be realized in three-dimensional structural models. nab includes subroutines to manipulate distance bounds in a convenient fashion, in order to carry out tasks such as working with fragments within a molecule or establishing bounds based on model structures.
4. Force field calculations (*e.g.* molecular dynamics and minimization) can be carried out with an implementation of the AMBER force field. This works in both three and four dimensions, but periodic simulations are not (yet) supported. However, the generalized Born models implemented in Amber are also implemented here, which allows many interesting simulations to be carried out without requiring periodic boundary conditions. The force field can be used to carry out minimization, molecular dynamics, or normal mode calculations. Conformational searching and docking can be carried out using a "low-mode" (LMOD) procedure that performs sampling exploring the potential energy surface along low-frequency vibrational directions.
5. nab also implements a form of regular expressions that we call *atom regular expressions*, which provide a uniform and convenient method for working on parts of molecules.

14. NAB: Introduction

6. Many of the general programming features of the *awk* language have been incorporated in nab. These include regular expression pattern matching, *hashedarrays* (i.e., arrays with strings as indices), the splitting of strings into fields, and simplified string manipulations.
7. There are built-in procedures for linking nab routines to other routines written in C or Fortran, including access to most library routines normally available in system math libraries.

Our hope is that nab will serve to formalize the step-by-step process that is used to build complex model structures, and will facilitate the management and use of higher level symbolic constraints. Writing a program to create a structure forces more of the model's assumptions to be explicit in the program itself. And an nab description can serve as a way to show a model's salient features, much like helical parameters are used to characterize duplexes.

The first three chapters of this document both introduces the language through a series of sample programs, and illustrates the programming interfaces provided. The examples are chosen not only to show the syntax of the language, but also to illustrate potential approaches to the construction of some unusual nucleic acids, including DNA double- and triple-helices, RNA pseudoknots, four-arm junctions, and DNA-protein interactions. A separate reference manual (in Chapter 4) gives a more formal and careful description of the requirements of the language itself.

The basic literature reference for the code is T. Macke and D.A. Case. Modeling unusual nucleic acid structures. In *Molecular Modeling of Nucleic Acids*, N.B. Leontes and J. SantaLucia, Jr., eds. (Washington, DC: American Chemical Society, 1998), pp. 379-393. Users are requested to include this citation in papers that make use of NAB.

The authors thank Jarrod Smith, Garry Gippert, Paul Beroza, Walter Chazin, Doree Sitkoff and Vickie Tsui for advice and encouragement. Special thanks to Neill White (who helped in updating documentation, in preparing the distance geometry database, and in testing and porting portions of the code), and to Will Briggs (who wrote the fiber-diffraction routines). Thanks also to Chris Putnam and M.L. Dodson for bug reports.

14.1. Background

Using a computer language to model polynucleotides follows logically from the fundamental nature of nucleic acids, which can be described as “conflicted” or “contradictory” molecules. Each repeating unit contains seven rotatable bonds (creating a very flexible backbone), but also contains a rigid, planar base which can participate in a limited number of regular interactions, such as base pairing and stacking. The result of these opposing tendencies is a family of molecules that have the potential to adopt a virtually unlimited number of conformations, yet have very strong preferences for regular helical structures and for certain types of loops.

The controlled flexibility of nucleic acids makes them difficult to model. On one hand, the limited range of regular interactions for the bases permits the use of simplified and more abstract geometric representations. The most common of these is the replacement of each base by a plane, reducing the representation of a molecule to the set of transformations that relate the planes to each other. On the other hand, the flexible backbone makes it likely that there are entire families of nucleic acid structures that satisfy the constraints of any particular modeling

problem. Families of structures must be created and compared to the model's constraints. From this we can see that modeling nucleic acids involves not just chemical knowledge but also three processes—abstraction, iteration and testing—that are the basis of programming.

Molecular computation languages are not a new idea. Here we briefly describe some past approaches to nucleic acid modeling, to provide a context for nab.

14.1.1. Conformation build-up procedures

MC-SYM[200–202] is a high level molecular description language used to describe single stranded RNA molecules in terms of functional constraints. It then uses those constraints to generate structures that are consistent with that description. MC-SYM structures are created from a small library of conformers for each of the four nucleotides, along with transformation matrices for each base. Building up conformers from these starting blocks can quickly generate a very large tree of structures. The key to MC-SYM's success is its ability to prune this tree, and the user has considerable flexibility in designing this pruning process.

In a related approach, Erie *et al.*[203] used a Monte-Carlo build-up procedure based on sets of low energy dinucleotide conformers to construct longer low energy single stranded sequences that would be suitable for incorporation into larger structures. Sets of low energy dinucleotide conformers were created by selecting one value from each of the sterically allowed ranges for the six backbone torsion angles and χ . Instead of an exhaustive build-up search over a small set of conformers, this method samples a much larger region of conformational space by randomly combining members of a larger set of initial conformers. Unlike strict build-up procedures, any member of the initial set is allowed to follow any other member, even if their corresponding torsion angles do not exactly match, a concession to the extreme flexibility of the nucleic acid backbone. A key feature determined the probabilities of the initial conformers so that the probability of each created structure accurately reflected its energy.

Tung and Carter[204, 205] have used a reduced coordinate system in the NAMOT (nucleic acid modeling tool) program to rotation matrices that build up nucleic acids from simplified descriptions. Special procedures allow base-pairs to be preserved during deformations. This procedure allows simple algorithmic descriptions to be constructed for non-regular structures like intercalation sites, hairpins, pseudoknots and bent helices.

14.1.2. Base-first strategies

An alternative approach that works well for some problems is the "base-first" strategy, which lays out the bases in desired locations, and attempts to find conformations of the sugar-phosphate backbone to connect them. Rigid-body transformations often provide a good way to place the bases. One solution to the backbone problem would be to determine the relationship between the helicoidal parameters of the bases and the associated backbone/sugar torsions. Work along these lines suggests that the relationship is complicated and non-linear.[206] However, considerable simplification can be achieved if instead of using the complete relationship between all the helicoidal parameters and the entire backbone, the problem is limited to describing the relationship between the helicoidal parameters and the backbone/sugar torsion angles of single nucleotides and then using this information to drive a constraint minimizer that tries to connect adjacent nucleotides. This is the approach used in JUMNA,[207] which decomposes

14. NAB: Introduction

the problem of building a model nucleic acid structure into the constraint satisfaction problem of connecting adjacent flexible nucleotides. The sequence is decomposed into 3'-nucleotide monophosphates. Each nucleotide has as independent variables its six helicoidal parameters, its glycosidic torsion angle, three sugar angles, two sugar torsions and two backbone torsions. JUMNA seeks to adjust these independent variables to satisfy the constraints involving sugar ring and backbone closure.

Even constructing the base locations can be a non-trivial modeling task, especially for non-standard structures. Recognizing that coordinate frames should be chosen to provide a simple description of the transformations to be used, Gabarro-Arpa *et al.*[208] devised "Object Command Language" (OCL), a small computer language that is used to associate parts of molecules called objects, with arbitrary coordinate frames defined by sets of their atoms or numerical points. OCL can "link" objects, allowing other objects' positions and orientations to be described in the frame of some reference object. Information describing these frames and links is written out and used by the program MORCAD[209] which does the actual object transformations.

OCL contains several elements of a molecular modeling language. Users can create and operate on sets of atoms called objects. Objects are built by naming their component atoms and to simplify creation of larger objects, expressions, IF statements, an iterated FOR loop and limited I/O are provided. Another nice feature is the equivalence between a literal 3-D point and the position represented by an atom's name. OCL includes numerous built-in functions on 3-vectors like the dot and cross products as well as specialized molecular modeling functions like creating a vector that is normal to an object. However, OCL is limited because these language elements can only be assembled into functions that define coordinate frames for molecules that will be operated on by MORCAD. Functions producing values of other data types and stand-alone OCL programs are not possible.

14.2. Methods for structure creation

As a structure-generating tool, nab provides three methods for building models. They are rigid-body transformations, metric matrix distance geometry, and molecular mechanics. The first two methods are good initial methods, but almost always create structures with some distortion that must be removed. On the other hand, molecular mechanics is a poor initial method but very good at refinement. Thus the three methods work well together.

14.2.1. Rigid-body transformations

Rigid-body transformations create model structures by applying coordinate transformations to members of a set of standard residues to move them to new positions and orientations where they are incorporated into the growing model structure. The method is especially suited to helical nucleic acid molecules with their highly regular structures. It is less satisfactory for more irregular structures where internal rearrangement is required to remove bad covalent or non-bonded geometry, or where it may not be obvious how to place the bases.

nab uses the matrix type to hold a 4×4 transformation matrix. Transformations are applied to residues and molecules to move them into new orientations or positions. nab does *not* require

that transformations applied to parts of residues or molecules be chemically valid. It simply transforms the coordinates of the selected atoms leaving it to the user to correct (or ignore) any chemically incorrect geometry caused by the transformation.

Every nab molecule includes a frame, or “handle” that can be used to position two molecules in a generalization of superimposition. Traditionally, when a molecule is superimposed on a reference molecule, the user first forms a correspondence between a set of atoms in the first molecule and another set of atoms in the reference molecule. The superimposition algorithm then determines the transformation that will minimize the rmsd between corresponding atoms. Because superimposition is based on actual atom positions, it requires that the two molecules have a common substructure, and it can only place one molecule on top of another and not at an arbitrary point in space.

The nab frame is a way around these limitations. A frame is composed of three orthonormal vectors originally aligned along the axes of a right handed coordinate frame centered on the origin. nab provides two builtin functions `setframe()` and `setframep()` that are used to reposition this frame based on vectors defined by atom expressions or arbitrary 3-D points, respectively. To position two molecules via their frames, the user moves the frames so that when they are superimposed via the nab builtin `alignframe()`, the two molecules have the desired orientation. This is a generalization of the methods described above for OCL.

14.2.2. Distance geometry

nab's second initial structure-creation method is *metric matrix distance geometry*,^[210, 211] which can be a very powerful method of creating initial structures. It has two main strengths. First, since it uses internal coordinates, the initial position of atoms about which nothing is known may be left unspecified. This has the effect that distance geometry models use only the information the modeler considers valid. No assumptions are required concerning the positions of unspecified atoms. The second advantage is that much structural information is in the form of distances. These include constraints from NMR or fluorescence energy transfer experiments, implied propinquities from chemical probing and footprinting, and tertiary interactions inferred from sequence analysis. Distance geometry provides a way to formally incorporate this information, or other assumptions, into the model-building process.

Distance geometry converts a molecule represented as a set of interatomic distances into a 3-D structure. nab has several builtin functions that are used together to provide metric matrix distance geometry. A `bounds` object contains the molecule's interatomic distance bounds matrix and a list of its chiral centers and their volumes. The function `newbounds()` creates a `bounds` object containing a distance bounds matrix containing initial upper and lower bounds for every pair of atoms, and a list of the molecule's chiral centers and their volumes. Distance bounds for pairs of atoms involving only a single residue are derived from that residue's coordinates. The 1,2 and 1,3 distance bounds are set to the actual distance between the atoms. The 1,4 distance lower bound is set to the larger of the sum of the two atoms van der Waals radii or their *syn* (torsion angle = 0o) distance, and the upper bound is set to their *anti* (torsion angle = 180o) distance. `newbounds()` also initializes the list of the molecule's chiral centers. Each chiral center is an ordered list of four atoms and the volume of the tetrahedron those four atoms enclose. Each entry in a nab residue library contains a list of the chiral centers composed entirely of atoms in that residue.

14. NAB: Introduction

Once a bounds object has been initialized, the modeler can use functions to tighten, loosen or set other distance bounds and chiralities that correspond to experimental measurements or parts of the model's hypothesis. The functions `andbounds()` and `orbounds()` allow logical manipulation of bounds. `setbounds_from_db()` Allows distance information from a model structure or a database to be incorporated into a part of the current molecule's bounds object, facilitating transfer of information between partially-built structures.

These primitive functions can be incorporated into higher-level routines. For example the functions `stack()` and `watsoncrick()` set the bounds between the two specified bases to what they would be if they were stacked in a strand or base-paired in a standard Watson/Crick duplex, with ranges of allowed distances derived from an analysis of structures in the Nucleic Acid Database.

After all experimental and model constraints have been entered into the bounds object, the function `tsmooth()` applies "triangle smoothing" to pull in the large upper bounds, since the maximum distance between two atoms can not exceed the sum of the upper bounds of the shortest path between them. Random pairwise metrization[212] can also be used to help ensure consistency of the bounds and to improve the sampling of conformational space. The function `embed()` finally takes the smoothed bounds and converts them into a 3-D object. The newly embedded coordinates are subject to conjugate gradient refinement against the distance and chirality information contained in bounds. The call to `embed()` is usually placed in a loop to explore the diversity of the structures the bounds represent.

14.2.3. Molecular mechanics

The final structure creation method that nab offers is *molecular mechanics*. This includes both energy minimization and molecular dynamics - simulated annealing. Since this method requires a good estimate of the initial position of every atom in a structure, it is not suitable for creating initial structures. However, given a reasonable initial structure, it can be used to remove bad initial geometry and to explore the conformational space around the initial structure. This makes it a good method for refining structures created either by rigid body transformations or distance geometry. nab has its own 3-D/4-D molecular mechanics package that implements several AMBER force fields and reads AMBER parameter and topology files. Solvation effects can also be modelled with generalized Born continuum models.

Our hope is that nab will serve to formalize the step-by-step process that is used to build complex model structures. It will facilitate the management and use of higher level symbolic constraints. Writing a program to create a structure forces one to make explicit more of the model's assumptions in the program itself. And an nab description can serve as a way to exhibit a model's salient features, much like helical parameters are used to characterize duplexes. So far, nab has been used to construct models for synthetic Holliday junctions,[213] calycyclin dimers,[214] HMG-protein/DNA complexes,[215] active sites of Rieske iron-sulfur proteins,[216] and supercoiled DNA.[217] The Examples chapter below provides a number of other sample applications.

14.3. Compiling nab Programs

Compiling nab programs is very similar to compiling other high-level language programs, such as C and Fortran. The command line syntax is

```
nab [-O] [-c] [-v] [-noassert] [-nodebug] [-o file] [-Dstring] file(s)
```

where

```
-O optimizes the object code
-c suppresses the linking stage with ld and produces a .o file
-v verbosely reports on the compile process
-noassert causes the compiler to ignore assert statements
-nodebug causes the compiler to ignore debug statements
-o file names the output file
-Dstring defines string to the C preprocessor
```

Linking Fortran and C object code with nab is accomplished simply by including the source files on the command line with the nab file. For instance, if a nab program *bar.nab* uses a C function defined in the file *foo.c*, compiling and linking optimized nab code would be accomplished by

```
nab -O bar.nab foo.c
```

The result is an executable *a.out* file.

14.4. Parallel Execution

The generalized Born energy routines (for both first and second derivatives) include directives that will allow for parallel execution on machines that support this option. Once you have some level of comfort and experience with the single-CPU version, you can enable parallel execution by supplying one of several parallelization options (*-openmp*, *-mpi* or *-scalapack*) to configure, by re-building the NAB compiler and by recompiling your NAB program.

The *-openmp* option enables parallel execution under OpenMP on shared-memory machines. To enable OpenMP execution, add the *-openmp* option to configure, re-build the NAB compiler and re-compile your NAB program. Then, if you set the *OMP_NUM_THREADS* environment variable to the number of threads that you wish to perform parallel execution, the Born energy computation will execute in parallel.

The *-mpi* option enables parallel execution under MPI on either clusters or shared-memory machines. To enable MPI execution, add the *-mpi* option to configure and re-build the NAB compiler. You will not need to modify your NAB programs; just execute them with an *mpirun* command.

The *-scalapack* option enables parallel execution under MPI on either clusters or shared-memory machines, and in addition uses the Scalable LAPACK (ScaLAPACK) library for parallel linear algebra computation that is required to calculate the second derivatives of the generalized Born energy, to perform Newton-Raphson minimization or to perform normal mode analysis. For computations that do not involve linear algebra (such as conjugate gradients minimization or molecular dynamics) the *-scalapack* option functions in the same manner as the

-mpi option. Do not use the *-mpi* and *-scalapack* options simultaneously. Use the *-scalapack* option only when ScaLAPACK has been installed on your cluster or shared-memory machine.

In order that the *-mpi* or *-scalapack* options result in a correct build of the NAB compiler, the configure script must specify linking of the MPI library, or ScaLAPACK and BLACS libraries, as part of that build. These libraries are specified for Sun machines in the `solaris_cc` section of the configure script. If you want to use MPI or ScaLAPACK on a machine other than a Sun machine, you will need to modify the configure script to link these libraries in a manner analogous to what occurs in the `solaris_cc` section of the script.

There are three options to specify the manner in which NAB supports linear algebra computation. The *-scalapack* option discussed above specifies ScaLAPACK. The *-perflib* option specifies Sun TM Performance Library TM, a multi-threaded implementation of LAPACK. If neither *-scalapack* nor *-perflib* is specified, then linear algebra computation will be performed by a single CPU using LAPACK. In this last case, the Intel MKL library will be used if the `MKL_HOME` environment variable is set at configure time. Absent that, if a `GOTO` environment variable is found, the GotoBLAS libraries will be used.

The parallel execution capability of NAB was developed primarily on Sun machines, and has also been tested on the SGI Altix platform. But it has been much less widely-used than have other parts of NAB, so you should certainly run some tests with your system to ensure that single-CPU and parallel runs give the same results.

The `$AMBERHOME/benchmarks/nab` directory has a series of timing benchmarks that can be helpful in assessing performance. See the README file there for more information.

14.5. First Examples

This section introduces `nab` via three simple examples. All `nab` programs in this user manual are set in Courier, a typewriter style font. The line numbers at the beginning of each line are not parts of the programs but have been added to make it easier to refer to specific program sections.

14.5.1. B-form DNA duplex

One of the goals of `nab` was that simple models should require simple programs. Here is an `nab` program that creates a model of a B-form DNA duplex and saves it as a PDB file.

```
1 // Program 1 - Average B-form DNA duplex
2 molecule m;
3
4 m = bdna( "gcgttaacgc" );
5 putpdb( "gcg10.pdb", m );
```

Line 2 is a declaration used to tell the `nab` compiler that the name `m` is a molecule variable, something `nab` programs use to hold structures. Line 4 creates the actual model using the predefined function `bdna()`. This function's argument is a literal string which represents the sequence of the duplex that is to be created. Here's how `bdna()` converts this string into a molecule. Each letter stands for one of the four standard bases: `a` for adenine, `c` for cytosine, `g`

for guanine and t for thymine. In a standard DNA duplex every adenine is paired with thymine and every cytosine with guanine in an antiparallel double helix. Thus only one strand of the double helix has to be specified. As `bdna()` reads the string from left to right, it creates one strand from 5' to 3' (5'-gcgttaacgc -3'), automatically creating the other antiparallel strand using Watson/Crick pairing. It uses a uniform helical step of 3.38 Å rise and 36.0o twist. Naturally, nab has other ways to create helical molecules with arbitrary helical parameters and even mismatched base pairs, but if you need some “average” DNA, you should be able to get it without having to specify every detail. The last line uses the nab builtin `putpdb()` to write the newly created duplex to the file `gcg10.pdb`.

Program 1 is about the smallest nab program that does any real work. Even so, it contains several elements common to almost all nab programs. The two consecutive forward slashes in line 1 introduce a comment which tells the nab compiler to ignore all characters between them and the end of the line. This particular comment begins in column 1, but that is not required as comments may begin in any column. Line 3 is blank. It serves no purpose other than to visually separate the declaration part from the action part. nab input is free format. Runs of white space characters—spaces, tabs, blank lines and page breaks—act like a single space which is required only to separate reserved words like `molecule` from identifiers like `m`. Thus white space can be used to increase readability.

14.5.2. Superimpose two molecules

Here is another simple nab program. It reads two DNA molecules and superimposes them using a rotation matrix made from a correspondence between their C1' atoms.

```

1  // Program 2 - Superimpose two DNA duplexes
2  molecule m, mr;
3  float r;
4
5  m = getpdb( "test.pdb" );
6  mr = getpdb( "gcg10.pdb" );
7  superimpose( m, "::C1'", mr, "::C1'" );
8  putpdb( "test.sup.pdb", m );
9  rmsd( m, "::C1'", mr, "::C1'", r );
10 printf( "rmsd = %8.3fn", r );

```

This program uses three variables—two molecules, `m` and `mr` and one float, `r`. An nab declaration can include any number of variables of the same type, but variables of different types must be in separate declarations. The builtin function `getpdb()` reads two molecules in PDB format from the files `test.pdb` and `gcg10.pdb` into the variables `m` and `mr`. The superimposition is done with the builtin function `superimpose()`. The arguments to `superimpose()` are two molecules and two “atom expressions”. nab uses atom expressions as a compact way of specifying sets of atoms. Atom expressions and atom names are discussed in more detail below but for now an atom expression is a pattern that selects one or more of the atoms in a molecule. In this example, they select all atoms with names C1'.

`superimpose()` uses the two atom expressions to associate the corresponding C1' carbons in the two molecules. It uses these correspondences to create a rotation matrix that when applied

to *m* will minimize the root mean square deviation between the pairs. It applies this matrix to *m*, “moving” it on to *mr*. The transformed molecule *m* is written out to the file *test.sup.pdb* in PDB format using the builtin function *putpdb()*. Finally the builtin function *rmsd()* is used to compute the actual root mean square deviation between corresponding atoms in the two superimposed molecules. It returns the result in *r*, which is written out using the C-like I/O function *printf()*. *rmsd()* also uses two atom expressions to select the corresponding pairs. In this example, they are the same pairs that were used in the superimposition, but any set of pairs would have been acceptable. An example of how this might be used would be to use different subsets of corresponding atoms to compute trial superimpositions and then use *rmsd()* over all atoms of both molecules to determine which subset did the best job.

14.5.3. Place residues in a standard orientation

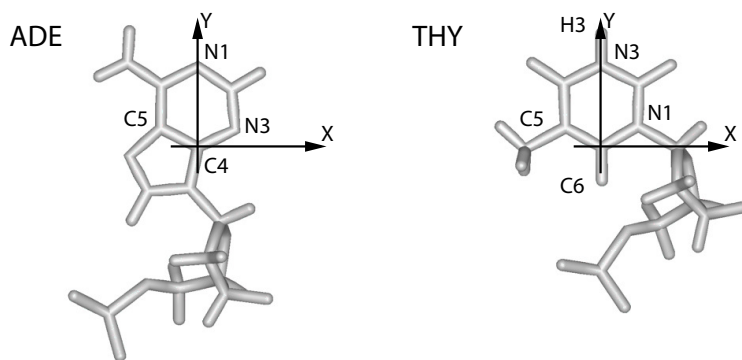
This is the last of the introductory examples. It places nucleic acid monomers in an orientation that is useful for building Watson/Crick base pairs. It uses several atom expressions to create a frame or handle attached to an *nab* molecule that permits easy movement along important “molecular directions”. In a standard Watson/Crick base pair the C4 and N1 atoms of the purine base and the H3, N3 and C6 atoms of the pyrimidine base are colinear. Such a line is obviously an important molecular direction and would make a good coordinate axis. Program 3 aligns these monomers so that this hydrogen bond is along the Y-axis.

```

1 // Program 3 - orient nucleic acid monomers
2 molecule m;
3
4 m = getpdb( "ADE.pdb" );
5 setframe( 2, m, // also for GUA
6           ">::C4",
7           ">::C5", ">::N3",
8           ">::C4", ">::N1" );
9 alignframe( m, NULL );
10 lputpdb( "ADE.std.pdb", m );
11
12 m = getpdb( "THY.pdb" );
13 setframe( 2, m, // also for CYT & URA
14           ">::C6",
15           ">::C5", ">::N1",
16           ">::C6", ">::N3" );
17 alignframe( m, NULL );
18 putpdb( "THY.std.pdb", m );

```

This program uses only one variable, the molecule *m*. Execution begins on line 4 where the builtin *getpdb()* is used to read in the coordinates of an adenine (created elsewhere) from the file *ADE.pdb*. The *nab* builtin *setframe()* creates a coordinate frame for this molecule using vectors defined by some of its atoms as shown in Figure 14.1. The first atom expression (line 6) sets the origin of this coordinate frame to be the coordinates of the C4 atom. The two atom expressions on line 7 set the X direction from the coordinates of the C5 to the coordinates of the N3. The last two atom expressions set the Y direction from the C4 to the N1. The Z-axis is created by

Figure 14.1.: *ADE and THY after execution of Program 3.*

the cross product $X \times Y$. Frames are thus like sets of local coordinates that can be attached to molecules and used to facilitate defining transformations; a more complete discussion is given in the section **Frames** below.

`nab` requires that the coordinate axes of all frames be orthogonal, and while the X and Y axes as specified here are close, they are not quite exact. `setframe()` uses its first parameter to specify which of the original two axes is to be used as a formal axis. If this parameter is 1, then the specified X axis becomes the formal X axis and Y is recreated from $Z \times X$; if the value is 2, then the specified Y axis becomes the formal Y axis and X is recreated from $Y \times Z$. In this example the specified Y axis is used and X is recreated. The builtin `alignframe()` transforms the molecule so that the X, Y and Z axes of the newly created coordinate frame point along the standard X, Y and Z directions and that the origin is at (0,0,0). The transformed molecule is written to the file `ADE.std.pdb`. A similar procedure is performed on a thymine residue with the result that the hydrogen bond between the H3 of thymine and the N1 of adenine in a Watson Crick pair is now along the Y axis of these two residues.

14.6. Molecules, Residues and Atoms

We now turn to a discussion of ways of describing and manipulating molecules. In addition to the general-purpose variable types like `float`, `int` and `string`, `nab` has three types for working with molecules: `molecule`, `residue` and `atom`. Like their chemical counterparts, `nab` molecules are composed of residues which are in turn composed of atoms. The residues in an `nab` molecule are organized into one or more named, ordered lists called strands. Residues in a strand are usually bonded so that the “exiting” atom of residue i is connected to the “entering” atom of residue $i + 1$. The residues in a strand need not be bonded; however, only residues in the same strand can be bonded.

Each of the three molecular types has a complex internal structure, only some of which is directly accessible at the `nab` level. Simple elements of these types, like the number of atoms in a molecule or the X coordinate of an atom are accessed via attributes—a suffix attached to a

molecule, residue or atom variable. Attributes behave almost like int, float and string variables; the only exception being that some attributes are read only with values that can't be changed. More complex operations on these types such as adding a residue to a molecule or merging two strands into one are handled with builtin functions. A complete list of nab builtin functions and molecule attributes can be found in the nab Language Reference.

14.7. Creating Molecules

The following functions are used to create molecules. Only an overview is given here; more details are in chapter 3.

```
molecule newmolecule();
int addstrand( molecule m, string str );
residue getresidue( string rname, string rlib );
residue transformres( matrix mat, residue res, string aex );
int addresidue( molecule m, string str, residue res );
int connectres( molecule m, string str,
               int rn1, string atm1, int rn2, string atm2 );
int mergestr( molecule m1, string str1, string endl,
             molecule m2, string str2, string end2 );
```

The general strategy for creating molecules with nab is to create a new (empty) molecule then build it one residue at a time. Each residue is fetched from a residue library, transformed to properly position it and added to a growing strand. A template showing this strategy is shown below. *mat*, *m* and *res* are respectively a matrix, molecule and residue variable declared elsewhere. Words in *italics* indicate general instances of things that would be filled in according to actual application.

```
1  ...
2  m = newmolecule();
3  addstrand( m, \fIstr-1\fC );
4  ...
5  for( ... ){
6  ...
7  res = getresidue( \fIres-name\fC, \fIres-lib\fC );
8  res = transformres( mat, res, NULL );
9  addresidue( m, \fIstr-name\fC, res );
10 ...
11 }
12 ...
```

In line 2, the function `newmolecule()` creates a molecule and stores it in *m*. The new molecule is empty—no strands, residues or atoms. Next `addstrand()` is used to add a strand named *str-1*. Strand names may be up to 255 characters in length and can include any characters except white space. Each strand in a molecule must have a unique name. There is no limit on the number of strands a molecule may have.

The actual structure would be created in the loop on lines 5-11. Each time around the loop, the function `getresidue()` is used to extract the next residue with the name *res-name* from some

residue library *res-lib* and stores it in the residue variable *res*. Next the function *transformres()* applies a transformation matrix, held in the matrix variable *mat* to the residue in *res*, which places it in the orientation and position it will have in the new molecule. Finally, the function *addresidue()* appends the transformed residue to the end of the chain of residues in the strand *str-name* of the new molecule.

Residues in each strand are numbered from 1 to N , where N is the number of residues in that strand. The residue order is the order in which they were inserted with *addresidue()*. While *nab* does not require it, nucleic acid chains are usually numbered from 5' to 3' and proteins chains from the N-terminus to the C-terminus. The residues in nucleic acid strands and protein chains are usually bonded with the outgoing end of residue i bonded to the incoming end of residue $i+1$. However, as this is not always the case, *nab* requires the user to explicitly make all interresidue bonds with the builtin *connectres()*.

connectres() makes bonds between two atoms in different residues of the same strand of a molecule. Only residues in the same strand can be bonded. *connectres()* takes six arguments. They are a molecule, the name of the strand containing the residues to be bonded, and two pairs each of a residue number and the name of an atom in that residue. As an example, this call to *connectres()*,

```
connectres( m, "sense", i, "O3'", i+1, "P" );
```

connects an atom named "O3'" in residue i to an atom named "P" in residue $i+1$, creating the phosphate bond that joins two nucleic acid monomers.

The function *mergestr()* is used to either move or copy the residues in one strand into another strand. Details are provided in chapter 3.

14.8. Residues and Residue Libraries

nab programs build molecules from residues that are parts of residue libraries, which are exactly those distributed with the Amber molecular mechanics programs (see <http://amber.scripps.edu>).

nab provides several functions for working with residues. All return a valid residue on success and NULL on failure. The function *getres()* is written in *nab* and its source is shown below. *transformres()* which applies a coordinate transformation to a residue and is discussed under the section **Matrices and Transformations**.

```
residue getresidue( string resname, string reslib );
residue getres( string resname, string reslib );
residue transformres( matrix mat, residue res, string aexp );
```

getresidue() extracts the residue with name *resname* from the residue library *reslib*. *reslib* is the name of a file that either contains the residue information or contains names of other files that contain it. *reslib* is assumed to be in the directory $\$NABHOME/reslib$ unless it begins with a slash (/)

A common task of many *nab* programs is the translation of a string of characters into a structure where each letter in the string represents a residue. Generally, some mapping of one or two character names into actual residue names is required. *nab* supplies the function *getres()*

14. NAB: Introduction

that maps the single character names a, c, g, t and u and their 5' and 3' terminal analogues into the residues ADE, CYT, GUA, THY and URA. Here is its source:

```
1 // getres() - map 1 letter names into 3 letter names
2 residue getres( string rname, string rlib )
3 {
4     residue res;
5     string maplto3[ hashed ];          // convert residue names
6
7     maplto3["A"] = "ADE";      maplto3["C"] = "CYT";
8     maplto3["G"] = "GUA";      maplto3["T"] = "THY";
9     maplto3["U"] = "URA";
10
11     maplto3["a"] = "ADE";      maplto3["c"] = "CYT";
12     maplto3["g"] = "GUA";      maplto3["t"] = "THY";
13     maplto3["u"] = "URA";
14
15     if( r in maplto3 ) {
16         res = getresidue( maplto3[ r ], rlib );
17     }else{
18         fprintf( stderr, "undefined residue %s\\n", r );
19         exit( 1 );
20     }
21     return( res );
22 };
```

getres() is the first of several nab functions that are discussed in this User Manual. The following explanation will cover not just getres() but will serve as an introduction to user defined nab functions in general.

An nab function is a named group of declarations and statements that is executed as a unit by using the function's name in an expression. nab functions can have special variables called parameters that allow the same function to operate on different data. A function definition begins with a header that describes the function, followed by the function body which is a list of statements and declarations enclosed in braces ({}) and ends with a semicolon. The header to getres() is on line 2 and the body is on lines 3 to 22.

Every nab function header begins with the reserved word that specifies its type, followed by the function's name followed by its parameters (if any) enclosed in parentheses. The parentheses are always required, even if the function does not have parameters. nab functions may return a single value of any of the 10 nab types. nab functions can not return arrays. In symbolic terms every nab function header uses this template:

type name(parameters?)

The parameters (if present) to an nab function are a comma separated list of type variable pairs:

type1 variable1, type2 variable2, ...

An nab function may have any number of parameters, including none. Parameters may of any of the 10 nab types, but unlike function values, parameters can be arrays, including *hashed* arrays. The function `getres()` has two parameters, the two string variables `resname` and `reslib`.

Parameters to nab functions are “called by reference” which means that they contain the actual data—not copies of it—that the function was called with. When an nab function parameter is assigned, the actual data in the calling function is changed. The only exception is when an expression is passed as a parameter to an nab function. In this case, the nab compiler evaluates the expression into a temporary (and invisible to the nab programmer) variable and then operates on its contents.

Immediately following the function header is the function body. It is a list of declarations followed by a list of statements enclosed in braces. The list of declarations, the list of statements or both may be empty. `getres()` has several statements, and a single declaration, the variable `res`. This variable is a *local variables*. Local variables are defined only when the function is active. If a local variable has the same name as variable defined outside of a it the local variable hides the global one. Local variables can not be parameters.

The statement part of `getres()` begins on line 6. It consists of several if statements organized into a decision tree. The action of this tree is to translate one of the strings A, , , T, etc., or their lower case equivalents into the corresponding three letter standard nucleic acid residue name and then extract that residue from `reslib` using the low level residue library function `getresidue()`. The value returned by `getresidue()` is stored in the local variable `res`, except when the input string is not one of those listed above. In that case, `getres()` writes a message to `stderr` indicating that it can not translate the input string and sets `res` to the value `NULL`. nab uses `NULL` to represent non-existent values of the types string, file, atom, residue, molecule and bounds. A value of `NULL` generally means that a variable is uninitialized or that an error occurred in creating it.

A function returns a value by executing a return statement, which is the reserved word `return` followed by an expression. The return statement evaluates the expression, sets the function value to it and returns control to the point just after the call. The expression is optional but if present the type of the expression must be the same as the type of the function or both must be numeric (int, float). If the expression is missing, the function still returns, but its value is undefined. `getres()` includes one return statements on line 20. A function also returns with an undefined value when it “runs off the bottom”, i.e., executes the last statement before the closing brace and that statement is not a return.

14.9. Atom Names and Atom Expressions

Every atom in an nab molecule has a name. This name is composed of the strand name, the residue *number* and the atom name. As both PDB and off formats require that all atoms in a residue have distinct names, the combination of strand name, residue number and atom name is unique for each atom in a single molecule. Atoms in different molecules, however, may have the same name.

Many nab builtins require the user to specify exactly which atoms are to be covered by the operation. nab does this with special strings called *atom expressions*. An atom expression is a pattern that matches one or more atom names in the specified molecule or residue. An atom

14. NAB: Introduction

expression consists of three parts—a strand part, a residue part and an atom part. The parts are separated by colons (:). Not all three parts are required. An atom expression with no colons consists of only a strand part; it selects *all* atoms in the selected strands. An atom expression with one colon consists of a strand part and a residue part; it selects *all* atoms in the selected residues in the selected strands. An empty part selects all strands, residues or atoms depending on which parts are empty.

nab patterns specify the *entire* string to be matched. For example, the atom pattern C matches only atoms named C, and not those named CA, HC, etc. To match any name that begins with C, use C*, to match any name ending with C, use *C and to match a C in any position use *C*. An atom expression is first parsed into its parts. The strand part is evaluated selecting one or more strands in a molecule. Next the residue part is evaluated. Only residues in selected strands can be selected. Finally the atom part is evaluated and only atoms in selected residues are selected. Here are some typical atom expressions and the atoms they match.

:ADE:	Select all atoms in any residue named ADE. All three parts are present but both the strand and atom parts are empty. The atom expression :ADE selects the same set of atoms.
::C,CA,N	select all atoms with names C, CA or N in all residues in all strands—typically the peptide backbone.
A:1-10,13,URA:C1'	Select atoms named C1' (the glycosyl-carbons) in residues 1 to 10 and 13 and in any residues named URA in the strand named A.
::C*[^']	Select all non-sugar carbons. The [^'] is an example of a negated character class. It matches any character in the last position except '.
::P,O?P,C[3-5]?,O[35]?	The nucleic acid backbone. This P selects phosphorous atoms. The O?P matches phosphate oxygens that have various second letters O1P, O2P or OAP or OBP. The C[3-5]? matches the backbone carbons, C3', C4', C5' or C3*, C4*, C5*. And the O[35]? matches the backbone oxygens O3', O5' or O3*, O5*.
:: or :	Select all atoms in the molecule.

An important property of nab atom expressions is that the order in which the strands, residues, and atoms are listed is unimportant. That is, the atom expression "2,1:5,2,3:N1,C1'" is the exact same atom expression as "1,2:3,2,5:C1',N1". All atom expressions are reordered, internal to nab, in increasing atom number. So, in the above example, the selected atoms will be selected in the following sequence:

```
1:2:N1, 1:2:C1', 1:3:N1, 1:3:C1', 1:5:N1, 1:5:C1', 2:2:N1, 2:2:C1',
2:3:N1, 2:3:C1', 2:5:N1, 2:5:C1'
```

The order in which atoms are selected internal to a specific residue are the order in which they appear in a nab PDB file. As seen in the above example, N1 appears before C1' in all nab nucleic acid residues and PDB files.

14.10. Looping over atoms in molecules

Another thing that many nab programs have to do is visit every atom of a molecule. nab provides a special form of its for-loop for accomplishing this task. These loops have this form:

```
for( a in m ) stmt;
```

a and *m* represent an atom and a molecule variable. The action of the loop is to set *a* to each atom in *m* in this order. The first atom is the first atom of the first residue of the first strand. This is followed by the rest of the atoms of this residue, followed by the atoms of the second residue, etc until all the atoms in the first strand have been visited. The process is then repeated on the second and subsequent strands in *m* until *a* has been set to every atom in *m*. The order of the strands in a molecule is the order in which they were created with `addstrand()`, the order of the residues in a strand is the order in which they were added with `addresidue()` and the order of the atoms in a residue is the order in which they are listed in the residue library entry that the residue is based on.

The following program uses two nested for-in loops to compute all the proton-proton distances in a molecule. Distances less than cutoff are written to stdout. The program uses the second argument on the command to hold the cutoff value. The program also uses the `==` operator to compare a character string, in this case an atom name to pattern, specified as a regular expression.

```
1 // Program 4 - compute H-H distances <= cutoff
2 molecule    m;
3 atom        ai, aj;
4 float       d, cutoff;
5
6 cutoff = atof( argv[ 2 ] );
7 m = getpdb( "gcg10.pdb" );
8
9 for( ai in m ){
10     if( ai.atomname !~ "H" )continue;
11     for( aj in m ){
12         if( aj.tatomnum <= ai.tatomnum )continue;
13         if( aj.atomname !~ "H" )continue;
14         if( ( d=distp(ai.pos,aj.pos)) <=cutoff){
15             printf(
16                 "%3d %-4s %-4s %3d %-4s %-4s %8.3f\\n",
17                 ai.tresnum, ai.resname, ai.atomname,
18                 aj.tresnum, aj.resname, aj.atomname,
19                 d );
20         }
21     }
22 }
```

The molecule is read into *m* using `getpdb()`. Two atom variables *ai* and *aj* are used to hold the pairs of atoms. The outer loop in lines 9-22 sets *ai* to each atom in *m* in the order discussed

above. Since this program is only interested in proton-proton distances, if *ai* is not a proton, all calculations involving that atom can be skipped. The if in line 10 tests to see if *ai* is a proton. It does so by testing to see if *ai*'s name, available via the *atomname* attribute doesn't match the regular expression "H". If it doesn't match then the program executes the *continue* statement also on line 10, which has the effect of advancing the outer loop to its next atom.

>From the section on attributes, *ai.atomname* behaves like a character string. It can be compared against other character strings or tested to see if it matches a pattern or regular expression. The two operators, *=~* and *!~* stand for *match* and *doesn't-match*. They also inform the nab compiler that the string on their right hand sides is to be treated like a regular expression. In this case, the regular expression "H" matches any name that contains the letter H, or any proton which is just what is required.

If *ai* is a proton, then the inner loop from 11-21 is executed. This sets *aj* to each atom in the same order as the loop in 9. Since distance is reflexive (*dist i, j = dist j, i*), and the distance between an atom and itself is 0, the inner loop uses the if on line 12 to skip the calculation on *aj* unless it follows *ai* in the molecule's atom order. Next the if on line 13 checks to see if *aj* is a proton, skipping to the next atom if it is not. Finally, the if on line 14 computes the distance between the two protons *ai* and *aj* and if it is *<= cutoff* writes the information out using the C-like I/O function *printf()*.

14.11. Points, Transformations and Frames

nab provides three kinds of geometric objects. They are the types *point* and *matrix* and the *frame* component of a molecule.

14.11.1. Points and Vectors

The nab type *point* is an object that holds three float values. These values can represent the X, Y and Z coordinates of a point or the components of 3-vector. The individual elements of a point variable are accessed via attributes or suffixes added to the variable name. The three point attributes are "x", "y" and "z". Many nab builtin functions use, return or create point values. Details of operations on points are given in chapter 3.

14.11.2. Matrices and Transformations

nab uses the *matrix* type to hold a 4×4 transformation matrix. Transformations are applied to residues and molecules to move them into new orientations and/or positions. Unlike a general coordinate transformation, nab transformations can not alter the scale (size) of an object. However, transformations can be applied to a subset of the atoms of a residue or molecule changing its shape. For example, nab would use a transformation to rotate a group of atoms about a bond. nab does *not* require that transformations applied to parts of residues or molecules be chemically valid. It simply transforms the coordinates of the selected atoms leaving it to the user to correct (or ignore) any chemically incorrect geometry caused by the transformation. nab uses the following builtin functions to create and use transformations.

```
matrix newtransform( float dx, float dy, float dz,
```

```

float rx, float ry, float rz );
matrix rot4( molecule m, string tail, string head, float angle );
matrix rot4p( point tail, point head, float angle );
matrix trans4( molecule m, string tail, string head, float distance );
matrix trans4p( point tail, point head, float distance );
residue transformres( matrix mat, residue r, string aex );
int transformmol( matrix mat, molecule m, string aex );

```

nab provides three ways to create a new transformation matrix. The function `newtransform()` creates a transformation matrix from 3 translations and 3 rotations. It is intended to position objects with respect to the standard X, Y, and Z axes located at (0,0,0). Here is how it works. Imagine two coordinate systems, X, Y, Z and X', Y', Z' that are initially superimposed. `newtransform()` first rotates the primed coordinate system about Z by rz degrees, then about Y by ry degrees, then about X by rx degrees. Finally the reoriented primed coordinate system is translated to the point (dx,dy,dz) in the unprimed system. The functions `rot4()` and `rot4p()` create a transformation matrix that effects a clockwise rotation by an angle (in degrees) about an axis defined by two points. The points can be specified implicitly by atom expressions applied to a molecule in `rot4()` or explicitly as points in `rot4p()`. If an atom expression in `rot4()` selects more than one atom, the average coordinate of all selected atoms is used as the point's value. (Note that a positive rotation angle here is defined to be clockwise, which is in accord with the IUPAC rules for defining torsional angles in molecules, but is opposite to the convention found in many other branches of mathematics.) Similarly, the functions `trans4()` and `trans4p()` create a transformation that effects a translation by a distance along the axis defined by two points. A positive translation is from tail to head.

`transformres()` applies a transformation to those atoms of `res` that match the atom expression `aex`. It returns a *copy* of the input residue with the changed coordinates. The input residue is unchanged. It returns NULL if the new residue could not be created. `transformmol()` applies a transformation to those atoms of `mol` that match `aex`. Unlike `transformres()`, `transformmol()` *changes* the coordinates of the input molecule. It returns the number of atoms selected by `aex`. In both functions, the special atom expression NULL selects all atoms in the input residue or molecule.

14.11.3. Frames

Every nab molecule includes a frame, a handle that allows arbitrary and precise movement of the molecule. This frame is set with the nab builtins `setframe()` and `setframep()`. It is initially set to the standard X, Y and Z directions centered at (0,0,0). `setframe()` creates a coordinate frame from atom expressions that specify the the origin, the X direction and the Y direction. If any atom expression selects more than one atom, the average of the selected atoms' coordinates is used. Z is created from $X \times Y$. Since the initial X and Y directions are unlikely to be orthogonal, the `use` parameter specifies which of the input X and Y directions is to become the formal X or Y direction. If `use` is 1, X is chosen and Y is recreated from $Z \times X$. If `use` is 2, then Y is chosen and X is recreated from $Y \times Z$. `setframep()` is identical except that the five points defining the frame are explicitly provided.

```

int setframe( int use, molecule mol, string origin,

```

14. NAB: Introduction

```
        string xtail, string xhead,  
        string ytail, string yhead );  
int setframe( int use, molecule mol, point origin,  
             point xtail, point xhead,  
             point ytail, point yhead );  
int alignframe( molecule mol, molecule mref );
```

alignframe() is similar to superimpose(), but works on the molecules' frames rather than selected sets of their atoms. It transforms mol to superimpose its *frame* on the *frame* of mref. If mref is NULL, alignframe() superimposes the frame of mol on the standard X, Y and Z coordinate system centered at (0,0,0).

Here's how frames and transformations work together to permit precise motion between two molecules. Corresponding frames are defined for two molecules. These frames are based on molecular directions. alignframe() is first used to align the frame of one molecule along with the standard X, Y and Z directions. The molecule is then moved and reoriented via transformations. Because its initial frame was along these molecular directions, the transformations are likely to be along or about the axes. Finally alignframe() is used to realign the transformed molecule on the frame of the fixed molecule.

One use of this method would be the rough placement of a drug into a groove on a DNA molecule to create a starting structure for restrained molecular dynamics. setframe() is used to define a frame for the DNA along the appropriate groove, with its origin at the center of the binding site. A similar frame is defined for the drug. alignframe() first aligns the drug on the standard coordinate system whose axes are now important directions between the DNA and the drug. The drug is transformed and alignframe() realigns the transformed drug on the DNA's frame.

14.12. Creating Watson Crick duplexes

Watson/Crick duplexes are fundamental components of almost all nucleic acid structures and nab provides several functions for use in creating them. They are

```
residue getres( string resname, string reslib );  
molecule bdna( string seq );  
molecule fd_helix( string helix_type, string seq, string acid_type );  
string wc_complement( string seq, string reslib, string natype );  
molecule wc_basepair( residue sres, residue ares );  
molecule wc_helix( string seq, string rlib, string natype,  
                   string aseq, string arlib, string anatype, float xoff,  
                   float incl, float twist, float rise, string opts );
```

All of these functions are written in nab allowing the user to modify or extend them as needed without having to modify the nab compiler.

Note: If you just want to create a regular helical structure with a given sequence, use the "fiber-diffraction" routine fd_helix(), which is discussed in Section 15.14. The methods discussed next are more general, and can be extended to more complicated problems, but they are also much harder to follow and understand. The construction of "unusual" nucleic acids was the

original focus of NAB; if you are using NAB for some other purpose (such as running Amber force field calculations) you should probably skip to Chapter 18 at this point.

14.12.1. `bdna()` and `fd_helix()`

The function `bdna()` which was used in the first example converts a string into a Watson/Crick DNA duplex using average DNA helical parameters.

```

1 // bdna() - create average B-form duplex
2 molecule bdna( string seq )
3 {
4     molecule m;
5     string cseq;
6     cseq = wc_complement( seq, "", "dna" );
7     m = wc_helix( seq, "", "dna",
8                   cseq, "", "dna",
9                   2.25, -4.96, 36.0, 3.38, "s5a5s3a3" );
10    return( m );
11 };

```

`bdna()` calls `wc_helix()` to create the molecule. However, `wc_helix()` requires both strands of the duplex so `bdna()` calls `wc_complement()` to create a string that represents the Watson/Crick complement of the sequence contained in its parameter `seq`. The string `"s5a5s3a3"` replaces both the *sense* and *anti* 5' terminal phosphates with hydrogens and adds hydrogens to both the *sense* and *anti* 3' terminal O3' oxygens. The finished molecule in `m` is returned as the function's value. If any errors had occurred in creating `m`, it would have the value `NULL`, indicating that `bdna()` failed.

Note that the simple method used in `bdna()` for constructing the helix is not very generic, since it assumes that the *internal* geometry of the residues in the (default) library are appropriate for this sort of helix. This is in fact the case for B-DNA, but this method cannot be trivially generalized to other forms of helices. One could create initial models of other helical forms in the way described above, and fix up the internal geometry by subsequent energy minimization. An alternative is to directly use fiber-diffraction models for other types of helices. The `fd_helix()` routine does this, reading a database of experimental coordinates from fiber diffraction data, and constructing a helix of the appropriate form, with the helix axis along *z*. More details are given in Section 15.14.

14.12.2. `wc_complement()`

The function `wc_complement()` takes three strings. The first is a sequence using the standard one letter code, the second is the name of an nab residue library, and the third is the nucleic acid type (RNA or DNA). It returns a string that contains the Watson/Crick complement of the input sequence in the same one letter code. The input string and the returned complement string have opposite directions. If the left end of the input string is the 5' base then the left end of the returned string will be the 3' base. The actual direction of the two strings depends on their use.

```

1 // wc_complement() - create a string that is the W/C

```

```

2 // complement of the string seq
3 string wc_complement( string seq, string rlib, string rlt )
4 // (note that rlib is unused: included only for backwards compatibility
5 {
6     string acbase, base, wcbase, wcseq;
7     int i, len;
8
9     if( rlt == "dna" )         acbase = "t";
10    else if( rlt == "rna" ) acbase = "u";
11    else{
12        fprintf( stderr,
13            "wc_complement: rlt (%s) is not dna/rna, no W/C comp.", rlt );
14        return( NULL );
15    }
16    len = length( seq );
17    wcseq = NULL;
18    for( i = 1; i <= len; i = i + 1 ){
19        base = substr( seq, i, 1 );
20        if( base == "a" || base == "A" )         wcbase = acbase;
21        else if( base == "c" || base == "C" )     wcbase = "g";
22        else if( base == "g" || base == "G" )     wcbase = "c";
23        else if( base == "t" || base == "T" )     wcbase = "a";
24        else if( base == "u" || base == "U" )     wcbase = "a";
25        else{
26            fprintf( stderr, "wc_complement: unknown base %sn", base );
27            return( NULL );
28        }
29        wcseq = wcseq + wcbase;
30    }
31    return( wcseq );
32 }

```

`wc_complement()` begins its work in line 9, where the nucleic acid type, as indicated by `rlt` as DNA or RNA is used to determine the correct complement for an a. The complementary sequence is created in the for loop that begins in line 18 and extends to line 30. The nab builtin `substr()` is used to extract single characters from the input sequence beginning with with position 1 and working from left to right until entire input sequence has been converted. The if-tree from lines 20 to 28 is used to set the character complementary to the current character, using the previously determined `acbase` if the input character is an a or A. Any character other than the expected a, c, g, t, u (or A, C, G, T, U) is an error causing `wc_complement()` to print an error message and return `NULL`, indicating that it failed. Line 29 shows how nab uses the infix `+` to concatenate character strings. When the entire string has been complemented, the for loop terminates and the complementary sequence now in `wcseq` is returned as the function value. Note that if the input sequence is empty, `wc_complement()` returns `NULL`, indicating failure.

14.12.3. `wc_helix()` Overview

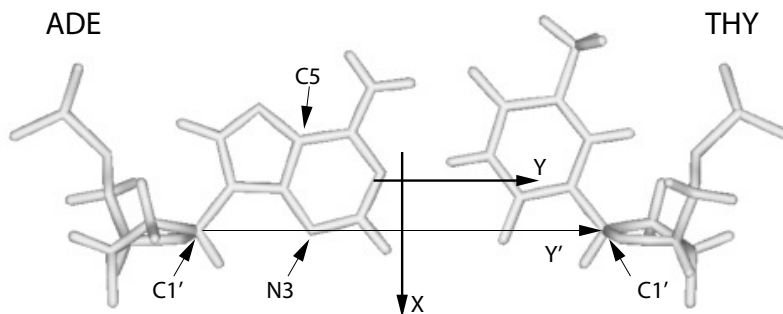
`wc_helix()` generates a uniform helical duplex from a sequence, its complement, two residue libraries and four helical parameters: `x-offset`, `inclination`, `twist` and `rise`. By using two residue libraries, `wc_helix()` can generate RNA/DNA heteroduplexes. `wc_helix()` returns an `nab` molecule containing two strands. The string `seq` becomes the "sense" strand and the string `aseq` becomes the "anti" strand. `seq` and `aseq` are required to be complementary although this is not checked. `wc_helix()` creates the molecule one base pair at a time. `seq` is read from left to right, `aseq` is read from right to left and corresponding letters are extracted and converted to residues by `getres()`. These residues are in turn combined into an idealized Watson/Crick base pair by `wc_basepair()`. An AT created by `wc_basepair()` is shown in Figure 2.

A Watson/Crick duplex can be modeled as a set of planes stacked in a helix. The numbers that describe the relationships between the planes and between the planes and the helical axis are called helical parameters. Planes can be defined for each base or base pair. Six numbers (three displacements and three angles) can be defined for every pair of planes; however, helical parameters for nucleic acid bases are restricted to the six numbers describing the relationship between the two bases in a base pair and the six numbers describing the relationship between adjacent base pairs. A complete description of helical parameters can be found in Dickerson.[218]

`wc_helix()` uses only four of the 12 helical parameters. It builds its helices from idealized Watson/Crick pairs. These pairs are planar so the three intra base angles are 0. In addition the displacements are displacements from the idealized Watson/Crick geometry and are also 0. The A and the T in Figure 2 are in plane of the page. `wc_helix()` uses four of the six parameters that relate a base pair to the helical axis. The helices created by `wc_helix()` have a single axis (the Z axis, not shown) which is at the intersection of the X and Y axes of Figure 2. Now imagine keeping the axes fixed in the plane of the paper and moving the base pair. X-offset is the displacement along the X axis between the Y axis and the line marked Y'. A positive X-offset is toward the arrow on the X-axis. Inclination is the rotation of the base pair about the X axis. A rotation that moves the A above the plane of page and the T below is positive. Twist involves a rotation of the base pair about the Z-axis. A counterclockwise twist is positive. Finally, rise is a displacement along the Z-axis. A positive rise is out of the page toward the reader.

14.12.4. `wc_basepair()`

The function `wc_basepair()` takes two residues and assembles them into a two stranded `nab` molecule containing one base pair. Residue `sres` is placed in the "sense" strand and residue `ares` is placed in the "anti" strand. The work begins in line 14 where `newmolecule()` is used to create an empty molecule stored in `m`. Two strands, `sense` and `anti` are added using `addstrand()`. In addition, two more molecules are created, `m_sense` for the sense residue and `m_anti` for the anti residue. The if-trees in lines 26-61 and 63-83 are used to select residue dependent atoms that will be used to move the base pairs into a convenient orientation for helix generation. The *purine*:C4 and *pyrimidine*:C6 distance which is residue dependent is also set. In line 62, `addresidue()` adds `sres` to the strand `sense` of `m_sense`. In line 84, `addresidue()` adds `ares` to the strand `anti` of `m_anti`. Lines 86 and 87 align the molecules containing the sense residue and anti

Figure 14.2.: *ADE:THY* from *wc_basepair()*.

residue so that sres and ares are on top of each other. Line 88 creates a transformation matrix that rotates *m_anti* (containing *ares*) 180o about the X-axis. After applying this transformation, the two bases are still occupying the same space but *ares* is now antiparallel to *sres*. Line 90 creates a transformation matrix that displaces *m_anti* and *ares* along the Y-axis by *sep*. The properly positioned molecules containing *sres* and *ares* are merged into a single molecule, *m*, completing the base pair. Lines 97-98 move this base pair to a more convenient orientation for helix generation. Initially the base as shown in Figure 14.2 is in the plane of page with origin on the C4 of the A. The calls to *setframe()* and *alignframe()* move the base pair so that the origin is at the intersection of the lines marked X and Y'.

```

1 // wc_basepair() - create Watson/Crick base pair
2 #define AT_SEP 8.29
3 #define CG_SEP 8.27
4
5 molecule wc_basepair( residue sres, residue ares )
6 {
7     molecule m, m_sense, m_anti;
8     float sep;
9     string srname, arname;
10    string xtail, xhead;
11    string ytail, yhead;
12    matrix mat;
13
14    m = newmolecule();
15    m_sense = newmolecule();
16    m_anti = newmolecule();
17    addstrand( m, "sense" );
18    addstrand( m, "anti" );
19    addstrand( m_sense, "sense" );
20    addstrand( m_anti, "anti" );

```



```

21
22 srname = getresname( sres );
23 arname = getresname( ares );
24 ytail = "sense::C1'";
25 yhead = "anti::C1'";
26 if( ( srname == "ADE" ) || ( srname == "DA" ) ||
27     ( srname == "RA" ) || ( srname =~ "[DR]A[35]" ) ){
28     sep = AT_SEP;
29     xtail = "sense::C5";
30     xhead = "sense::N3";
31     setframe( 2, m_sense,
32         "::-C4", "::-C5", "::-N3", "::-C4", "::-N1" );
33 }else if( ( srname == "CYT" ) || ( srname =~ "[DR]C[35]*" ) ){
34     sep = CG_SEP;
35     xtail = "sense::C6";
36     xhead = "sense::N1";
37     setframe( 2, m_sense,
38         "::-C6", "::-C5", "::-N1", "::-C6", "::-N3" );
39 }else if( ( srname == "GUA" ) || ( srname =~ "[DR]G[35]*" ) ){
40     sep = CG_SEP;
41     xtail = "sense::C5";
42     xhead = "sense::N3";
43     setframe( 2, m_sense,
44         "::-C4", "::-C5", "::-N3", "::-C4", "::-N1" );
45 }else if( ( srname == "THY" ) || ( srname =~ "DT[35]*" ) ){
46     sep = AT_SEP;
47     xtail = "sense::C6";
48     xhead = "sense::N1";
49     setframe( 2, m_sense,
50         "::-C6", "::-C5", "::-N1", "::-C6", "::-N3" );
51 }else if( ( srname == "URA" ) || ( srname =~ "RU[35]*" ) ){
52     sep = AT_SEP;
53     xtail = "sense::C6";
54     xhead = "sense::N1";
55     setframe( 2, m_sense,
56         "::-C6", "::-C5", "::-N1", "::-C6", "::-N3" );
57 }else{
58     fprintf( stderr,
59         "wc_basepair : unknown sres %s\\n",srname );
60     exit( 1 );
61 }
62 addresidue( m_sense, "sense", sres );
63 if( ( arname == "ADE" ) || ( arname == "DA" ) ||
64     ( arname == "RA" ) || ( arname =~ "[DR]A[35]" ) ){
65     setframe( 2, m_anti,
66         "::-C4", "::-C5", "::-N3", "::-C4", "::-N1" );
67 }else if( ( arname == "CYT" ) || ( arname =~ "[DR]C[35]*" ) ){
68     setframe( 2, m_anti,
69         "::-C6", "::-C5", "::-N1", "::-C6", "::-N3" );

```

```

70     }else if( ( arname == "GUA" ) || ( arname =~ "[DR]G[35]*" ) ){
71         setframe( 2, m_anti,
72             "::-C4", "::-C5", "::-N3", "::-C4", "::-N1" );
73     }else if( ( arname == "THY" ) || ( arname =~ "DT[35]*" ) ){
74         setframe( 2, m_anti,
75             "::-C6", "::-C5", "::-N1", "::-C6", "::-N3" );
76     }else if( ( arname == "URA" ) || ( arname =~ "RU[35]*" ) ){
77         setframe( 2, m_anti,
78             "::-C6", "::-C5", "::-N1", "::-C6", "::-N3" );
79     }else{
80         fprintf( stderr,
81             "wc_basepair : unknown ares %s\\n", arname );
82         exit( 1 );
83     }
84     addresidue( m_anti, "anti", ares );
85
86     alignframe( m_sense, NULL );
87     alignframe( m_anti, NULL );
88     mat = newtransform( 0., 0., 0., 180., 0., 0. );
89     transformmol( mat, m_anti, NULL );
90     mat = newtransform( 0., sep, 0., 0., 0., 0. );
91     transformmol( mat, m_anti, NULL );
92     mergestr( m, "sense", "last", m_sense, "sense", "first" );
93     mergestr( m, "anti", "last", m_anti, "anti", "first" );
94
95     freemolecule( m_sense ); freemolecule( m_anti );
96
97     setframe( 2, m, "::-C1'", xtail, xhead, ytail, yhead );
98     alignframe( m, NULL );
99     return( m );
100 };

```

14.12.5. wc_helix() Implementation

The function `wc_helix()` assembles base pairs from `wc_basepair()` into a helical duplex. It is a fairly complicated function that uses several transformations and shows how `mergestr()` is used to combine smaller molecules into a larger one. In addition to creating complete duplexes, `wc_helix()` can also create molecules that contain only one strand of a duplex. Using the special value `NULL` for either `seq` or `aseq` creates a duplex that omits the residues for the `NULL` sequence. The molecule still contains two strands, `sense` and `anti`, but the strand corresponding to the `NULL` sequence has zero residues. `wc_helix()` first determines which strands are required, then creates the first base pair, then creates the subsequent base pairs and assembles them into a helix and finally packages the requested strands into the returned molecule.

Lines 20-34 test the input sequences to see which strands are required. The variables `has_s` and `has_a` are flags where a value of 1 indicates that `seq` and/or `aseq` was requested. If an input sequence is `NULL`, `wc_complement()` is used to create it and the appropriate flag is set to 0. The `nab` builtin `setreslibkind()` is used to set the nucleic acid type so that the proper residue (DNA

or RNA) is extracted from the residue library.

The first base pair is created in lines 42-63. The two letters corresponding the 5' base of seq and the 3' base of aseq are extracted using the nab builtin substr(), converted to residues using getresidue() and assembled into a base pair by wc_basepair(). This base pair is oriented as in Figure 2 with the origin at the intersection of the lines X and Y'. Two transformations are created, xomat for the x-offset and inmat for the inclination and applied to this pair.

Base pairs 2 to slen-1 are created in the for loop in lines 66-87. substr() is used to extract the appropriate letters from seq and aseq which are converted into another base pair by getresidue() and wc_basepair(). Four transformations are applied to these base pairs - two to set the x-offset and the inclination and two more to set the twist and the rise. Next m2, the molecule containing the newly created properly positioned base pair must be bonded to the previously created molecule in m1. Since nab only permits bonds between residues in the same strand, mergestr() must be used to combine the corresponding strands in the two molecules before connectres() can create the bonds.

Because the two strands in a Watson/Crick duplex are antiparallel, adding a base pair to one end requires that one residue be added *after* the *last* residue of one strand and that the other residue added *before* the *first* residue of the other strand. In wc_helix() the sense strand is extended after its last residue and the anti strand is extended before its first residue. The call to mergestr() in line 79 extends the sense strand of m1 with the the residue of the sense strand of m2. The residue of m2 is added after the "last" residue of of the sense strand of m1. The final argument "first" indicates that the residue of m2 are copied in their original order m1:sense:last is followed by m2:sense:first. After the strands have been merged, connectres() makes a bond between the O3' of the next to last residue (i-1) and the P of the last residue (i). The next call to mergestr() works similarly for the residues in the anti strands. The residue in the anti strand of m2 are copied into the the anti strand of m1 *before* the first residue of the anti strand of m1 m2:anti:last precedes m1:anti:first . After merging connectres() creates a bond between the O3' of the new first residue and the P of the second residue.

Lines 121-130 create the returned molecule m3. If the flag has_s is 1, mergestr() copies the entire sense strand of m1 into the empty sense strand of m3. If the flag has_a is 1, the anti strand is also copied.

```

1 // wc_helix() - create Watson/Crick duplex
2 string wc_complement();
3 molecule wc_basepair();
4 molecule wc_helix(
5     string seq, string sreslib, string snatype,
6     string aseq, string areslib, string anatype,
7     float xoff, float incl, float twist, float rise,
8     string opts )
9 {
10     molecule m1, m2, m3;
11     matrix xomat, inmat, mat;
12     string arname, srname;
13     string sreslib_use, areslib_use;
14     string loup[ hashed ];
15     residue sres, ares;
16     int      has_s, has_a;

```

14. NAB: Introduction

```
17 int i, slen;
18 float   ttwist, trise;
19
20 has_s = 1; has_a = 1;
21 if( sreslib == "" ) sreslib_use = "all_nucleic94.lib";
22     else sreslib_use = sreslib;
23 if( areslib == "" ) areslib_use = "all_nucleic94.lib";
24     else areslib_use = areslib;
25
26 if( seq == NULL && aseq == NULL ){
27     fprintf( stderr, "wc_helix: no sequence\\n" );
28     return( NULL );
29 }else if( seq == NULL ){
30     seq = wc_complement( aseq, areslib_use, snatype );
31     has_s = 0;
32 }else if( aseq == NULL ){
33     aseq = wc_complement( seq, sreslib_use, anatype );
34     has_a = 0;
35 }
36
37 slen = length( seq );
38 loup["g"] = "G"; loup["a"] = "A";
39 loup["t"] = "T"; loup["c"] = "C";
40
41 //             handle the first base pair:
42 setreslibkind( sreslib_use, snatype );
43 srname = "D" + loup[ substr( seq, 1, 1 ) ];
44 if( opts =~ "s5" )
45     sres = getresidue( srname + "5", sreslib_use );
46 else if( opts =~ "s3" && slen == 1 )
47     sres = getresidue( srname + "3", sreslib_use );
48 else sres = getresidue( srname, sreslib_use );
49
50 setreslibkind( areslib_use, anatype );
51 arname = "D" + loup[ substr( aseq, 1, 1 ) ];
52 if( opts =~ "a3" )
53     ares = getresidue( arname + "3", areslib_use );
54 else if( opts =~ "a5" && slen == 1 )
55     ares = getresidue( arname + "5", areslib_use );
56 else ares = getresidue( arname, areslib_use );
57 m1 = wc_basepair( sres, ares );
58 freeresidue( sres ); freeresidue( ares );
59 xomat = newtransform(xoff, 0., 0., 0., 0., 0. );
60 transformmol( xomat, m1, NULL );
61 inmat = newtransform( 0., 0., 0., incl, 0., 0.);
62 transformmol( inmat, m1, NULL );
63
64 //             add in the main portion of the helix:
65 trise = rise; ttwist = twist;
```

```

66 for( i = 2; i <= slen-1; i = i + 1 ){
67     sname = "D" + loup[ substr( seq, i, 1 ) ];
68     setreslibkind( sreslib, sdatatype );
69     sres = getresidue( sname, sreslib_use );
70     aname = "D" + loup[ substr( aseq, i, 1 ) ];
71     setreslibkind( areslib, anatype );
72     ares = getresidue( aname, areslib_use );
73     m2 = wc_basepair( sres, ares );
74     freeresidue( sres ); freeresidue( ares );
75     transformmol( xomat, m2, NULL );
76     transformmol( inmat, m2, NULL );
77     mat = newtransform( 0., 0., trise, 0., 0., ttwist );
78     transformmol( mat, m2, NULL );
79     mergestr( m1, "sense", "last", m2, "sense", "first" );
80     connectres( m1, "sense", i-1, "O3'", i, "P" );
81     mergestr( m1, "anti", "first", m2, "anti", "last" );
82     connectres( m1, "anti", 1, "O3'", 2, "P" );
83     trise = trise + rise;
84     ttwist = ttwist + twist;
85     freemolecule( m2 );
86 }
87
88
89 i = slen;          // add in final residue pair:
90
91 if( i > 1 ){
92     sname = substr( seq, i, 1 );
93     sname = "D" + loup[ substr( seq, i, 1 ) ];
94     setreslibkind( sreslib, sdatatype );
95     if( opts =~ "s3" )
96         sres = getres( sname + "3", sreslib_use );
97     else
98         sres = getres( sname, sreslib_use );
99     aname = "D" + loup[ substr( aseq, i, 1 ) ];
100    setreslibkind( areslib, anatype );
101    if( opts =~ "a5" )
102        ares = getres( aname + "5", areslib_use );
103    else
104        ares = getres( aname, areslib_use );
105
106    m2 = wc_basepair( sres, ares );
107    freeresidue( sres ); freeresidue( ares );
108    transformmol( xomat, m2, NULL );
109    transformmol( inmat, m2, NULL );
110    mat = newtransform( 0., 0., trise, 0., 0., ttwist );
111    transformmol( mat, m2, NULL );
112    mergestr( m1, "sense", "last", m2, "sense", "first" );
113    connectres( m1, "sense", i-1, "O3'", i, "P" );
114    mergestr( m1, "anti", "first", m2, "anti", "last" );

```

```

115     connectres( m1, "anti", 1, "O3'", 2, "P" );
116     trise = trise + rise;
117     ttwist = ttwist + twist;
118     freemolecule( m2 );
119 }
120
121 m3 = newmolecule();
122 addstrand( m3, "sense" );
123 addstrand( m3, "anti" );
124 if( has_s )
125     mergestr( m3, "sense", "last", m1, "sense", "first" );
126 if( has_a )
127     mergestr( m3, "anti", "last", m1, "anti", "first" );
128 freemolecule( m1 );
129
130 return( m3 );
131 };

```

14.13. Structure Quality and Energetics

Up to this point, all the structures in the examples have been built using only transformations. These transformations properly place the purine and pyrimidine rings. However, since they are rigid body transformations, they will create distorted sugar/backbone geometry if any internal sugar/backbone rearrangements are required to accommodate the base geometry. The amount of this distortion depends on both the input residues and transformations applied and can vary from trivial to so severe that the created structures are useless. *nab* offers two methods for fixing bad sugar/backbone geometry. They are molecular mechanics and distance geometry. *nab* provides distance geometry routines and has its own molecular mechanics package. The latter is based on the *LEaP* program, which is part of the *AMBER* suite of programs developed at the University of California, San Francisco and at The Scripps Research Institute.

14.13.1. Creating a Parallel DNA Triplex

Parallel DNA triplexes are thought to be intermediates in homologous DNA recombination. These triplexes, investigated by Zhurkin *et al.*[219] are called R-form DNA, and are believed to exist in two distinct conformations. In the presence of recombination proteins (eg. RecA), they adopt an extended conformation that is underwound with respect to standard helices (a twist of 20°) and very large base stacking distances (a rise of 5.1 Å). However, in the absence of recombination proteins, R-form DNA exists in a "collapsed" form that resembles conventional triplexes but with two very important differences—the two parallel strands have the same sequence and the triplex can be made from any Watson/Crick duplex regardless of its base composition. The remainder of this section discusses how this triplex could be modeled and two *nab* programs that implement that strategy.

If the degrees of freedom of a triplex are specified by the helicoidal parameters required to place the bases, then a triplex of N bases has $6(N - 1)$ degrees of freedom, an impossibly

large number for any but trivial N . Fortunately, the nature of homologous recombination allows some simplifying assumptions. Since the recombination must work on *any* duplex, the overall shape of the triplex must be sequence independent. This implies that each helical step uses the same set of transformational parameters which reduces the size of the problem to six degrees of freedom once the individual base triads have been created.

The individual triads are created by assuming that they are planar, that the third base is hydrogen bonded on the major groove side of the base pair as it appears in a standard Watson/Crick duplex, that the original Watson Crick base pair pair is essentially undisturbed by the insertion of the third base and finally that the third base belongs at the point that maximizes its hydrogen bonding with respect to the original Watson/Crick base pair. After the optimized triads have been created, they are assembled into dimers. The dimers assume that the helical axis passes through the center of the circle defined by the positions of the three C1' atoms. Several instances of a two parameter family (rise, twist) of dimers are created for each of the 16 pairs of triads and minimized.

14.13.2. Creating Base Triads

Here is an nab program that computes the vacuum energy of XY:X base triads as a function of the position and orientation of the X (non-Watson/Crick) base. A minimum energy AU:A found by the program along with the potential energy surface keyed to the position of the second A is shown in Figure 3. The program creates a single Watson/Crick DNA base pair and then computes the energy of a third DNA base at each position of a user defined rectangular grid. Since hydrogen bonding is both distance and orientation dependent the program allows the user to specify a range of orientations to try at each grid point. The orientation giving the lowest energy at each grid point and its associated energy are written to a file. The position and orientation giving the lowest overall energy is saved and is used to *recreate* the best triad after the search is completed.

```

1 // Program 5 - Investigate energies of base triads
2 molecule m;
3 residue tr;
4 string sb, ab, tb;
5 matrix rmat, tmat;
6
7 file ef;
8 string mfnm, efnm;
9 point txyz[ 35 ];
10 float x, lx, hx, xi, mx;
11 float y, ly, hy, yi, my;
12 float rz, lrz, hrz, rzi, urz, mrz, brz;
13
14 int prm;
15 point xyz[ 100 ], force[ 100 ];
16 float me, be, energy;
17
18 scanf( "%s %s %s", sb, ab, tb );
19 scanf( "%lf %lf %lf", lx, hx, xi );

```

```

20 scanf( "%lf %lf %lf", ly, hy, yi );
21 scanf( "%lf %lf %lf", lrz, hrz, rzi );
22
23 mfnm = sprintf( "%s%s%s.triad.min.pdb", sb, ab, tb );
24 efnm = sprintf( "%s%s%s.energy.dat", sb, ab, tb );
25
26 m = wc_helix(sb, "", "dna", ab,
27             "", "dna", 2.25, 0.0, 0.0, 0.0 );
28
29 addstrand( m, "third" );
30 tr = getres( tb, "all_nucleic94.lib" );
31 addresidue( m, "third", tr );
32 setxyz_from_mol( m, "third:", txyz );
33
34 putpdb( m, "temp.pdb" ); m = getpdb_prm( "temp.pdb", "learpc.ff99SB", "", 0 );
35 mme_init( m, NULL, "::ZZZ", xyz, NULL );
36
37 ef = fopen( efnm, "w" );
38
39 mrz = urz = lrz - 1;
40 for( x = lx; x <= hx; x = x + xi ){
41     for( y = ly; y <= hy; y = y + yi ){
42         brz = urz;
43         for( rz = lrz; rz <= hrz; rz = rz + rzi ){
44             setmol_from_xyz( m, "third:", txyz );
45             rmat=newtransform( 0., 0., 0., 0., 0., rz );
46             transformmol( rmat, m, "third:" );
47             tmat=newtransform( x, y, 0., 0., 0., 0. );
48             transformmol( tmat, m, "third:" );
49
50             setxyz_from_mol( m, NULL, xyz );
51             energy = mme( xyz, force, 1 );
52
53             if( brz == urz ){
54                 brz = rz; be = energy;
55             }else if( energy < be ){
56                 brz = rz; be = energy;
57             }
58             if( mrz == urz ){
59                 me = energy;
60                 mx = x; my = y; mrz = rz;
61             }else if( energy < me ){
62                 me = energy;
63                 mx = x; my = y; mrz = rz;
64             }
65         }
66     }
67     fprintf( ef, "%10.3f %10.3f %10.3f %10.3fn",
68             x, y, brz, be );
69 }

```



```

69 }
70 fclose( ef );
71
72 setmol_from_xyz( m, "third::", txyz );
73 rmat = newtransform( 0.0, 0.0, 0.0, 0.0, 0.0, mrz );
74 transformmol( rmat, m, "third::" );
75 tmat = newtransform( mx, my, 0.0, 0.0, 0.0, 0.0 );
76 transformmol( tmat, m, "third::" );
77 putpdb( mfnm, m );

```

Program 5 begins by reading in a description of the desired triad and data defining the location and granularity of the search area. It does this with the calls to the `nab` builtin `scanf()` on lines 18-21. `scanf()` uses its first argument as a format string which directs the conversion of text versions of int, float and string values into their internal formats. The first call to `scanf()` reads the three letters that specify the bases, the next two calls read the X and Y location, extent and granularity of the the search rectangle and the last call reads in the first, last and increment values that will be used specify the orientation of the third base at each point on the search grid.

Lines 23 and 24 respectively, create the names of the files that will hold the best structure found and the values of the potential energy surface. The file names are created using the builtin `sprintf()`. Like `scanf()` this function also uses its first argument as a format string, used here to construct a string from the data values that follow it in the parameter list. The action of these calls is to replace the each format descriptor (`%s`) with the values of the corresponding string variable in the parameter list. The file names created for the AU:A shown in Figure 3 were AUA.triad.min.pdb and AUA.energy.dat. Format expressions and formatted I/O including the I/O like `sprintf()` are discussed in the sections **Format Expressions** and **Ordinary I/O Functions** of the **nab Language Reference**.

The triad is created in two major steps in lines 26-32. First a Watson/Crick base pair is created with `wc_helix()`. The base pair has an X-offset of 2.25 Å and an inclination of 0.0 meaning it lies in the XY plane. Twist and rise although they are not used in creating a single base pair are also set to 0.0. The X-offset which is that of standard B-DNA was chosen to facilitate extension of triplexes made from the triads created here with standard duplex DNA. Absent this consideration any X-offset including 0.0 would have been satisfactory. A third strand ("third") is added to `m`, the string `tb` is converted into a DNA residue and this residue is added to the new strand. Finally in the coordinates of the third strand are saved in the point array `txyz`. Referring to Figure 3, the third base is located directly on top of the Watson/Crick pair. A purine would have its C4 atom at the origin and its C4-N1 vector along the Y axis; a pyrimidine its C6 at the origin and its C6-N3 vector along the Y axis. Obviously this is not a real structure; however, as will be seen in the next section, this initial placement greatly simplifies the transformations required to explore the search area.

14.13.3. Finding the lowest energy triad

The energy calculation begins in line 34 and extends to line 69. Elements of the general molecular mechanics code skeleton discussed in the **Language Reference** chapter are seen at lines 34-35 and lines 50-51. Initialization takes place in lines 34 and 35 with the call to `getpdb_prm()` to prepare the information needed to compute molecular mechanics energies. The

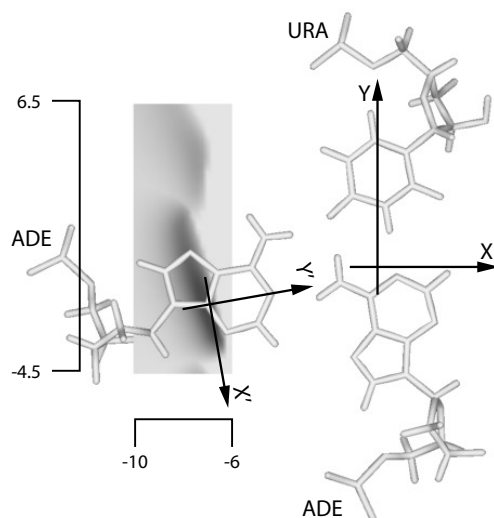


Figure 14.3.: *Minimum energy AUA triad and the potential energy surface.*

force field routine is initialized in line 35, asking that all atoms be allowed to move. The actual energy calculation is done in lines 50 and 51. `setxyz_from_mol()` copies the current conformation of `mol` into the point array `xyz` and then `mme()` evaluates the energy of this conformation. Note that the energy evaluation is in a loop, in this case nested inside the three loops that control the conformational search.

The search area shown in Figure 14.3 is on the left side of the Watson/Crick base pair. This corresponds to inserting the third base into the major groove of the duplex. Now as the third base is initially positioned at the origin with its hydrogen bonding edge pointing towards the top of the page, it must be both moved to the left or in the $-X$ direction and rotated approximately -90° so that its hydrogen bonding sites can interact with those on the left side of the Watson/Crick pair.

The search is executed by the three nested for loops in lines 40, 41 and 43. They control the third base's X and Y position and its orientation in the XY plane. Two transformations are used to place the base. The first step of the placement process is in line 44 where the nab builtin `setmol_from_xyz()` is used to restore the original (untransformed) coordinates of the base. The call to `newtransform()` in line 45 creates a transformation matrix that will point the third base so that its hydrogen bonding sites are aimed in the positive X direction. A second transformation matrix created on line 47 is used to move the properly oriented third base to a point on the search area. The call to `setxyz_from_mol()` extracts the coordinates of this conformation into `xyz` and `mme()` computes and returns its energy.

The remainder of the loop determines if this is either the best overall energy or the best energy for this grid point. Lines 53-57 compute the best energy at this point and lines 58-64 compute the best overall energy. The complexity arises from the fact that the energy returned by `mme()` can be any float value. Thus it is not possible to pick a value that is guaranteed to be higher

than any value returned during the search. The solution is to use the value from the first iteration of the loop as the value to test against. The two variables `mrz` and `brz` are used to indicate the very first iteration and the first iteration of the `rz` loop. The gray rectangle of Figure 14.3 shows the vacuum energy of the best AU:A triad found when the origin of the X' Y' axes are at that point on the rectangle. Darker grays are lower energies. Figure 14.3 shows the best AU:A found.

14.13.4. Assembling the Triads into Dimers

Once the minimized base triads have been created, they must be assembled into triplexes. Since these triplexes are believed to be intermediates in homologous recombination, their structure should be nearly sequence independent. This means that they can be assembled by applying the same set of helical parameters to each optimized triad. However, several things still need to be determined. These are the location of the helical axis and just what helical parameters are to be applied. This code assumes that the three backbone strands are roughly on the surface of a cylinder whose axis is the global helical axis. In particular the helical axis is the center of the circle defined by the three C1' atoms in each triad. While the four circles defined by the four minimized triads are not exactly the same, their radii are within X Å of each other with the XY:X triad having the largest offset of Y Å. The code makes two additional assumptions. The sugar rings are all in the C2'-endo conformation and the triads are not inclined with respect to the helical axis. The program that creates and evaluates the dimers is shown below. A detailed explanation of the program follows the listing.

```

1 // Program 6 - Assemble triads into dimers
2 molecule    gettriad( string mname )
3 {
4     molecule    m;
5     point       p1, p2, p3, pc;
6     matrix      mat;
7
8     if( mname == "a" ){
9         m = getpdb( "ata.triad.min.pdb" );
10        setpoint( m, "A:ADE:C1'", p1 );
11        setpoint( m, "B:THY:C1'", p2 );
12        setpoint( m, "C:ADE:C1'", p3 );
13    }else if( mname == "c" ){
14        m = getpdb( "cgc.triad.min.pdb" );
15        setpoint( m, "A:CYT:C1'", p1 );
16        setpoint( m, "B:GUA:C1'", p2 );
17        setpoint( m, "C:CYT:C1'", p3 );
18    }else if( mname == "g" ){
19        m = getpdb( "gcg.triad.min.pdb" );
20        setpoint( m, "A:GUA:C1'", p1 );
21        setpoint( m, "B:CYT:C1'", p2 );
22        setpoint( m, "C:GUA:C1'", p3 );
23    }else if( mname == "t" ){
24        m = getpdb( "tat.triad.min.pdb" );
25        setpoint( m, "A:THY:C1'", p1 );
26        setpoint( m, "B:ADE:C1'", p2 );

```

14. NAB: Introduction

```
27     setpoint( m, "C:THY:C1'", p3 );
28 }
29 circle( p1, p2, p3, pc );
30 mat = newtransform( -pc.x, -pc.y, -pc.z, 0.0, 0.0, 0.0 );
31 transformmol( mat, m, NULL );
32 setreskind( m, NULL, "DNA" );
33 return( m );
34 };
35
36 int mk_dimer( string ti, string tj )
37 {
38     molecule    mi, mj;
39     matrix      mat;
40     int         sid;
41     float       ri, tw;
42     string      ifname, sfname, mfname;
43     file        idx;
44
45     int         natoms;
46     float       dgrad, fret;
47     float       box[ 3 ];
48     float       xyz[ 1000 ];
49     float       fxyz[ 1000 ];
50     float       energy;
51
52     sid = 0;
53     mi = gettriad( ti );
54     mj = gettriad( tj );
55     mergestr( mi, "A", "last", mj, "A", "first" );
56     mergestr( mi, "B", "first", mj, "B", "last" );
57     mergestr( mi, "C", "last", mj, "C", "first" );
58     connectres( mi, "A", 1, "O3'", 2, "P" );
59     connectres( mi, "B", 1, "O3'", 2, "P" );
60     connectres( mi, "C", 1, "O3'", 2, "P" );
61
62     putpdb( "temp.pdb", mi );
63     mi = getpdb_prm( "temp.pdb", "leaprc.ff99SB", "", 0 );
64
65     ifname = sprintf( "%s%s3.idx", ti, tj );
66     idx = fopen( ifname, "w" );
67     for( ri = 3.2; ri <= 4.4; ri = ri + .2 ){
68         for( tw = 25; tw <= 45; tw = tw + 5 ){
69             sid = sid + 1;
70             fprintf( idx, "%3d %5.1f %5.1f", sid, ri, tw );
71
72             mi = gettriad( ti );
73             mj = gettriad( tj );
74
75             mat = newtransform( 0.0, 0.0, ri, 0.0, 0.0, tw );
```

```

76     transformmol( mat, mj, NULL );
77
78     mergestr( mi, "A", "last", mj, "A", "first" );
79     mergestr( mi, "B", "first", mj, "B", "last" );
80     mergestr( mi, "C", "last", mj, "C", "first" );
81     connectres( mi, "A", 1, "O3'", 2, "P" );
82     connectres( mi, "B", 1, "O3'", 2, "P" );
83     connectres( mi, "C", 1, "O3'", 2, "P" );
84
85     sfname = sprintf( "%s%s3.%03d.pdb", ti, tj, sid );
86     putpdb( sfname, mi );    // starting coords
87
88     natoms = getmolyz( mi, NULL, xyz );
89     mme_init( mi, NULL, "::ZZZ", xyz, NULL );
90
91     dgrad = 3*natoms*0.001;
92     conjgrad( xyz, 3*natoms, fret, mme, dgrad, 10., 100 );
93     energy = mme( xyz, fxyz, 1 );
94
95     setmol_from_xyz( mi, NULL, xyz );
96     mfname = sprintf( "%s%s3.%03d.min.pdb", ti, tj, sid );
97     putpdb( mfname, mi );    // minimized coords
98 }
99 }
100 fclose( idx );
101 };
102
103 int i, j;
104 string ti, tj;
105 for( i = 1; i <= 4; i = i + 1 ){
106     for( j = 1; j <= 4; j = j + 1 ){
107         ti = substr( "acgt", i, 1 );
108         tj = substr( "acgt", j, 1 );
109         mk_dimer( ti, tj );
110     }
111 }

```

Program 6 assembles, minimizes and writes the final energies of a family of dimers for each of the 16 pairs of optimized triads. The program is long but straightforward. It is organized into two subroutines followed by a main program. The first subroutine `gettriad()` is defined in lines 2-34, the second subroutine `mk_dimer()` in lines 36-101 and the main program in lines 103-111. The overall organization is that the main program controls the sequence of the dimers beginning with AA and continuing with AC, AG, ... and on up to TT. Each time it selects the sequence of the dimer, it calls `mk_dimer()` to explore the family of structures defined by variation in the rise and twist. `mk_dimer()` in turn calls `gettriad()` to fetch and orient the specified base triples.

The function `gettriad()` (lines 2-34) takes a string with one of the four values "a", "c", "g" or "t". The if-tree in lines 8-28 uses this string to select the coordinates of the corresponding optimized triad. The if-tree sets the value of the three points p1, p2 and p3 that will be used to define the

14. NAB: Introduction

circle whose center will intersect the global helical axis. Once these points are defined, the nab builtin `circle()` (line 29) returns the center of the circle they define in `pc`. The builtin `circle()` returns a 1 if the three points do not define a circle and a 0 if they do. In this case it is known that the positions of the three C1' atoms are well behaved, so the return value is ignored. The selected triad is properly centered in lines 30-31. Each residue of the triad is set to be of type "DNA" via the call to `setreskind()` in line 32 so that its atomic charges and force field potentials can be set correctly to perform the minimization. The new molecule is returned as the function's value in line 33.

The dimers are created by the function `mk_dimers()` that is defined in lines 36-101. The process uses two stages. The molecule is first prepared for molecular mechanics in lines 53-63 and then dimers are created and minimized in the two nested loops in lines 67-99. The results of the minimizations are stored in a file whose name is derived from the name of the triads in the dimer. For example, the results for an AA would be in the file "aa3.idx". There is one file for each of the 16 dimers. The file name is created in line 65 and opened for writing in line 66. It is closed just before the function returns in line 100. Each line of the file contains a number that identifies the dimer's parameters followed by its rise, twist and final (minimized) energy.

In order to perform molecular on a molecule the nab program must create a parameter structure for it. This structure contains the topology of the molecule and parameters for the various force field terms, such as bond lengths and angles, torsions, chirality and planarity. This is done in lines 53 – 63. The particular dimer is created. The function `gettriad()` is called twice to return the two properly centered triads in the molecules `mi` and `mj`. Next the three strands of `mj` are merged into the three strands of `mi` to create a triplex of length 2. The "A" and "B" strands form the Watson/Crick pairs of the triplex and the "C" strand contains the strand that is parallel to the "A" strand. The three calls to `connectres()` create an O3'-P bond between the newly added residue and the existing residues in each of the three strands. After all this is done, the call to `getpdb_prm()` in line 63 builds the parameter structure, returning 1 on failure and 0 on success.

This section of code seems simple enough except for one thing—the two triads in the dimer are obviously directly on top of each other. However, this is not a problem because `getpdb_prm()` ignores the molecule's coordinates. Instead it uses the molecule's residue names to get each residue's internal coordinates and other information from a library which it uses to up the parameter and topology structure required by the minimization routines.

The dimers are built and minimized in the two nested loops in lines 69-104. The outer loop varies the rise from 3.2 to 4.4 Å by 0.2 Å, and the inner loop varies the twist from 250 to 450 in steps of 50, creating 35 different starting dimers. The variable `sid` is a number that identifies each (rise,twist) pair. It is inserted into the file names of the starting coordinates (lines 85-86) and minimized coordinates (lines 96-97) to make it easy to identify them.

Each dimer is created in lines 72-83. The two specified triads are returned by the calls to `gettriad()` as the molecule's `mi` and `mj`. Next the triad in `mj` is transformed to give it the current rise and twist with respect to the triad in `mi`. The transformed triad in `mj` is merged into `mi` and bonded to `mi`. These starting coordinates are written to a file whose name contains both the dimer sequence and `sid`. For example, the first dimer for AA would be "aa3.01.pdb", the 01 indicating that this dimer used a rise of 3.2 Å and a twist of 250.

The minimization is performed in lines 88-95. The call to `setxyz_from_mol()` extracts the current atom positions of `mi` into the array `xyz`. The coordinates are passed to `mme_init()` which initializes the molecular mechanics system. The actual minimization is done with the call to

conjgrad() which performs 100 cycles of conjugate gradient minimization, printing the results every 10 cycles. The final energy is written to the file idx and the molecule's original coordinates are updated with the minimized coordinates by the call to setmol_from_xyz(). Once all dimers have been made for this sequence the loops terminate. The last thing done by mk_dimer() before it returns to the main program is to close the file containing the energy results for this family of dimer.

15. NAB: Language Reference

15.1. Introduction

nab is a computer language used to create, modify and describe models of macromolecules, especially those of unusual nucleic acids. The following sections provide a complete description of the nab language. The discussion begins with its lexical elements, continues with sections on expressions, statements and user defined functions and concludes with an explanation of each of nab's builtin functions. Two appendices contain a more detailed and formal description of the lexical and syntactic elements of the language including the actual lex and yacc input used to create the compiler. Two other appendices describe nab's internal data structures and the C code generated to support some of nab's higher level operations.

15.2. Language Elements

An nab program is composed of several basic lexical elements: identifiers, reserved words, literals, operators and special characters. These are discussed in the following sections.

15.2.1. Identifiers

An identifier is a sequence of letters, digits and underscores beginning with a letter. Upper and lower case letters are distinct. Identifiers are limited to 255 characters in length. The underscore (`_`) is a letter. Identifiers beginning with underscore must be used carefully as they may conflict with operating system names and nab created temporaries. Here are some nab identifiers.

```
mol i3 twist TWIST Watson_Crick_Base_Pair
```

15.2.2. Reserved Words

Certain identifiers are reserved words, special symbols used by nab to denote control flow and program structure. Here are the nab reserved words:

allocate	assert	atom	bounds	break
continue	deallocate	debug	delete	dynamic
else	file	for	float	hashed
if	in	int	matrix	molecule
point	residue	return	string	while

15.2.3. Literals

Literals are self defining terms used to introduce constant values into expressions. nab provides three types of literals: integers, floats and character strings. Integer literals are sequences of one or more decimal digits. Float literals are sequences of decimal digits that include a decimal point and/or are followed by an exponent. An exponent is the letter e or E followed by an optional + or - followed by one to three decimal digits. The exponent is interpreted as “times 10 to the power of *exp*” where *exp* is the number following the e or E. All numeric literals are base 10. Here are some integer and float literals:

1 3.14159 5 .234 3.0e7 1E-7

String literals are sequences of characters enclosed in double quotes ("). A double quote is placed into a string literal by preceding it with a backslash (\). A backslash is inserted into a string by preceding it with a backslash. Strings of zero length are permitted.

"" "a string" "string with a \" "string with a \\"

Non-printing characters are inserted into strings via escape sequences: one to three characters following a backslash. Here are the nab string escapes and their meanings:

<code>\a</code>	Bell (a for audible alarm)
<code>\b</code>	Back space
<code>\f</code>	Form feed (new page)
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\"</code>	Literal double quote
<code>\\</code>	Literal backspace
<code>\ooo</code>	Octal character
<code>\xhh</code>	Hex character (hh is 1 or 2 hex digits)

Here are some strings with escapes:

"Molecule\tResidue\tAtom\n"
"\252Real quotes\272"

The second string has octal values, `\252`, the left double quote, and `\272`, the right double quote.

15.2.4. Operators

nab uses several additional 1 or 2 character symbols as operators. Operators combine literals and identifiers into expressions.

Operator	Meaning	Precedence	Associates
()	expression grouping	9	
[]	array indexing	9	
.	select attribute	8	
unary -	negation	8	right to left
!	not	8	
^	cross product	6	left to right
@	dot product	6	
*	multiplication	6	left to right
/	division	6	left to right
%	modulus	6	left to right
+	addition, concatenation	5	left to right
binary -	subtraction	5	left to right
<	less than	4	
<=	less than or equal to	4	
==	equal	4	
!=	not equal	4	
>=	greater than or equal to	4	
>	greater than	4	
=~	match	4	
!~	doesn't match	4	
in	hashed array member or atom in molecule	4	
&&	and	3	
	or	2	
=	assignment	1	right to left

15.2.5. Special Characters

nab uses braces ({}) to group statements into compound statements and statements and declarations into function bodies. The semicolon (;) is used to terminate statements. The comma (,) separates items in parameter lists and declarations. The sharp (#) used in column 1 designates a preprocessor directive, which invokes the standard C preprocessor to provide constants, macros and file inclusion. A # in any other column, except in a comment or a literal string is an error. Two consecutive forward slashes (//) indicate that the rest of the line is a comment which is ignored. All other characters except white space (spaces, tabs, newlines and formfeeds) are illegal except in literal strings and comments.

15.3. Higher-level constructs

15.3.1. Variables

A variable is a name given to a part of memory that is used to hold data. Every nab variable has type which determines how the computer interprets the variable's contents. nab provides

10 data types. They are the numeric types `int` and `float` which are translated into the underlying C compiler's `int` and `double` respectively.*

The `string` type is used to hold null (zero byte) terminated (C) character strings. The `file` type is used to access files (equivalent to C's `FILE *`). There are three types—`atom`, `residue` and `molecule` for creating and working with molecules. The `point` type holds three `float` values which can represent the X, Y and Z coordinates of a point or the components of a 3-vector. The `matrix` type holds 16 `float` values in a 4×4 matrix and the `bounds` type is used to hold distance bounds and other information for use in distance geometry calculations.

`nab` string variables are mapped into C `char *` variables which are allocated as needed and freed when possible. However, all of this is invisible at the `nab` level where strings are atomic objects. The `atom`, `residue`, `molecule` and `bounds` types become pointers to the appropriate C structs. `point` and `matrix` are implemented as `float [3]` and `float [4][4]` respectively. Again the `nab` compiler automatically generates all the C code required to make these types appear as atomic objects.

Every `nab` variable must be declared. All declarations for functions or variables in the main block must precede the first executable statement of that block. Also all declarations in a user defined `nab` function must precede the first executable statement of that function. An `nab` variable declaration begins with the reserved word that specifies the variable's type followed by a comma separated list of identifiers which become variables of that type. Each declaration ends with a semicolon.

```
int i, j, j;  
matrix mat;  
point origin;
```

Six `nab` types—`string`, `file`, `atom`, `residue`, `molecule` and `bounds` use the predefined identifier `NULL` to indicate a non-existent object of these types. `nab` builtin functions returning objects of these types return `NULL` to indicate that the object could not be created. `nab` considers a `NULL` value to be false. The empty `nab` string "" is *not* equal to `NULL`.

15.3.2. Attributes

Four `nab` types—`atom`, `residue`, `molecule` and `point`—have attributes which are elements of their internal structure directly accessible at the `nab` level. Attributes are accessed via the select operator (`.`) which takes a variable as its left hand operand and an attribute name (an identifier) as its right. The general form is

```
var.attr
```

Most attributes behave exactly like ordinary variables of the same type. However, some attributes are read only. They are not permitted to appear as the left hand side of an assignment. When a read only attribute is passed to an `nab` function, it is copied into temporary variable which in turn is passed to the function. Read only attributes are not permitted to appear as destination variables in `scanf()` parameter lists. Attribute names are kept separate from variable and function names and since attributes can only appear to the right of select there is no conflict between variable and attribute names. For example, if `x` is a point, then

x // the point variable **x**
x.x // x coordinate of **x**
.x // Error!

Here is the complete list of nab attributes.

Atom attributes	Type	Write?	Meaning
atomname	string	yes	Ordinarily taken from columns 13-16 of an input pdb file, or from a residue library. Spaces are removed.
atomnum	int	no	The number of the atom starting at 1 for <i>each</i> strand in the molecule.
tatomnum	int	no	The <i>total</i> number of the atom starting at 1. Unlike atomnum, tatomnum does not restart at 1 for each strand.
fullname	string	no	The fully qualified atom name, having the form <i>strandnum:resnum:atomname</i> .
resid	string	yes	The <i>resid</i> of the residue containing this atom; see the Residue attributes table.
resname	string	yes	The name of the residue containing this atom.
resnum	int	no	The number of the residue containing the atom. resnum starts at 1 for <i>each</i> strand.
tresnum	int	no	The <i>total</i> number of the residue containing this atom starting at 1. Unlike resnum, tresnum does not restart at 1 for each strand.
strandname	string	yes	The name of the strand containing this atom.
strandnum	int	no	The number of the strand containing this atom.
pos	point	yes	point variable giving the atom's position.
x,y,z	float	yes	The Cartesian coordinates of this atom
charge	float	yes	Atomic charge
radius	float	yes	Dielectric radius
int1	int	yes	User-definable integer
float1	float	yes	User-definable float

Residue attributes	Type	Write?	Meaning
resid	string	yes	A 6-character string, ordinarily taken from columns 22-27 of a PDB file. It can be re-set to something else, but should always be either empty or exactly 6 characters long, since this string is used (if it is not empty) by <i>putpdb</i> .
resname	string	yes	Three-character identifier
resnum	int	no	The number of the residue. <i>resnum</i> starts at 1 for <i>each</i> strand.
tresnum	int	no	The <i>total</i> number of the residue, starting at 1. Unlike <i>resnum</i> , <i>tresnum</i> does not restart at 1 for each strand.
strandname	string	yes	The name of the strand containing this residue.
strandnum	int	no	The number of the strand containing this residue.

Molecule attributes	Type	Write?	Meaning
natoms	int	no	The total number of atoms in the molecule.
nresidues	int	no	The total number of residues in the molecule.
nstrands	int	no	The total number of strands in the molecule.

15.3.3. Arrays

nab supports two kinds of arrays—ordinary arrays where the selector is a comma separated list of integer expressions and associative or “hashed” arrays where the selector is a character string. The set of character strings that is associated with data in a hashed array is called its keys. Array elements may be of any *nab* type. All the dimensions of an ordinary array are indexed from 1 to N_d , where N_d is the size of the d th dimension. Non parameter array declarations are similar to scalar declarations except the variable name is followed by either a comma separated list of integer constants surrounded by square brackets ([]) for ordinary arrays or the reserved word *hashed* in square brackets for associative arrays. Associative arrays have no predefined size.

```
float energy[ 20 ], surface[ 13,13 ];
int attr[ dynamic, dynamic ];
molecule structs[ hashed ];
```

The syntax for multi-dimensional arrays like that for Fortran, not C. The *nab2c* compiler linearizes all index references, and the underlying C code sees only single-dimension arrays. Arrays are stored in “column-order”, so that the most-rapidly varying index is the first index, as in Fortran. Multi-dimensional int or float arrays created in *nab* can generally be passed to Fortran routines expecting the analogous construct.

Dynamic arrays are not allocated space upon program startup, but are created and freed by the *allocate* and *deallocate* statements:

```

allocate attr[ i, j ];
....
deallocattr attr;

```

Here *i* and *j* must be integer expressions that may be evaluated at run-time. It is an error (generally fatal) to refer to the contents of such an array before it has been allocated or after it has been deallocated.

15.3.4. Expressions

Expressions use operators to combine variables, constants and function values into new values. nab uses standard algebraic notation ($a+b*c$, etc) for expressions. Operators with higher precedence are evaluated first. Parentheses are used to alter the evaluation order. The complete list of nab operators with precedence levels and associativity is listed under **Operators**.

nab permits mixed mode arithmetic in that int and float data may be freely combined in expressions as long as the operation(s) are defined. The only exceptions are that the modulus operator (%) does not accept float operands, and that subscripts to ordinary arrays must be integer valued. In all other cases except parameter passing and assignment, when an int and float are combined by an operator, the int is converted to float then the operation is executed. In the case of parameter passing, nab requires (but does not check) that actual parameters passed to functions have the same type as the corresponding formal parameters. As for assignment (=) the right hand side is converted to the type of the left hand side (as long as both are numeric) and then assigned. nab treats assignment like any other binary operator which permits multiple assignments ($a=b=c$) as well as “embedded” assignments like:

```

if( mol = newmolecule() ) ...

```

nab relational operators are strictly binary. Any two objects can be compared provided that both are numeric, both are string or both are the same type. Comparisons for objects other than int, float and string are limited to tests for equality. Comparisons between file, atom, residue, molecule and bounds objects test for “pointer” equality, meaning that if the pointers are the same, the objects are same and thus equal, but if the pointers are different, no inference about the actual objects can be made. The most common comparison on objects of these types is against NULL to see if the object was correctly created. Note that as nab considers NULL to be false the following expressions are equivalent.

```

if( var == NULL )... is the same as if( !var )...
if( var != NULL )... is the same as if( var )...

```

The Boolean operators && and || evaluate only enough of an expression to determine its truth value. nab considers the value 0 to be false and *any* non-zero value to be true. nab supports direct assignment and concatenation of string values. The infix + is used for string concatenation.

nab provides several infix vector operations for point values. They can be assigned and point valued functions are permitted. Two point values can be added or subtracted. A point can be multiplied or divided by a float or an int. The unary minus can be applied to a point which has the same effect as multiplying it by -1. Finally, the at sign (@) is used to form the dot product of two points and the circumflex (^) is used to form their cross product.

15.3.5. Regular expressions

The \sim and $!\sim$ operators (match and not match) have strings on the left-hand-sides and *regular expression* strings on their right-hand-sides. These regular expressions are interpreted according to standard conventions drawn from the UNIX libraries.

15.3.6. Atom Expressions

An atom expression is a character string that contains one or more patterns that match a set of atom names in a molecule. Atom expressions contain three substrings separated by colons (:). They represent the strand, residue and atom parts of the atom expression. Each subexpression consists of a comma (,) separated list of patterns, or for the residue part, patterns and/or number ranges. Several atom expressions may be placed in a single character string by separating them with the vertical bar (|).

Patterns in atom expressions are similar to Unix shell expressions. Each pattern is a sequence of 1 or more single character patterns and/or stars (*). The star matches *zero* or more occurrences of *any* single character. Each part of an atom expression is composed of a comma separated list of limited regular expressions, or in the case of the residue part, limited regular expressions and/or ranges. A *range* is a number or a pair of numbers separated by a dash. A *regular expression* is a sequence of ordinary characters and “metacharacters”. Ordinary characters represent themselves, while the metacharacters are operators used to construct more complicated patterns from the ordinary characters. All characters except ?, *, [,], -, ,(comma), : and | are ordinary characters. Regular expressions and the strings they match follow these rules.

aexpr	matches
x	An ordinary character matches itself.
?	A question mark matches any single character.
*	A star matches any run of zero or more characters. The pattern * matches anything.
[xyz]	A character class. It matches a single occurrence of any character between the [and the].
[^xyz]	A “negated” character class. It matches a single occurrence of any character not between the ^ and the]. Character ranges, f-l, are permitted in both types of character class. This is a shorthand for all characters beginning with f up to and including l. Useful ranges are 0-9 for all the digits and a-zA-Z for all the letters.
-	The dash is used to delimit ranges in characters classes and to separate numbers in residue ranges.
\$	The dollar sign is used in a residue range to represent the “last” residue without having to know its number.
,	The comma separates regular expressions and/or ranges in an atom expression part.
:	The colon separates the parts of an atom expression.
	The vertical bar separates atom expressions in the same character string.
\	The backslash is used as an escape. Any character including metacharacters following a backslash matches itself.

Atom expressions match the *entire* name. The pattern `C`, matches only `C`, not `CA`, `HC`, etc. To match any name that begins with `C` use `C*`; to match any name that ends with `C`, use `*C`; to match any name containing a `C`, use `*C*`. A table of examples was given in chapter 2.

15.3.7. Format Expressions

A format expression is a special character string that is used to direct the conversion between the computer's internal data representations and their character equivalents. `nab` uses the underlying C compiler's `printf()/scanf()` system to provide formatted I/O. This section provides a short introduction to this system. For the complete description, consult any standard C reference. Note that since `nab` supports fewer types than its underlying C compiler, formatted I/O options pertaining to the data subtypes (`h`, `l`, `L`) are not applicable to `nab` format expressions.

An input format string is a mixture of ordinary characters, *spaces* and format descriptors. An output format string is mixture of ordinary characters including spaces and format descriptors. Each format descriptor begins with a percent sign (%) followed by several optional characters describing the format and ends with single character that specifies the type of the data to be converted. Here are the most common format descriptors. The ... represent optional characters described below.

<code>%...c</code>	convert a character
<code>%...d</code>	convert and integer
<code>%...lf</code>	convert a float
<code>%...s</code>	convert a string
<code>%%</code>	convert a literal %

Input and output format descriptors and format expressions resemble each other and in many cases the same format expression can be used for both input and output. However, the two types of format descriptors have different options and their actions are sufficiently distinct to consider in some detail. Generally, C based formatted output is more useful than C based formatted input.

When an input format expression is executed, it is scanned at most once from left to right. If the current format expression character is an ordinary character (anything but space or %), it must match the current character in the input stream. If they match then both the current character of the format expression and current character of the stream are advanced one character to the right. If they don't match, the scan ends. If the current format expression character is a space or a run of spaces and if the current input stream is one or more "white space" characters (space, tab, *newline*), then both the format and input stream are advanced to the next non-white space character. If the input format is one or more spaces but the current character of the input stream is non-blank, then only the format expression is advanced to the next non-blank character. If the current format character is a percent sign, the format descriptor is used to convert the next "field" in the input stream. A field is a sequence of non-blank characters surrounded by white space or the beginning or end of the stream. This means that a format descriptor will *skip* white space including newlines to find non blank characters to convert, even if it is the first element of the format expression. This implicit scanning is what limits the ability of C based formatted input to read fixed format data that contains any spaces.

15. NAB: Language Reference

Note that `lf` is used to input a NAB *float* variable, rather than the `f` argument that would be used in C. This is because *float* in NAB is converted to *double* in the output C code (see *defreal.h* if you want to change this behavior.) Ideally, the NAB compiler should parse the format string, and make the appropriate substitutions, but this is not (yet) done: NAB translates the format string directly into the C code, so that the NAB code must also generally use `lf` as a format descriptor for floating point values.

`nab` input format descriptors have two options, a field width, and an assignment suppression indicator. The field width is an integer which specifies how much of current *field* and not the input stream is to be converted. Conversion begins with the first character of the field and stops when the correct number of characters have been converted or white space is encountered. A star (*) option indicates that the field is to be converted, but the result of the conversion is not stored. This can be used to skip unwanted items in a data stream. The order of the two options does not matter.

The execution of an output format expression is somewhat different. It is scanned once from left to right. If the current character is not a percent sign, it placed on the output stream. Thus spaces have no special significance in formatted output. When the scan encounters a percent sign it replaces the entire format descriptor with the properly formatted value of the corresponding output expression.

Each output format descriptor has four optional attributes—width, alignment, padding and precision. The width is the *minimum* number of characters the data is to occupy for output. Padding controls how the field will be filled if the number of characters required for the data is less than the field width. Alignment specifies whether the data is to start in the first character of the field (left aligned) or end in the last (right aligned). Finally precision, which applies only to string and float conversions controls how much of the string is be converted or how many digits should follow the decimal point.

Output field attributes are specified by optional characters between the initial percent sign and the final data type character. Alignment is first, with left alignment specified by a minus sign (-). Any other character after the percent sign indicates right alignment. Padding is specified next. Padding depends on both the alignment and the type of the data being converted. Character conversions (`%c`) are always filled with spaces, regardless of their alignment. Left aligned conversions are also always filled with spaces. However, right aligned string and numeric conversions can use a 0 to indicate that left fill should be zeroes instead of spaces. In addition numeric conversions can also specify an optional + to indicate that non-negative numbers should be preceded by a plus sign. The default action for numeric conversions is that negative numbers are preceded by a minus, and other numbers have no sign. If both 0 and + are specified, their order does not matter.

Output field width and precision are last and are specified by one or two integers or stars (*) separated by a period (.). The first number (or star) is the field width, the second is its precision. If the precision is not specified, a default precision is chosen based on the conversion type. For floats (`%f`), it is six decimal places and for strings it is the entire string. Precision is not applicable to character or integer conversions and is ignored if specified. Precision may be specified without the field width by use of single integer (or star) preceded by a period. Again, the action is conversion type dependent. For strings (`%s`), the action is to print the first *N* characters of the string or the entire string, whichever is shorter. For floats (`%f`), it will print *N* decimal places but will extend the field to whatever size if required to print the whole number

part of the float. The use of the star (*) as an output width or precision indicates that the width or precision is specified as the next argument in the conversion list which allows for runtime widths and precisions.

Output format options	
<i>Alignment</i>	
-	left justified
default	right justified
<i>Padding</i>	
0	%d, %f, %s only, left fill with zeros, right fill with spaces.
+	%d, %f only, precede non-negative numbers with a +.
default	left and right fill with spaces.
<i>Width & precision</i>	
W	<i>minimum</i> field width of <i>W</i> . <i>W</i> is either an integer or a * where the star indicates that the width is the next argument in the parameter list.
W.P	<i>minimum</i> field width of <i>W</i> , with a precision of <i>P</i> . <i>W,P</i> are integers or stars, where stars indicate that they are to be set from the appropriate arguments in the parameter list. Precision is ignored for %c and %d.
.P	%s, print the first <i>P</i> characters of the string or the entire string whichever is shorter. %f, print <i>P</i> decimal places in a field wide enough to hold the integer and fractional parts of the number. %c and %d, use whatever width is required. Again <i>P</i> is either an integer or a star where the star indicates that it is to be taken from the next expression in the parameter list.
default	%c, %d, %s, use whatever width is required to exactly hold the data. %f, use a precision of 6 and whatever width is required to hold the data.

15.4. Statements

nab statements describe the action the nab program is to perform. The expression statement evaluates expressions. The if statement provides a two way branch. The while and for statements provide loops. The break statement is used to “short circuit” or exit these loops. The continue statement advances a for loop to its next iteration. The return statement assigns a function’s value and returns control to the caller. Finally a list of statements can be enclosed in braces ({}) to create a compound statement.

15.4.1. Expression Statement

An expression statement is an expression followed by a semicolon. It evaluates the expression. Many expression statements include an assignment operator and its evaluation will update the values of those variables on the left hand side of the assignment operator. These kinds of expression statements are usually called “assignment statements” in other languages. Other expression statements consist of a single function call with its result ignored. These statements take the place of “call statements” in other languages. Note that an expression statement can contain *any* expression, even ones that have no lasting effect.

```
mref = getpdb( "5p21.pdb" ); // "assignment" stmt
m = getpdb( "6q21.pdb" );
superimpose( m,"::CA",mref,"::CA" ); // "call" stmt
0; // expression stmt.
```

15.4.2. Delete Statement

nab provides the delete statement to remove elements of hashed arrays. The syntax is

```
delete h_array[ str ];
```

where *h_array* is a hashed array and *str* is a string valued expression. If the specified element is in *h_array* it is removed; if not, the statement has no effect.

15.4.3. If Statement

The if statement is used to choose between two options based on the value of the if expression. There are two kinds of if statements—the simple if and the if-else. The simple if contains an expression and a statement. If the expression is true (any non-zero value), the statement is executed. If the expression is false (0), the statement is skipped.

```
if( expr ) true_stmt;
```

The if-else statement places two statements under control of the if. One is executed if the expression is true, the other if it is false.

```
if( expr )
    true_stmt;
else
    false_stmt;
```

15.4.4. While Statement

The while statement is used to execute the statement under its control as long as the the while expression is true (non-zero). A compound statement is required to place more than one statement under the while statement's control.

```
while( expr ) stmt;
while( expr ) {
    stmt_1;
    stmt_2;
    ...
    stmt_N;
}
```

15.4.5. For Statement

The for statement is a loop statement that allows the user to include initialization and an increment as well as a loop condition in the loop header. The single statement under the control of the for statement is executed as long as the condition is true (non-zero). A compound statement is required to place more than one statement under control of a for. The general form of the for statement is

```
for( expr_1; expr_2; expr_3 ) stmt;
```

which behaves like

```
expr_1;  
while( expr_2 ) {  
    stmt;  
    expr_3;  
}
```

expr_3 is generally an expression that computes the next value of the loop index. Any or all of *expr_1*, *expr_2* or *expr_3* can be omitted. An omitted *expr_2* is considered to be true, thus giving rise to an “infinite” loop. Here are some for loops.

```
for( i = 1; i <= 10; i = i + 1 )  
printf( "%3d\n", i ); // print 1 to 10  
for( ; ; ) // "infinite" loop  
{  
    getcmd( cmd ); // Exit better be in  
    docmd( cmd ); // getcmd() or docmd().  
}
```

nab also includes a special kind of for statement that is used to range over all the entries of a hashed array or all the atoms of a molecule. The forms are

```
// hashed version  
for( str in h_array ) ~stmt;  
// molecule version  
for( a in mol ) ~stmt;
```

In the first code fragment, *str* is string and *h_array* is a hashed array. This loop sets *str* to each key or string associated with data in *h_array*. Keys are returned in increasing lexical order. In the second code fragment *a* is an atom and *mol* is a molecule. This loop sets *a* to each atom in *mol*. The first atom is the first atom in the first residue of the first strand. Once all the atoms in this residue have been visited, it moves to the first atom of the next residue in the first strand. Once all atoms in all residues in the first strand have been visited, the process is repeated on the second and subsequent strands in *mol* until all atoms have been visited. The order of the strands of molecule is the order in which they were created using `addstrand()`. Residues in each strand are numbered from 1 to *N*. The order of the atoms in a residue is the order in which the atoms were listed in the reslib entry or pdbfile that that residue derives from.

15.4.6. Break Statement

Execution of a break statement exits the immediately enclosing for or while loop. By placing the break under control of an if conditional exits can be created. break statements are only permitted inside while or for loops.

```
for( expr_1; expr_2; expr_3 ) {  
    ...  
    if( expr ) break; // "break" out of loop  
    ...  
}
```

15.4.7. Continue Statement

Execution of a continue statement causes the immediately enclosing for loop to skip to its next value. If the next value causes the loop control expression to be false, the loop is exited. continue statements are permitted only inside while and for loops.

```
for( expr_1; expr_2; expr_3 ) {  
    ... if( expr ) continue; // "continue" with next value  
    ...  
}
```

15.4.8. Return Statement

The return statement has two uses. It terminates execution of the current function returning control to the point immediately following the call and when followed by an optional expression, returns the value of the expression as the value of the function. A function's execution also ends when it "runs off the bottom". When a function executes the last statement of its definition, it returns even if that statement is not a return. The value of the function in such cases is undefined.

```
return expr; // return the value expr  
return; // return, function value undefined.
```

15.4.9. Compound Statement

A compound statement is a list of statements enclosed in braces. Compound statements are required when a loop or an if has to control more than one statement. They are also required to associate an else with an if other than the nearest unpaired one. Compound statements may include other compound statements. Unlike C, nab compound statements are not blocks and may not include declarations.

15.5. Structures

A struct is collection of data elements, where the elements are accessed via their names. Unlike arrays which require all elements of an array to have the same type, elements of a

structure can have different types. Users define a struct via the reserved word ‘struct’. Here’s a simple example, a struct that could be used to hold a complex number.

```
struct cmplx_t { float r, i; } c;
```

This declares a nab variable, ‘c’, of user defined type ‘struct cmplx_t’. The variable, c, has two float valued elements, ‘c.r’, ‘c.i’ which can be used like any other nab float variables:

```
c.r = -2.0; ... 5*c.i ... printf( "c.r,i = %8.3f, %8.3f\n", c.r, c.i );
```

Now, let’s look more closely at that struct declaration.

```
struct cmplx_t { float r, i; } c;
```

As mentioned before, every nab struct begins with the reserved word struct. This must be followed by an identifier called the structure tag, which in this example is ‘cmplx_t’. Unlike C/C++, a nab struct can not be anonymous.

Following the structure tag is a list of the struct’s element declarations surrounded by a left and right curly bracket. Element declarations are just like ordinary nab variable declarations: they begin with the type, followed by a comma separated list of variables and end with a semicolon. nab structures must contain at least one declaration containing at least one variable. Also, nab struct elements are currently restricted to scalar values of the basic nab types, so nab structs can not contain arrays or other structs. Note that in our example, both elements are in one declaration, but two declarations would have worked as well.

The whole assembly ‘struct ... }’ serves to define a new type which can be used like any other nab type to declare variables of that type, in this example, a single scalar variable, ‘c’. And finally, like all other nab variable declarations, this one also ends with a semicolon.

Although nab structs can not contain arrays, nab allows users to create arrays, including dynamic and hashed arrays of structs. For example

```
struct cmplx_t { float r, i; } a[ 10 ], da[ dynamic ], ha[ hashed ];
```

declares an ordinary, dynamic and hashed array of struct cmplx_t.

Up til now, we’ve only looked at complete struct declaration. Our example

```
struct cmplx_t { float r, i; } c;
```

contains all the parts of a struct declaration. However there are two other forms of struct declarations. The first one is to define a type, as opposed to declaring variables:

```
struct cmplx_t { float r, i; };
```

defines a new type ‘struct cmplx_t’ but does not declare any variables of this type. This is quite useful in that the type can be placed in a header file allowing it to be shared among parts of a larger program.

The other form of a struct declaration is this short form:

```
struct cmplx_t cv1, cv2;
```

15. NAB: Language Reference

This form can only be used once the type has been defined, either via a type declaration (ie not variable) or a complete type + variable declaration. In fact, once a struct type has been defined, all subsequent declarations of variables of that type, including parameters, must use the short form.

```
struct cmplx_t { float r, i; }; // define type type `struct cmplx_t'
struct cmplx_t c, ct[ 10 ]; // define some vars
int f( int s, struct cmplx_t ct[1] ) // func taking array of
                                   // struct cmplx_t { ... };
```

15.6. Functions

A function is a named group of declarations and statements that is executed as a unit by using the function's name in an expression. Functions may include special variables called parameters that enable the same function to work on different data. All nab functions return a value which can be ignored in the calling expression. Expression statements consisting of a single function call where the return value is ignored resemble procedure call statements in other languages.

All parameters to user defined nab functions are passed by reference. This means that each nab parameter operates on the actual data that was passed to the function during the call. Changes made to parameters during the execution of the function will persist after the function returns. The only exception to this is if an expression is passed in as a parameter to a user defined nab function. In this case, nab evaluates the expression, stores its value in a compiler created temporary variable and uses that temporary variable as the actual parameter. For example if a user were to pass in the constant 1 to an nab function which in turned used it and then assigned it the value 6, the 6 would be stored in the temporary location and the external 1 would be unchanged.

15.6.1. Function Definitions

An nab function definition begins with a header that describes the function value type, the function name and the parameters if any. If a function does not have parameters, an empty parameter list is still required. Following the header is a list of declarations and statements enclosed in braces. The function's declarations must precede all of its statements. A function can include zero or more declarations and/or zero or more statements. The empty function—no declarations and no statements is legal.

The function header begins with the reserved word specifying the type of the function. All nab functions must be typed. An nab function can return a single value of any nab type. nab functions can not return nab arrays. Following the type is an identifier which is the name of the function. Each parameter declaration begins with the parameter type followed by its name. Parameter declarations are enclosed in parentheses and separated by commas. If a function has no parameters, there is nothing between the parentheses. Here is the general form of a function definition:

```
ftype fname( ptype1 parm1, ... )
```



```

{
    decls
    stmts
};

```

15.6.2. Function Declarations

nab requires that every function be declared or made known to the compiler before it is used. Unfortunately this is not possible if functions used in one source file are defined in other source files or if two functions are mutually recursive. To solve these problem, nab permits functions to be declared as well as defined. A function declaration resembles the header of a function definition. However, in place of the function body, the declaration ends with a semicolon or a semicolon preceded by either the word `c` or the word `fortran` indicating the external function is written in C or Fortran instead of nab.

```
ftype fname( ptype1 parm1, ... ) flang;
```

15.7. Points and Vectors

The nab type point is an object that holds three float values. These values can represent the X, Y and Z coordinates of a point or the components of 3-vector. The individual elements of a point variable are accessed via attributes or suffixes added to the variable name. The three point attributes are "x", "y" and "z". Many nab builtin functions use, return or create point values. When used in this context, the three attributes represent the point's X, Y and Z coordinates. nab allows users to combine point values with numbers in expressions using conventional algebraic or infix notation. nab does not support operations between numbers and points where the number must be converted into a vector to perform the operation. For example, if `p` is a point then the expression `p + 1.` is an error, as nab does not know how to expand the scalar `1.` into a 3-vector. The following table contains nab point and vector operations. `p`, `q` are point variables; `s` a numeric expression.

Operator	Example	Precedence	Explanation
<i>Unary -</i>	- <code>p</code>	8	Vector negation, same as <code>-1 * p.</code>
<code>^</code>	<code>p ^ q</code>	7	Compute the cross or vector product of <code>p</code> , <code>q</code> .
<code>@</code>	<code>p @ q</code>	6	Compute the scalar or dot product of <code>p</code> , <code>q</code> .
<code>*</code>	<code>s * p</code>	6	Multiply <code>p</code> by <code>s</code> , same as <code>p * s.</code>
<code>/</code>	<code>p / s</code>	6	Divide <code>p</code> by <code>s</code> , <code>s / p</code> not allowed.
<code>+</code>	<code>p + q</code>	5	Vector addition
<i>Binary -</i>	<code>p - q</code>	5	Vector subtraction
<code>==</code>	<code>p == q</code>	4	Test if <code>p</code> and <code>q</code> equal.
<code>!=</code>	<code>p != q</code>	4	Test if <code>p</code> and <code>q</code> are different.
<code>=</code>	<code>p = q</code>	1	Set the value of <code>p</code> to <code>q</code> .

15.8. String Functions

nab provides the following awk-like string functions. Unlike awk, the nab functions do not have optional parameters or builtin variables that control the actions or receive results from these functions. nab strings are indexed from 1 to N where N is the number of characters in the string.

```
int length( string s );
int index( string s, string t );
int match( string s, string r, int rlength );
string substr( string s, int pos, int len );
int split( string s, string fields[], string fsep );
int sub( string r, string s, string t );
int gsub( string r, string s, string t );
```

length() returns the length of the string *s*. Both "" and NULL have length 0. index() returns the position of the left most occurrence of *t* in *s*. If *t* is not in *s*, index() returns 0. match returns the position of the longest leftmost substring of *s* that matches the regular expression *r*. The length of this substring is returned in *rlength*. If no substring of *s* matches *r*, match() returns 0 and *rlength* is set to 0. substr() extracts the substring of length *len* from *s* beginning at position *pos*. If *len* is greater than the rest of the string beginning at *pos*, return the substring from *pos* to N where N is the length of the string. If *pos* is < 1 or $> N$, return "".

split() partitions *s* into fields separated by *fsep*. These field strings are returned in the array *fields*. The number of fields is returned as the function value. The array *fields* must be allocated before split() is called and must be large enough to hold all the field strings. The action of split() depends on the value of *fsep*. If *fsep* is a string containing one or more blanks, the fields of *s* are considered to be separated by *runs* of white space. Also, leading and trailing white space in *s* do not indicate an empty initial or final field. However, if *fsep* contains any value but blank, then fields are considered to be delimited by *single* characters from *fsep* and initial and/or trailing *fsep* characters do represent initial and/or trailing fields with values of "". NULL and the empty string "" have 0 fields. If both *s* and *fsep* are composed of only white space then *s* also has 0 fields. If *fsep* is not white space and *s* consists of nothing but characters from *fsep*, *s* will have $N + 1$ fields of "" where N is the number of characters of *s*.

sub() replaces the leftmost, longest substring of the *target string* *t* that matches the *regular expression* *r* with the *substitution string* *s*. gsub() replaces all non-overlapping substrings of *t* that match the regular expression *r* with the string *s*. Each function returns the number of substitutions made. Unlike awk, the regular expression *r* is a string variable, with no surrounding '/' characters. For example:

```
int nmatch;
string regexp, substitute, target;
target = "water, water, everywhere";
regexp = "at";
substitute = "ith";
nmatch = gsub( regexp, substitute, target);
```

After this, *target* will contain "wither, wither, everywhere", and *nmatch* will be 2.

The special substitute character '&' stands for the precise substring matched by the regular expression. Hence

```
target = "daabaaa";
sub( "a+", "c&c", target);
```

will yield "dcaacbbaaa". Note what "leftmost, longest substring" means here: "leftmost" takes precedence over "longest".

15.9. Math Functions

nab provides the builtin mathematical functions shown in Table 15.1. Since nab is intended for chemical structure calculations which always measure angles in degrees, the argument to the trig functions—`cos()`, `sin()` and `tan()`—and the return value of the inverse trig functions—`acos()`, `asin()`, `atan()` and `atan2()`—are in degrees instead of radians as they are in other languages. Note that the pseudo-random number functions have a different calling sequence than in earlier versions of NAB; you may have to edit and re-compile earlier programs that used those routines.

15.10. System Functions

```
int exit( int i );
int system( string cmd );
```

The function `exit()` terminates the calling nab program with return status `i`. `system()` invokes a subshell to execute `cmd`. The subshell is always `/bin/sh`. The return value of `system()` is the return value of the subshell and not the command it executed.

15.11. I/O Functions

nab uses the C I/O model. Instead of special I/O statements, nab I/O is done via calls to special builtin functions. These function calls have the same syntax as ordinary function calls but some of them have different semantics, in that they accept both a variable number of parameters and the parameters can be various types. nab uses the underlying C compiler's `printf()/scanf()` system to perform I/O on int, float and string objects. I/O on point is via their float `x`, `y` and `z` attributes. molecule I/O is covered in the next section, while bounds can be written using `dumpbounds()`. Transformation matrices can be written using `dumpmatrix()`, but there is currently no builtin for reading them. The value of an nab file object may be written by treating as an integer. Input to file variables is not defined.

15.11.1. Ordinary I/O Functions

nab provides these functions for stream or `FILE *` I/O of int, float and string objects.

15. NAB: Language Reference

<i>Inverse Trig Functions.</i>	
float acos(float x);	Return $\cos^{-1}(x)$ in degrees.
float asin(float x);	Return $\sin^{-1}(x)$ in degrees.
float atan(float x);	Return $\tan^{-1}(x)$ in degrees.
float atan2(float x);	Return $\tan^{-1}(y/x)$ in degrees. By keeping x and y separate, 90o can be returned without encountering a zero divide. Also, atan2 will return an angle in the full range [-180o, 180o].
<i>Trig Functions</i>	
float cos(float x);	Return $\cos(x)$, where x is in degrees.
float sin(float x);	Return $\sin(x)$, where x is in degrees.
float tan(float x);	Return $\tan(x)$, where x is in degrees.
<i>Conversion Functions.</i>	
float atof(string str);	Interpret the next run of non blank characters in str as a float and return its value. Return 0 on error.
int atoi(string str);	Interpret the next run of non blank characters in str as an int and return its value. Return 0 on error.
<i>Other Functions.</i>	
float rand2();	Return a pseudo-random number in (0,1).
float gauss(float mean, float sd);	Return a pseudo-random number taken from a Gaussian distribution with the given mean and standard deviation. The rand2() and gauss() routines share a common seed.
int setseed(int seed);	Reset the pseudo-random number sequence with the new seed, which must be a negative integer.
int rseed();	Use the system time() command to set the random number sequence with a reasonably random seed. Returns the seed it used; this could be used in a later call to setseed() to regenerate the same sequence of pseudo-random values.
float ceil(float x);	Return $\lceil x \rceil$.
float exp(float x);	Return e^x .
float cosh(float x);	Return the hyperbolic cosine of x.
float fabs(float x);	Return $ x $.
float floor(float x);	Return $\lfloor x \rfloor$.
float fmod(float x, float y);	Return r, the remainder of x with respect to y; the signs of r and y are the same.
float log(float x);	Return the natural logarithm of x.
float log10(float x);	Return the base 10 logarithm of x.
float pow(float x, float y);	Return x^y , $x > 0$.
float sinh(float x);	Return the hyperbolic sine of x.
float tanh(float x);	Return the hyperbolic tangent of x.
float sqrt(float x);	Return positive square root of x, $x \geq 0$.

Table 15.1.: NAB built-in mathematical functions

```

int fclose( file f );
file fopen( string fname, string mode );
int unlink( string fname );
int printf( string fmt, ... );
int fprintf( file f, string fmt, ... );
string sprintf( string fmt, ... );
int scanf( string fmt, ... );
int fscanf( file f, string fmt, ... );
int sscanf( string str, string fmt, ... );
string getline( file f );

```

`fclose()` closes (disconnects) the file represented by `f`. It returns 0 on success and -1 on failure. All open nab files are automatically closed when the program terminates. However, since the number of open files is limited, it is a good idea to close open files when they are no longer needed. The system call `unlink` removes (deletes) the file.

`fopen()` attempts to open (prepare for use) the file named `fname` with mode `mode`. It returns a valid nab file on success, and NULL on failure. Code should thus check for a return value of NULL, and do the appropriate thing. (An alternative, `safe_fopen()` sends an error message to `stderr` and exits on failure; this is sometimes a convenient alternative to `fopen()` itself, fitting with a general bias of nab system functions to exit on failure, rather than to return error codes that must always be processed.) Here are the most common values for mode and their meanings. For other values, consult any standard C reference.

fopen() mode values	
"r"	Open for reading. The file <code>fname</code> must exist and be readable by the user.
"w"	Open for writing. If the file exists and is writable by the user, truncate it to zero length. If the file does not exist, and if the directory in which it will exist is writable by the user, then create it.
"a"	Open for appending. The file must exist and be writable by the user.

The three functions `printf()`, `fprintf()` and `sprintf()` are for formatted (ASCII) output to `stdout`, the file `f` and a string. Strictly speaking, `sprintf()` does not perform output, but is discussed here because it acts as if "writes" to a string. Each of these functions uses the format string `fmt` to direct the conversion of the expressions that follow it in the parameter list. Format strings and expressions are discussed **Format Expressions**. The first format descriptor of `fmt` is used to convert the first expression after `fmt`, the second descriptor, the next expression etc. If there are more expressions than format descriptors, the extra expressions are not converted. If there are fewer expressions than format descriptors, the program will likely die when the function tries to convert non-existent data.

The three functions `scanf()`, `fscanf()` and `sscanf()` are for formatted (ASCII) input from `stdin`, the file `f` and the string `str`. Again, `sscanf()` does not perform input but the function behaves like it is "reading" from `str`. The action of these functions is similar to their output counterparts in that the format expression in `fmt` is used to direct the conversion of characters in the input and

store the results in the variables specified by the parameters following `fmt`. Format descriptors in `fmt` correspond to variables following `fmt`, with the first descriptor corresponding to the first variable, etc. If there are fewer descriptors than variables, then extra variables are not assigned; if there are more descriptors than variables, the program will most likely die due to a reference to a non-existent address.

There are two very important differences between `nab` formatted I/O and C formatted I/O. In C, formatted input is assigned through pointers to the variables (`&var`). In `nab` formatted I/O, the compiler automatically supplies the addresses of the variables to be assigned. The second difference is when a string object receives data during an `nab` formatted I/O. `nab` strings are allocated when needed. However, in the case of any kind of `scanf()` to a string or the implied (and hidden) writing to a string with `sprintf()`, the number of characters to be written to the string is unknown until the string has been written. `nab` automatically allocates strings of length 256 to hold such data with the idea that 256 is usually big enough. However, there will be cases where it is not big enough and this will cause the program to die or behave strangely as it will overwrite other data.

Also note that the default precision for floats in `nab` is double precision (see `$NABHOME/src/defreal.h`, since this could be changed, or may be different on your system.) Formats for floats for the `scanf` functions then need to be `"%lf"` rather than `"%f"`.

The `getline()` function returns a string that has the next line from file `f`. The end-of-line character has been stripped off.

15.11.2. matrix I/O

NAB uses 4x4 matrices to represent coordinate transformations:

```

r  r  r  0
r  r  r  0
r  r  r  0
dx dy dz 1

```

The `r`'s are a 3x3 rotation matrix, and the `d`'s are the translations along the X,Y and Z axes.

NAB coordinates are row vectors which are transformed by appending a 1 to each point (`x,y,z`) \rightarrow (`x,y,z,1`), post multiplying by the transformation matrix, and then discarding the final 1 in the new point.

Two builtins are provided for reading/writing transformation matrices.

```
matrix getmatrix(string filename);
```

Read the matrix from the file with name `filename`. Use `"-"` to read a matrix from `stdin`. A matrix is 4 lines of 4 numbers. A line of less than 4 numbers is an error, but anything after the 4th number is ignored. Lines beginning with a `'#'` are comments. Lines after the 4th data line are not read. Return a matrix with all zeroes on error, which can be tested:

```

mat = getmatrix("bad.mat");
if (!mat) { fprintf(stderr, "error reading matrix\n"); ... }

```

Keep in mind that `nab` transformations are intended for use on molecular coordinates, and that transformations like scaling and shearing [which can not be created with `nab` directly but can now be introduced via `getmatrix()`] may lead to incorrect or non-sensical results.

```
int putmatrix(string filename, matrix mat);
```

Write matrix `mat` to file with name `filename`. Use "-" to write a matrix to stdout. There is currently no way to write matrix to stderr. A matrix is written as 4 lines of 4 numbers. Return 0 on success and 1 on failure.

15.12. Molecule Creation Functions

The nab molecule type has a complex and dynamic internal structure organized in a three level hierarchy. A molecule contains zero or more named strands. Strand names are strings of any characters except white space and can not exceed 255 characters in length. Each strand in a molecule must have a unique name. Strands in different molecules may have the same name. A strand contains zero or more residues. Residues in each strand are numbered from 1. There is no upper limit on the number of residues a strand may contain. Residues have names, which need not be unique. However, the combination of *strand-name:res-num* is unique for every residue in a molecule. Finally residues contain one or more atoms. Each atom name in a residue should be distinct, although this is neither required nor checked by nab. nab uses the following functions to create and modify molecules.

```
molecule newmolecule();
molecule copymolecule( molecule mol );
int freemolecule( molecule mol );
int freeresidue( residue r );
int addstrand( molecule mol, string sname );
int addresidue( molecule mol, string sname, residue res );
int connectres( molecule mol, string sname, int res1, string aname1,
               int res2, string aname2 );
int mergestr( molecule mol1, string str1, string end1, molecule mol2, string str2, string end2 );
```

`newmolecule()` creates an "empty" molecule—one with no strands, residues or atoms. It returns NULL if it can not create it. `copymolecule()` makes a copy of an existing molecule and returns a NULL on failure. `freemolecule()` and `freeresidue()` are used to deallocate memory set aside for a molecule or residue. In most programs, these functions are usually not necessary, but should be used when a large number of molecules are being copied. Once a molecule has been created, `addstrand()` is used to add one or more named strands. Strands can be added at any to a molecule. There is no limit on the number of strands in a molecule. Strands can be added to molecules created by `getpdb()` or other functions as long as the strand names are unique. `addstrand()` returns 0 on success and 1 on failure. Finally `addresidue()` is used to add residues to a strand. The first residue is numbered 1 and subsequent residues are numbered 2, 3, etc. `addresidue()` also returns 0 on success and 1 on failure.

nab requires that users explicitly make all inter-residue bonds. `connectres()` makes a bond between two atoms of *different* residues of the strand with name `sname`. It returns 0 on success and 1 on failure. Atoms in different strands can not be bonded. The bonding between atoms in a residue is set by the residue library entry and can not be changed at runtime at the nab level.

The last function `mergestr()` is used to merge two strands of the same molecule or copy a strand of the second molecule into a strand of the first. The residues of a strand are ordered

from 1 to N , where N is the number of residues in that strand. `nab` imposes no chemical ordering on the residues in a strand. However, since the strands are generally ordered, there are four ways to combine the two strands. `mergestr()` uses the two values "first" and "last" to stand for residues 1 and N . The four combinations and their meanings are shown in the next table. In the table, `str1` has N residues and `str2` has M residues.

end1	end2	Action
first	first	The residues of <code>str2</code> are reversed and then inserted before those of <code>str1</code> : $M, \dots, 2, 1 : 1, 2, \dots, N$
first	last	The residues of <code>str2</code> are inserted before those of <code>str1</code> : $1, 2, \dots, M : 1, 2, \dots, N$
last	first	The residues of <code>str2</code> are inserted after those of <code>str1</code> : $1, 2, \dots, N : 1, 2, \dots, M$
last	last	The residues of <code>str2</code> are reversed and then inserted after those of <code>str1</code> : $1, 2, \dots, N : M, \dots, 2, 1$

15.13. Creating Biopolymers

```
molecule linkprot( string strandname, string seq, string reslib );
molecule link_na( string strandname, string seq, string reslib, string natype,
string opts );
```

Although many `nab` functions don't care what kind of molecule they operate on, many operations require molecules that are compatible with the Amber force field libraries (see Chapter 6). The best and most general way to do this is to use `tleap` commands, described in Chapter 8). The `linkprot()` and `link_na()` routines given here are limited commands that may sometimes be useful, and are included for backwards compatibility with earlier versions of NAB.

`linkprot()` takes a strand identifier and a sequence, and returns a molecule with this sequence. The molecule has an extended structure, so that the ϕ , ψ and ω angles are all 180°. The `reslib` input determines which residue library is used; if it is an empty string, the AMBER 94 all-atom library is used, with charged end groups at the N and C termini. All `nab` residue libraries are denoted by the suffix `.rlb` and LEaP residue libraries are denoted by the suffix `.lib`. If `reslib` is set to "nneut", "cneut" or "neut", then neutral groups will be used at the N-terminus, the C-terminus, or both, respectively.

The `seq` string should give the amino acids using the one-letter code with upper-case letters. Some non-standard names are: "H" for histidine with the proton on the δ position; "h" for histidine with the proton at the ϵ position; "3" for protonated histidine; "n" for an acetyl blocking group; "c" for an HNMe blocking group, "a" for an NH₂ group, and "w" for a water molecule. If the sequence contains one or more "|" characters, the molecule will consist of separate polypeptide strands broken at these positions.

The `link_na()` routine works much the same way for DNA and RNA, using an input residue library to build a single-strand with correct local geometry but arbitrary torsion angles connecting one residue to the next. `natype` is used to specify either DNA or RNA. If the `opts` string contains a "5", the 5' residue will be "capped" (a hydrogen will be attached to the O5' atom); if this string contains a "3" the O3' atom will be capped.

The newer (and generally recommended) way to generate biomolecules uses the *get_pdb_prm()* function described in Chapter 18.

15.14. Fiber Diffraction Duplexes in NAB

The primary function in NAB for creating Watson-Crick duplexes based on fibre-diffraction data is *fd_helix*:

molecule *fd_helix*(string *helix_type*, string *seq*, string *acid_type*);

fd_helix() takes as its arguments three strings - the helix type of the duplex, the sequence of one strand of the duplex, and the acid type (which is "dna" or "rna"). Available helix types are as follows:

Helix type options for <i>fd_helix()</i>	
<i>arna</i>	Right Handed A-RNA (Arnott)
<i>aprna</i>	Right Handed A'-RNA (Arnott)
<i>lbdna</i>	Right Handed B-DNA (Langridge)
<i>abdna</i>	Right Handed B-DNA (Arnott)
<i>sbdna</i>	Left Handed B-DNA (Sasisekharan)
<i>adna</i>	Right Handed A-DNA (Arnott)

The molecule returns contains a Watson-Crick double-stranded helix, with the helix axis along z. For a further explanation of the *fd_helix* code, please see the code comments in the source file *fd_helix.nab*.

References for the fibre-diffraction data:

1. Structures of synthetic polynucleotides in the A-RNA and A'-RNA conformations. X-ray diffraction analyses of the molecule conformations of (polyadenylic acid) and (polyinosinic acid).(polycytidylic acid). Arnott, S.; Hukins, D.W.L.; Dover, S.D.; Fuller, W.; Hodgson, A.R. *J.Mol. Biol.* (1973), 81(2), 107-22.
2. Left-handed DNA helices. Arnott, S; Chandrasekaran, R; Birdsall, D.L.; Leslie, A.G.W.; Ratliff, R.L. *Nature* (1980), 283(5749), 743-5.
3. Stereochemistry of nucleic acids and polynucleotides. Lakshimanarayanan, A.V.; Sasisekharan, V. *Biochim. Biophys. Acta* 204, 49-53.
4. Fuller, W., Wilkins, M.H.F., Wilson, H.R., Hamilton, L.D. and Arnott, S. (1965). *J. Mol. Biol.* 12, 60.
5. Arnott, S.; Campbell Smith, P.J.; Chandrasekaran, R. in *Handbook of Biochemistry and Molecular Biology, 3rd Edition. Nucleic Acids—Volume II*, Fasman, G.P., ed. (Cleveland: CRC Press, 1976), pp. 411-422.

15.15. Reduced Representation DNA Modeling Functions

nab provides several functions for creating the reduced representation models of DNA described by R. Tan and S. Harvey.[220] This model uses only 3 pseudo-atoms to represent a base pair. The pseudo atom named CE represents the helix axis, the atom named SI represents the position of the sugar-phosphate backbone on the sense strand and the atom named MA points into the major groove. The plane described by these three atoms (and a corresponding virtual atom that represents the anti sugar-phosphate backbone) represents quite nicely an all atom watson-crick base pair plane.

```
molecule dna3( int nbases, float roll, float tilt, float twist, float rise );
molecule dna3_to_allatom( molecule m_dna3, string seq, string aseq, string reslib, string natype );
molecule allatom_to_dna3( molecule m_allatom, string sense, string anti );
```

The function `dna3()` creates a reduced representation DNA structure. `dna3()` takes as parameters the number of bases `nbases`, and four helical parameters `roll`, `tilt`, `twist`, and `rise`.

`dna3_to_allatom()` makes an all-atom dna model from a `dna3` molecule as input. The molecule `m_dna3` is a `dna3` molecule, and the strings `seq` and `aseq` are the sense and anti sequences of the all-atom helix to be constructed. Obviously, the number of bases in the all-atom model should be the same as in the `dna3` model. If the string `aseq` is left blank (""), the sequence generated is the `wc_complement()` of the sense sequence. `reslib` names the residue library from which the all-atom model is to be constructed. If left blank, this will default to `all_nucleic94.lib`. The last parameter is either "dna" or "rna" and defaults to `dna` if left blank.

The `allatom_to_dna3()` function creates a `dna3` model from a double stranded all-atom helix. The function takes as parameters the input all-atom molecule `m_allatom`, the name of the sense strand in the all-atom molecule, `sense` and the name of the anti strand, `anti`.

15.16. Molecule I/O Functions

nab provides several functions for reading and writing molecule and residue objects.

```
residue getresidue( string rname, string rlib );
molecule getpdb( string fname [, string options ] );
molecule getcif( string fname, string blockId );
int putpdb( string fname, molecule mol [, string options ] );
int putcif( string fname, molecule mol );
int putbnd( string fname, molecule mol );
int putdist( string fname, molecule mol );
```

The function `getresidue()` returns a copy of the residue with name `rname` from the residue library named `rlib`. If it can not do so it returns the value `NULL`.

The function `getpdb()` converts the contents of the PDB file with name `fname` into an nab molecule. `getpdb()` creates bonds between any two atoms in the same residue if their distance is less than: 1.20 Å if either atom is a hydrogen, 2.20 Å if either atom is a sulfur, and 1.85 Å otherwise. Atoms in different residues are never bonded by `getpdb()`.

<i>keyword</i>	<i>meaning</i>
-pqr	Put charges and radii into the columns following the xyz coordinates.
-nobocc	Do not put occupancy and b-factor into the columns following the xyz coordinates. This is implied if <i>-pqr</i> is present, but may also be used to save space in the output file, or for compatibility with programs that do not work well if such data is present.
-brook	Convert atom and residue names to the conventions used in Brookhaven PDB (version 2) files. This often gives greater compatibility with other software that may expect these conventions to hold, but the conversion may not be what is desired in many cases. Also, put the first character of the atom name in column 78, a preliminary effort at identifying it as in the most recent PDB format. If the <i>-brook</i> flag is not present, no conversion of atom and residue names is made, and no id is in column 78.
-wwpdb	Same as the <i>-brook</i> option, except use the "wwPDB" (aka version 3) residue and atom naming scheme.
-nocid	Do not put the chain-id (see the description of <i>getpdb</i> , above) in the output (i.e., if this flag is present, the chain-id column will be blank). This can be useful when many water molecules are present.
-allcid	If set, create a chain ID for every strand in the molecule being written. Use the strand's name if it is an upper case letter, else use the next free upper case letter. Use a blank if no more upper case letters are available. Default is false.
-tr	Do not start numbering residues over again when a new chain is encountered, i.e., the residue numbers are consecutive across chains, as required by some force-field programs like Amber.

Table 15.2.: *Options for putpdb.*

getpdb() creates a new strand each time the chain id changes or if the chain id remains the same and a TER card is encountered. The strand name is the chain id if it is not blank and "*N*", where *N* is the number of that strand in the molecule beginning with 1. For example, a PDB file containing chain with no chain ID, followed by chain A, followed by another blank chain would have three strands with names "1", "A" and "3". *getpdb()* returns a molecule on success and NULL on failure.

The optional final argument to *getpdb* can be used for a variety of purposes, which are outlined in the table below.

The (experimental!) function *getcif* is like *getpdb*, but reads an mmCIF (macro-molecular crystallographic information file) formatted file, and extracts "atom-site" information from data block *blockID*. You will need to compile and install the *cifparse* library in order to use this.

The next group of builtins write various parts of the molecule *mol* to the file *fname*. All return 0 on success and 1 on failure. If *fname* exists and is writable, it is overwritten without warning. *putpdb()* writes the molecule *mol* into the PDB file *fname*. If the "resid" of a residue has been set (either by using *getpdb* to create the molecule, or by an explicit operation in an *nab* routine)

then columns 22-27 of the output pdb file will use it; otherwise, nab will assign a chain-id and residue number and use those. In this latter case, a molecule with a single strand will have a blank chain-id; if there is more than one strand, each strand is written as a separate chain with chain id "A" assigned to the first strand in mol, "B" to the second, etc. Options for putpdb are given in Table 15.2.

putbnd() writes the bonds of mol into frame. Each bond is a pair of integers on a line. The integers refer to atom records in the corresponding PDB-style file. putdist() writes the interatomic distances between all atoms of mol a_i, a_j where $i < j$, in this seven column format.

```
rnum1 rname1 aname1 rnum2 rname2 aname2 distance
```

15.17. Other Molecular Functions

```
matrix superimpose( molecule mol, string aex1, molecule r_mol, string aex2 );
int rmsd( molecule mol, string aex1, molecule r_mol, string aex2, float r );
float angle( molecule mol, string aex1, string aex2, string aex3 );
float anglep( point pt1, point pt2, point pt3 );
float torsion( molecule mol, string aex1, string aex2, string aex3, string aex4 );
float torsionp( point pt1, point pt2, point pt3, point pt4 );
float dist( molecule mol, string aex1, string aex2 );
float distp( point pt1, point pt2 );
int countmolatoms( molecule mol, string aex );
int sugarpuckeranal( molecule mol, int strandnum, int startres, int endres );
int helixanal( molecule mol );
int plane( molecule mol, string aex, float A, float B, float C );
float molsurf( molecule mol, string aex, float probe_rad );
```

superimpose() transforms molecule mol so that the root mean square deviation between corresponding atoms in mol and r_mol is minimized. The corresponding atoms are those selected by the atom expressions aex1 applied to mol and aex2 applied to r_mol. The atom expressions must select the same number of atoms in each molecule. No checking is done to insure that the atoms selected by the two atom expressions actually correspond. superimpose() returns the transformation matrix it found. rmsd() computes the root mean square deviation between the pairs of corresponding atoms selected by applying aex1 to mol and aex2 to r_mol and returns the value in r. The two atom expressions must select the same number of atoms. Again, it is the user's responsibility to insure the two atom expressions select corresponding atoms. rmsd() returns 0 on success and 1 on failure.

angle() and anglep() compute the angle in degrees between three points. angle() uses atoms expressions to determine the average coordinates of the sets. anglep() takes as an argument three explicit points. Similarly, torsion() and torsionp() compute a torsion angle in degrees defined by four points. torsion() uses atom expressions to specify the points. These atom expression match sets of atoms in mol. The points are defined by the average coordinates of the sets. torsionp() uses four explicit points. Both functions return 0 if the torsion angle is not defined.

`dist()` and `distp()` compute the distance in angstroms between two explicit atoms. `dist()` uses atom expressions to determine which atoms to include in the calculation. An atom expression which selects more than one atom results in the distance being calculated from the average coordinate of the selected atoms. `distp()` returns the distance between two explicit points. The function `countmolatoms()` returns the number of atoms selected by `aex` in `mol`.

`sugarpuckeranal()` is a function that reports the various torsion angles in a nucleic acid structure. `helixanal()` is an interactive helix analysis function based on the methods described by Babcock *et al.* [122]

The `plane()` routine takes an atom expression `aex` and calculates the least-squares plane and returns the answer in the form $z = Ax + By + C$. It returns the number of atoms used to calculate the plane.

The `molsurf()` routine is an NAB adaptation of Paul Beroza's program of the same name. It takes coordinates and radii of atoms matching the atom expression `aex` in the input molecule, and returns the molecular surface area (the area of the solvent-excluded surface), in square angstroms. To compute the solvent-accessible area, add the probe radius to each atom's radius (using a `for(a in m)` loop), and call `molsurf` with a zero value for `probe_rad`.

15.18. Debugging Functions

`nab` provides the following builtin functions that allow the user to write the contents of various `nab` objects to an ASCII file. The file must be opened for writing before any of these functions are called.

```
int dumpmatrix( file f, matrix mat );
int dumpbounds( file f, bounds b, int binary );
float dumpboundsviolations( file f, bounds b, int cutoff );
int dumpmolecule( file f, molecule mol, int dres, int datom, int dbond );
int dumpresidue( file f, residue res, int datom, int dbond );
int dumpatom( file f, residue res, int anum, int dbond );
int assert( condition );
int debug( expression(s) );
```

`dumpmatrix()` writes the 16 float values of `mat` to the file `f`. The matrix is written as four rows of four numbers. `dumpbounds()` writes the distance bounds information contained in `b` to the file `f` using this eight column format:

```
atom-number1 atom-number2 lower upper
```

If `binary` is set to a non-zero value, equivalent information is written in binary format, which can save disk-space, and is much faster to read back in on subsequent runs.

`dumpboundsviolations()` writes all the bounds violations in the `bounds` object that are more than `cutoff`, and returns the bounds violation energy. `dumpmolecule()` writes the contents of `mol` to the file `f`. If `dres` is 1, then detailed residue information will also be written. If `datom` or `dbond` is 1, then detailed atom and/or bond information will be written. `dumpresidue()` writes the contents of residue `res` to the file `f`. Again if `datom` or `dbond` is 1, detailed information about that residue's atoms and bonds will be written. Finally `dumpatom()` writes the contents of the

15. NAB: Language Reference

atom anum of residue res to the file f. If dbond is 1, bonding information about that atom is also written.

The `assert()` statement will evaluate the condition expression, and terminate (with an error message) if the expression is not true. Unlike the corresponding "C" language construct (which is a macro), code is generated at compile time to indicate both the file and line number where the assertion failed, and to parse the condition expression and print the values of subexpressions inside it. Hence, for a code fragment like:

```
i=20; MAX=17;  
assert( i < MAX );
```

the error message will provide the assertion that failed, its location in the code, and the current values of "i" and "MAX". If the `-noassert` flag is set at compile time, `assert` statements in the code are ignored.

The `debug()` statement will evaluate and print a comma-separated expression list along with the source file(s) and line number(s). Continuing the above example, the statement

```
debug( i, MAX );
```

would print the values of "i" and "MAX" to *stdout*, and continue execution. If the `-nodebug` flag is set at compile time, `debug` statements in the code are ignored.

15.19. Time and date routines

NAB incorporates a few interfaces to time and date routines:

```
string date();  
string timeofday();  
string ftime( string fmt );
```

The `date()` routine returns a string in the format "03/08/1999", and the `timeofday()` routine returns the current time as "13:45:00". If you need access to more sophisticated time and date functions, the `ftime()` routine is just a wrapper for the standard C routine `strftime`, where the format string is used to determine what is output; see standard C documentation for how this works.

15.20. Computational resource consumption functions

NAB has a small number of functions to provide information about computational resources used during the run:

```
int mme_timer();  
int mme_rism_max_memory();
```

`mme_timer()` provides tables of execution times for `mme` functions executed. It does not provide a complete summary nor does it include functions not in the `mme` family. It is, however, useful for identifying the most expensive routines. `mme_rism_max_memory()` reports the maximum amount of memory allocated during a 3D-RISM calculation.

16. NAB: Rigid-Body Transformations

This chapter describes NAB functions to create and manipulate molecules through a variety of rigid-body transformations. This capability, when combined with distance geometry (described in the next chapter) offers a powerful approach to many problems in initial structure generation.

16.1. Transformation Matrix Functions

nab uses 4×4 matrices to hold coordinate transformations. nab provides these functions to create transformation matrices.

```
matrix newtransform( float dx, float dy, float dz, float rx, float ry, float rz );  
matrix rot4( molecule mol, string aex1, string aex2, float ang );  
matrix rot4p( point p1, point p2, float angle );
```

newtransform() creates a 4×4 matrix that will rotate an object by rz degrees about the Z axis, ry degrees about the Y axis, rx degrees about the X axis and then translate the rotated object by dx, dy, dz along the X, Y and Z axes. All rotations and transformations are with respect to the standard X, Y and Z axes centered at (0,0,0). rot4() and rot4p() create transformation matrices that rotate an object about an arbitrary axis. The rotation amount is in degrees. rot4() uses two atom expressions to define an axis that goes from aex1 to aex2. If an atom expression matches more than one atom in mol, the average of the coordinates of the matched atoms are used. If an atom expression matches no atoms in mol, the zero matrix is returned. rot4p() uses explicit points instead of atom expressions to specify the axis. If p1 and p2 are the same, the zero matrix is returned.

16.2. Frame Functions

Every nab molecule has a “frame” which is three orthonormal vectors and their origin. The frame acts like a handle attached to the molecule allowing control over its movement. Two frames attached to different molecules allow for precise positioning of one molecule with respect to the other. These functions are used in frame creation and manipulation. All return 0 on success and 1 on failure.

```
int setframe( int use, molecule mol, string org, string xtail, string xhead,  
             string ytail, string yhead );  
int setframep( int use, molecule mol, point org, point xtail, point xhead,  
              point ytail, point yhead );  
int alignframe( molecule mol, molecule r_mol );
```

`setframe()` and `setframep()` create coordinate frames for molecule `mol` from an origin and two independent vectors. In `setframe()`, the origin and two vectors are specified by atom expressions. These atom expressions match sets of atoms in `mol`. The average coordinates of the selected sets are used to define the origin (`org`), an X-axis (`xtail` to `xhead`) and a Y-axis (`ytail` to `yhead`). The Z-axis is created as $X \times Y$. Since it is unlikely that the original X and Y axes are orthogonal, the parameter `use` specifies which of them is to be a real axis. If `use == 1`, then the specified X-axis is the real X-axis and Y is recreated from $Z \times X$. If `use == 2`, then the specified Y-axis is the real Y-axis and X is recreated from $Y \times Z$. `setframep()` works exactly the same way except the vectors and origin are specified as explicit points.

`alignframe()` transforms `mol` to superimpose its frame on the frame of `r_mol`. If `r_mol` is NULL, `alignframe()` transforms `mol` to superimpose its frame on the standard X,Y,Z directions centered at (0,0,0).

16.3. Functions for working with Atomic Coordinates

`nab` provides several functions for getting and setting user defined sets of molecular coordinates.

```
int setpoint( molecule mol, string aex, point pt );
int setxyz_from_mol( molecule mol, string aex, point pts[] );
int setxyzw_from_mol( molecule mol, string aex, float xyzw[] );
int setmol_from_xyz( molecule mol, string aex, point pts[] );
int setmol_from_xyzw( molecule mol, string aex, float xyzw[] );
int transformmol( matrix mat, molecule mol, string aex );
residue transformres( matrix mat, residue res, string aex );
```

`setpoint()` sets `pt` to the average value of the coordinates of all atoms selected by the atom expression `aex`. If no atoms were selected it returns 1, otherwise it returns a 0. `setxyz_from_mol()` copies the coordinates of all atoms selected by the atom expression `aex` to the point array `pt`. It returns the number of atoms selected. `setmol_from_xyz()` replaces the coordinates of the selected atoms from the values in `pt`. It returns the number of replaced coordinates. The routines `setxyzw_from_mol` and `setmol_from_xyzw` work in the same way, except that they use four-dimensional coordinates rather than three-dimensional sets.

`transformmol()` applies the transformation matrix `mat` to those atoms of `mol` that were selected by the atom expression `aex`. It returns the number of atoms selected. `transformres()` applies the transformation matrix `mat` to those atoms of `res` that were selected by the atom expression `aex` and returns a transformed *copy* of the input residue. It returns NULL if the operation failed.

16.4. Symmetry Functions

Here we describe a set of NAB routines that provide an interface for rigid-body transformations based on crystallographic, point-group, or other symmetries. These are primarily higher-level ways to creating and manipulating sets of transformation matrices corresponding to common types of symmetry operations.

16.4.1. Matrix Creation Functions

```

int MAT_cube( point pts[3], matrix mats[24] )
int MAT_ico( point pts[3], matrix mats[60] )
int MAT_octa( point pts[3], matrix mats[24] )
int MAT_tetra( point pts[3], matrix mats[12] )
int MAT_dihedral( point pts[3], int nfold, matrix mats[1] )
int MAT_cyclic( point pts[2], float ang, int cnt, matrix mats[1] )
int MAT_helix( point pts[2], float ang, float dst, int cnt, matrix mats[1] )
int MAT_orient( point pts[4], float angs[3], matrix mats[1] )
int MAT_rotate( point pts[2], float ang, matrix mats[1] )
int MAT_translate( point pts[2], float dst, matrix mats[1] )

```

These two groups of functions produce arrays of matrices that can be applied to objects to generate point group symmetries (first group) or useful transformations (second group). The operations are defined with respect to a center and a set of axes specified by the points in the array `pts[]`. Every function requires a center and one axis which are `pts[1]` and the vector `pts[1]→pts[2]`. The other two points (if required) define two additional directions: `pts[1]→pts[3]` and `pts[1]→pts[4]`. How these directions are used depends on the function.

The point groups generated by the functions `MAT_cube()`, `MAT_ico()`, `MAT_octa()` and `MAT_tetra()` have three internal 2-fold axes. While these 2-fold are orthogonal, the 2 directions specified by the three points in `pts[]` need only be independent (not parallel). The 2-fold axes are constructed in this fashion. Axis-1 is along the direction `pts[1]→pts[2]`. Axis-3 is along the vector `pts[1]→pts[2] × pts[1]→pts[3]` and axis-2 is recreated along the vector `axis-3 × axis-1`. Each of these four functions creates a fixed number of matrices.

Dihedral symmetry is generated by an *N*-fold rotation about an axis followed by a 2-fold rotation about a second axis orthogonal to the first axis. `MAT_dihedral()` produces matrices that generate this symmetry. The *N*-fold axis is `pts[0]→pts[1]` and the second axis is created by the same orthogonalization process described above. Unlike the previous point group functions the number of matrices created by `MAT_dihedral()` is not fixed but is equal to $2 \times nfold$.

`MAT_cyclic()` creates *cnt* matrices that produce uniform rotations about the axis `pts[1]→pts[2]`. The rotations are in multiples of the angle *ang* beginning with 0, and increasing by *ang* until *cnt* matrices have been created. *cnt* is required to be > 0 , but *ang* can be 0, in which case `MAT_cyclic` returns *cnt* copies of the identity matrix.

`MAT_helix()` creates *cnt* matrices that produce a uniform helical twist about the axis `pts[1]→pts[2]`. The rotations are in multiples of *ang* and the translations in multiples of *dst*. *cnt* must be > 0 , but either *ang* or *dst* or both may be zero. If *ang* is not 0, but *dst* is, `MAT_helix()` produces a uniform plane rotation and is equivalent to `MAT_cyclic()`. An *ang* of 0 and a non-zero *dst* produces matrices that generate a uniform translation along the axis. If both *ang* and *dst* are 0, the `MAT_helix()` creates *cnt* copies of the identity matrix.

The three functions `MAT_orient()`, `MAT_rotate()` and `MAT_translate()` are not really symmetry operations but are auxiliary operations that are useful for positioning the objects which are to be operated on by the true symmetry operators. Two of these functions `MAT_rotate()` and `MAT_translate()` produce a single matrix that either rotates or translates an object along the axis `pts[1]→pts[2]`. A zero *ang* or *dst* is acceptable in which case the function creates an identity

16. NAB: Rigid-Body Transformations

matrix. Except for a different user interface these two functions are equivalent to the nab builtins `rot4p()` and `tran4p()`.

`MAT_orient()` creates a matrix that rotates a object about the three axes `pts[1]→pts[2]`, `pts[1]→pts[3]` and `pts[1]→pts[4]`. The rotations are specified by the values of the array `angs[]`, with `ang[1]` the rotation about axis-1 etc. The rotations are applied in the order axis-3, axis-2, axis-1. The axes remained fixed throughout the operation and zero angle values are acceptable. If all three angles are zero, `MAT_orient()` creates an identity matrix.

16.4.2. Matrix I/O Functions

```
int MAT_fprint( file f, int nmats, matrix mats[1] )
int MAT_sprint( string str, int nmats, matrix mats[1] )
int MAT_fscan( file f, int smats, matrix mats[1] )
int MAT_sscan( string str, int smats, matrix mats[1] )
string MAT_getsyminfo()
```

This group of functions is used to read and write nab matrix variables. The two functions `MAT_fprint()` and `MAT_sprint()` write the the matrix to the file `f` or the string `str`. The number of matrices is specified by the parameter `nmats` and the matrices are passed in the array `mats[]`.

The two functions `MAT_fscan()` and `MAT_sscan()` read matrices from the file `f` or the string `str` into the array `mats[]`. The parameter `smats` is the size of the matrix array and if the source file or string contains more than `smats` only the first `smats` will be returned. These two functions return the number of matrices read unless there the number of matrices is greater than `smat` or the last matrix was incomplete in which case they return -1.

In order to understand the last function in this group, `MAT_getsyminfo()`, it is necessary to discuss both the internal structure the nab matrix type and one of its most important uses. The nab matrix type is used to hold transformation matrices. Although these are atomic objects at the nab level, they are actually 4×4 matrices where the first three elements of the fourth row are the X Y and Z components of the translation part of the transformation. The matrix print functions write each matrix as four lines of four numbers separated by a single space. Similarly the matrix read functions expect each matrix to be represented as four lines of four white space (any number of tabs and spaces) separated numbers. The print functions use `%13.6e` for each number in order to produce output with aligned columns, but the scan functions only require that each matrix be contained in four lines of four numbers each.

Most nab programs use matrix variables as intermediates in creating structures. The structures are then saved and the matrices disappear when the program exits. Recently nab was used to create a set of routines called a “symmetry server”. This is a set of nab programs that work together to create matrix streams that are used to assemble composite objects. In order to make it most general, the symmetry server produces only matrices leaving it to the user to apply them. Since these programs will be used to create hierarchies of symmetries or transformations we decided that the external representation (files or strings) of matrices would consist of two kinds of information — required lines of row values and optional lines beginning with the character `#` some of which are used to contain information that describes how these matrices were created.

`MAT_getsyminfo()` is used to extract this symmetry information from either a matrix file or a string that holds the contents of a matrix file. Each time the user calls `MAT_fscan()` or

`MAT_sscan()`, any symmetry information present in the source file or string is saved in private buffer. The previous contents of this buffer are overwritten and lost. `MAT_getsyminfo()` returns the contents of this buffer. If the buffer is empty, indicating no symmetry information was present in either the source file or string, `MAT_getsyminfo()` returns `NULL`.

16.5. Symmetry server programs

This section describes a set of nab programs that are used together to create composite objects described by a hierarchical nest of transformations. There are four programs for creating and operating on transformation matrices: `matgen`, `matmerge`, `matmul` and `matextract`, a program, `transform`, for transforming PDB or point files, and two programs, `tss_init` and `tss_next` for searching spaces defined by transformation hierarchies. In addition to these programs, all of this functionality is available directly at the nab level via the `MAT_` and `tss_` builtins described above.

16.5.1. matgen

The program `matgen` creates matrices that correspond to a symmetry or transformation operation. It has one required argument, the name of a file containing a description of this operation. The created matrices are written to `stdout`. A single `matgen` may be used by itself or two or more `matgen` programs may be connected in a pipeline producing nested symmetries.

`matgen -create sydef-1 | matgen symdef-2 | ... | matgen symdef-N`

Because a `matgen` can be in the middle of a pipeline, it automatically looks for an stream of matrices on `stdin`. This means the first `matgen` in a pipeline will wait for an EOF (generally Ctl-D) from the terminal unless connected to an empty file or equivalent. In order to avoid the nuisance of having to create an empty matrix stream the first `matgen` in a pipeline should use the `-create` flag which tells `matgen` to ignore `stdin`.

If input matrices are read, each input matrix left multiplies the first generated matrix, then the second etc. The table below shows the effect of a `matgen` performing a 2-fold rotation on an input stream of three matrices.

Input:	IM_1, IM_2, IM_3
Operation:	2-fold rotation: R_1, R_2
Output:	$IM_1 \times R_1, IM_2 \times R_1, IM_3 \times R_1, IM_1 \times R_2, IM_2 \times R_2, IM_3 \times R_2$

16.5.2. Symmetry Definition Files

Transformations are specified in text files containing several lines of keyword/value pairs. These lines define the operation, its associated axes and other parameters such as angles, a distance or count. Most keywords have a default value, although the operation, center and axes are always required. Keyword lines may be in any order. Blank lines and most lines starting with a sharp (#) are ignored. Lines beginning with `#S{`, `#S+` and `#S}` are structure comments that describe how the matrices were created. These lines are required to search the space defined by

16. NAB: Rigid-Body Transformations

the transformation hierarchy and their meaning and use is covered in the section on “Searching Transformation Spaces”. A complete list of keywords, their acceptable values and defaults is shown below.

Keyword	Default Value	Possible Values
symmetry	None	cube, cyclic, dihedral, dodeca, helix, ico, octa, tetra.
transform	None	orient, rotate, translate.
name	mPid	Any string of nonblank characters.
noid	false	true, false.
axestype	relative	absolute, relative.
center	None	Any three numbers separated by tabs or spaces.
axis, axis1	None	
axis2	None	
axis3	None	
angle,angle1	0	Any number.
angle2	0	
angle3	0	
dist	0	
count	1	Any integer.

axis and axis1 are synonyms as are angle and angle1.

The symmetry and transform keywords specify the operation. One or the other but not both must be specified.

The name keyword names a particular symmetry operation. The default name is m immediately followed by the process ID, eg m2286. name is used by the transformation space search routines tss_init and tss_next and is described later in the section “Searching Transformation Spaces”.

The noid keyword with value true suppresses generation of the identity matrix in symmetry operations. For example, the keywords below

```

symmetry cyclic
noid false
center 0 0 0
axis 0 0 1
count 3

```

produce three matrices which perform rotations of 0o, 120o and 240o about the Z-axis. If noid is true, only the two non-identity matrices are created. This option is useful in building objects with two or three orthogonal 2-fold axes and is discussed further in the example “Icosahedron from Rotations”. The default value of noid is false.

The axestype, center and axis* keywords defined the symmetry axes. The center and axis* keywords each require a point value which is three numbers separated by tabs or spaces. Numbers may integer or real and in fixed or exponential format. Internally all numbers are converted to nab type float which is actually double precision. No space is permitted between the minus sign of a negative number and the digits.

The interpretation of these points depends on the value of the keyword axestype. If it is absolute then the axes are defined as the vectors center→axis1, center→axis2 and center→axis3.

If it relative, then the axes are vectors whose directions are **O→axis1**, **O→axis2** and **O→axis3** with their origins at center. If the value of center is 0,0,0, then absolute and relative are equivalent. The default value axestype is relative; center and the axis* do not have defaults.

The angle keywords specify the rotation about the axes. angle1 is associated with axis1 etc. Note that angle and angle1 are synonyms. The angle is in degrees, with positive being in the counterclockwise direction as you sight from the axis point to the center point. Either an integer or real value is acceptable. No space is permitted between the minus sign of a negative number and its digits. All angle* keywords have a default value of 0.

The dist keyword specifies the translation along an axis. The positive direction is from center to axis. Either integer or real value is acceptable. No space is permitted between the minus sign of a negative number and its digits. The default value of dist is 0.

The count keyword is used in three related ways. For the cyclic value of the symmetry it specifies count matrices, each representing a rotation of 360/count. It also specifies the same rotations about the non 2-fold axis of dihedral symmetry. For helix symmetry, it indicates that count matrices should be created, each with a rotation of angle. In all cases the default value is 1.

This table shows which keywords are used and/or required for each type of operation.

symmetry	name	noid	axestype	center	axes	angles	dist	count
cube	mPid	false	relative	Required	1,2	-	-	-
cyclic	mPid	false	relative	Required	1	-	-	D=1
dihedral	mPid	false	relative	Required	1,2	-	-	D=1
dodeca	mPid	false	relative	Required	1,2	-	-	-
helix	mPid	false	relative	Required	1	1,D=0	D=0	D=1
ico	mPid	false	relative	Required	1,2	-	-	-
octa	mPid	false	relative	Required	1,2	-	-	-
tetra	mPid	false	relative	Required	1,2	-	-	-
transform	name	noid	axestype	center	axes	angles	dist	count
orient	mPid	-	relative	Required	All	All,D=0	-	-
rotate	mPid	-	relative	Required	1	1,D=0	-	-
translate	mPid	-	relative	Required	1	-	D=0	-

16.5.3. matmerge

The matmerge program combines 2-4 files of matrices into a single stream of matrices written to stdout. Input matrices are in files whose names are given on as arguments on the matmerge command line. For example, the command line below

matmerge A.mat B.mat C.mat

copies the matrices from A.mat to stdout, followed by those of B.mat and finally those of C.mat. Thus matmerge is similar to the Unix cat command. The difference is that while they are called matrix files, they can contain special comments that describe how the matrices they contain were created. When matrix files are merged, these comments must be collected and grouped so that they are kept together in any further matrix processing.

16.5.4. matmul

The matmul program takes two files of matrices, and creates a new stream of matrices formed by the pair wise product of the matrices in the input streams. The new matrices are written to stdout. If the number of matrices in the two input files differ, the last matrix of the shorter file is replicated and applied to all remaining matrices of the longer file. For example, if the file 3.mat has three matrices and the file 5.mat has five, then the command "matmul 3.mat 5.mat" would result in the third matrix of 3.mat multiplying the third, forth and fifth matrices of 5.mat.

16.5.5. matextract

The matextract is used to extract matrices from the matrix stream presented on stdin and writes them to stdout. Matrices are numbered from 1 to N, where N is the number of matrices in the input stream. The matrices are selected by giving their numbers as the arguments to the matextract command. Each argument is comma or space separated list of one or more ranges, where a range is either a number or two numbers separated by a dash (-). A range beginning with - starts with the first matrix and a range ending with - ends with the last matrix. The range - selects all matrices. Here are some examples.

Command	Action
matextract 2	Extract matrix number 2.
matextract 2,5	Extract matrices number 2 and 5.
matextract 2 5	Extract matrices number 2 and 5.
matextract 2-5	Extract matrices number 2 up to and including 5.
matextract -5	Extract matrices 1 to 5.
matextract 2-	Extract all matrices beginning with number 2.
matextract -	Extract all matrices.
matextract 2-4,7 13 15,19-	Extract matrices 2 to 4, 7, 13, 15 and all matrices numbered 19 or higher.

16.5.6. transform

The transform program applies matrices to an object creating a composite object. The matrices are read from stdin and the new object is written to stdout. transform takes one argument, the name of the file holding the object to be transformed. transform is limited to two types of objects, a molecule in PDB format, or a set of points in a text file, three space/tab separated numbers/line. The name of object file is preceded by a flag specifying its type.

Command	Action
transform -pdb X.pdb	Transform a PDB format file.
transform -point X.pts	Transform a set of points.

17. NAB: Distance Geometry

The second main element in NAB for the generation of initial structures is distance geometry. The next subsection gives a brief overview of the basic theory, and is followed by sections giving details about the implementation in NAB.

17.1. Metric Matrix Distance Geometry

A popular method for constructing initial structures that satisfy distance constraints is based on a metric matrix or "distance geometry" approach.[210, 221] If we consider describing a macromolecule in terms of the distances between atoms, it is clear that there are many constraints that these distances must satisfy, since for N atoms there are $N(N-1)/2$ distances but only $3N$ coordinates. General considerations for the conditions required to "embed" a set of interatomic distances into a realizable three-dimensional object forms the subject of distance geometry. The basic approach starts from the *metric matrix* that contains the scalar products of the vectors \mathbf{x}_i that give the positions of the atoms:

$$g_{ij} \equiv \mathbf{x}_i \cdot \mathbf{x}_j \quad (17.1)$$

These matrix elements can be expressed in terms of the distances d_{ij} :

$$g_{ij} = 2(d_{i0}^2 + d_{j0}^2 - d_{ij}^2) \quad (17.2)$$

If the origin ("0") is chosen at the centroid of the atoms, then it can be shown that distances from this point can be computed from the interatomic distances alone. A fundamental theorem of distance geometry states that a set of distances can correspond to a three-dimensional object only if the metric matrix \mathbf{g} is rank three, i.e., if it has three positive and $N-3$ zero eigenvalues. This is not a trivial theorem, but it may be made plausible by thinking of the eigenanalysis as a principal component analysis: all of the distance properties of the molecule should be describable in terms of three "components," which would be the x , y and z coordinates. If we denote the eigenvector matrix as \mathbf{w} and the eigenvalues λ , the metric matrix can be written in two ways:

$$g_{ij} = \sum_{k=1}^3 x_{ik} x_{jk} = \sum_{k=1}^3 w_{ik} w_{jk} \lambda_k \quad (17.3)$$

The first equality follows from the definition of the metric tensor, Eq. (1); the upper limit of three in the second summation reflects the fact that a rank three matrix has only three non-zero eigenvalues. Eq. (3) then provides an expression for the coordinates \mathbf{x}_i in terms of the eigenvalues and eigenvectors of the metric matrix:

$$x_{ik} = \lambda_k^{1/2} w_{ik} \quad (17.4)$$

If the input distances are not exact, then in general the metric matrix will have more than three non-zero eigenvalues, but an approximate scheme can be made by using Eq. (4) with the three largest eigenvalues. Since information is lost by discarding the remaining eigenvectors, the resulting distances will not agree with the input distances, but will approximate them in a certain optimal fashion. A further "refinement" of these structures in three-dimensional space can then be used to improve agreement with the input distances.

In practice, even approximate distances are not known for most atom pairs; rather, one can set upper and lower bounds on acceptable distances, based on the covalent structure of the protein and on the observed NOE cross peaks. Then particular instances can be generated by choosing (often randomly) distances between the upper and lower bounds, and embedding the resulting metric matrix.

Considerable attention has been paid recently to improving the performance of distance geometry by examining the ways in which the bounds are "smoothed" and by which distances are selected between the bounds.[222, 223] The use of triangle bound inequalities to improve consistency among the bounds has been used for many years, and NAB implements the "random pairwise metrization" algorithm developed by Jay Ponder.[212] Methods like these are important especially for underconstrained problems, where a goal is to generate a reasonably random distribution of acceptable structures, and the difference between individual members of the ensemble may be quite large.

An alternative procedure, which we call "random embedding", implements the procedure of deGroot *et al.* for satisfying distance constraints.[224] This does not use the embedding idea discussed above, but rather randomly corrects individual distances, ignoring all couplings between distances. Doing this a great many times turns out to actually find fairly good structures in many cases, although the properties of the ensembles generated for underconstrained problems are not well understood. A similar idea has been developed by Agrafiotis,[225] and we have adopted a version of his "learning parameter" strategy into our implementation.

Although results undoubtedly depend upon the nature of the problem and the constraints, in many (most?) cases, randomized embedding will be both faster and better than the metric matrix strategy. Given its speed, randomized embedding should generally be tried first.

17.2. Creating and manipulating bounds, embedding structures

A variety of metric-matrix distance geometry routines are included as builtins in nab.

```

bounds newbounds( molecule mol, string opts );
int andbounds( bounds b, molecule mol, string aex1, string aex2,
              float lb, float ub );
int orbounds( bounds b, molecule mol, string aex1, string aex2,
              float lb, float ub );
int setbounds( bounds b, molecule mol, string aex1, string aex2,
              float lb, float ub );
int showbounds( bounds b, molecule mol, string aex1, string aex2 );
int useboundsfrom( bounds b, molecule mol1, string aex1, molecule mol2,

```


17.2. Creating and manipulating bounds, embedding structures

Option	type	Default	Action
-rbm	string	None	The value of the option is the name of a file containing the bounds matrix for this molecule. This file would ordinarily be made by the dump-bounds command.
-binary			If this flag is present, bounds read in with the <i>-rbm</i> will expect a binary file created by the dumpbounds command.
-nocov			If this flag is present, no covalent (bonding) information will be used in constructing the bounds matrix.
-nchi	int	4	The option containing the keyword <i>nchi</i> allocates <i>n</i> extra chiral atoms for each residue of this molecule. This allows for additional chirality information to be provided by the user. The default is 4 extra chiral atoms per residue.

Table 17.1.: *Options to newbounds.*

```

    string aex2, float deviation );
int setboundsfromdb( bounds b, molecule mol, string aex1, string aex2,
    string dbase, float mul );
int setchivol( bounds b, molecule mol, string aex1, string aex2, string aex3, string aex4, float vol );
int setchiplane( bounds b, molecule mol, string aex );
float getchivol( molecule mol, string aex1, string aex2, string aex3, string aex4 );
float getchivolp( point p1, point p2, point p3, point p4 );
int tsmooth( bounds b, float delta );
int geodesics( bounds b );
int dg_options( bounds b, string opts );
int embed( bounds b, float xyz[] );

```

The call to `newbounds()` is necessary to establish a bounds matrix for further work. This routine sets lower bounds to van der Waals limits, along with bounds derived from the input geometry for atoms bonded to each other, and for atoms bonded to a common atoms (i.e., so-called 1-2 and 1-3 interactions.) Upper and lower bounds for 1-4 interactions are set to the maximum and minimum possibilities (the *max* (*syn* , "van der Waals limits") and *anti* distances). `newbounds()` has a string as its last parameter. This string is used to pass in options that control the details of how those routines execute. The string can be NULL, "" or contain one or more *options* surrounded by white space. The formats of an option are

```

-name=value
-name to select the default value if it exists.

```

The options to `newbounds()` are listed in Table 17.1.

The next five routines use atom expressions `aex1` and `aex2` to select two sets of atoms. Each of these four routines returns the number of bounds set or changed. For each pair of atoms (*a1* in `aex1` and *a2* in `aex2`) and `bounds()` sets the lower bound to `max (current_lb, lb)` and the upper bound to the `min (current_ub, ub)`. If `ub < current_lb` or if `lb > current_ub`, the bounds for that pair are unchanged. The routine `orbounds()` works in a similar fashion, except that it uses the less restrictive of the two sets of bounds, rather than the more restrictive one.

17. NAB: Distance Geometry

The `setbounds()` call updates the bounds, overwriting whatever was there. `showbounds()` prints all the bounds between the atoms selected in the first atom expression and those selected in the second atom expression. The `useboundsfrom()` routine sets the the bounds between all the selected atoms in *mol1* according to the geometry of a reference molecule, *mol2*. The bounds are set between every pair of atoms selected in the first atom expression, *aex1* to the distance between the corresponding pair of atoms selected by *aex2* in the reference molecule. In addition, a slack term, *deviation*, is used to allow some variance from the reference geometry by decreasing the lower bound and increasing the upper bound between every pair of atoms selected. The amount of increase or decrease depends on the distance between the two atoms. Thus, a *deviation* of 0.25 will result in the lower bound set between two atoms to be 75% of the actual distance separating the corresponding two atoms selected in the reference molecule. Similarly, the upper bound between two atoms will be set to 125% of the actual distance separating the corresponding two atoms selected in the reference molecule. For instance, the call

```
useboundsfrom(b, mol1, "1:2:C1',N1", mref, "3:4:C1',N1", 0.10 );
```

sets the lower bound between the C1' and N1 atoms in strand 1, residue 2 of molecule *mol1* to 90% of the distance between the corresponding pair of atoms in strand 3, residue 4 of the reference molecule, *mref*. Similarly, the upper bound between the C1' and N1 atoms selected in *mol1* is set to 110% of the distance between the corresponding pair of atoms in *mref*. A *deviation* of 0.0 sets the upper and lower bounds between every pair of atoms selected to be the actual distance between the corresponding reference atoms. If *aex1* selects the same atoms as *aex2*, the bounds between those atoms selected will be constrained to the current geometry. Thus the call,

```
useboundsfrom(b, mol1, "1:1:", mol1, "1:1", 0.0 );
```

essentially constrains the current geometry of all the atoms in strand 1, residue 1, by setting the upper and lower bounds to the actual distances separating each atom pair. `useboundsfrom()` only checks the number of atoms selected by *aex1* and compares it to the number of atoms selected by *aex2*. If the number of atoms selected by both atom expressions are not equal, an error message is output. Note, however, that there is no checking on the atom types selected by either atom expression. Hence, it is important to understand the method in which nab atom expressions are evaluated. For more information, refer to Section 2.6, "Atom Names and Atom Expressions".

The `useboundsfrom()` function can also be used with distance geometry "templates", as discussed in the next subsection.

The routine `setchivol()` uses four atom expressions to select exactly four different atoms and sets the volume of the chiral (ordered) tetrahedron they describe to `vol`. Setting `vol` to 0 forces the four atoms to be planar. `setchivol()` returns 0 on success and 1 on failure. `setchivol()` does not affect any distance bounds in `b` and may precede or follow triangle smoothing.

Similar to `setchivol()`, `setchiplane()` enforces planarity across four or more atoms by setting the chiral volume to 0 for every quartet of atoms selected by *aex*. `setchiplane()` returns the number of quartets constrained. Note: If the number of chiral constraints set is larger than the default number of chiral objects allocated in the call to `newbounds()`, a chiral table overflow will

result. Thus, it may be necessary to allocate space for additional chiral objects by specifying a larger number for the option *nchi* in the call to `newbounds()`.

`getchivol()` takes as an argument four atom expressions and returns the chiral volume of the tetrahedron described by those atoms. If more than one atom is selected for a particular point, the atomic coordinate is calculated from the average of the atoms selected. Similarly, `getchivolp()` takes as an argument four parameters of type `point` and returns the chiral volume of the tetrahedron described by those points.

After bounds and chirality have been set in this way, the general approach would be to call `tsmooth()` to carry out triangle inequality smoothing, followed by `embed()` to create a three-dimensional object. This might then be refined against the distance bounds by a conjugate-gradient minimization routine. The `tsmooth()` routine takes two arguments: a bounds object, and a tolerance parameter *delta*, which is the amount by which an upper bound may exceed a lower bound without triggering a triangle error. For most circumstances, *delta* would be chosen as a small number, like 0.0005, to allow for modest round-off. In some circumstances, however, *delta* could be larger, to allow some significant inconsistencies in the bounds (in the hopes that the problems would be fixed in subsequent refinement steps.) If the `tsmooth()` routine detects a violation, it will (arbitrarily) adjust the upper bound to equal the lower bound. Ideally, one should fix the bounds inconsistencies before proceeding, but in some cases this fix will allow the refinements to proceed even when the underlying cause of the inconsistency is not corrected.

For larger systems, the `tsmooth()` routine becomes quite time-consuming as it scales $O(^3)$. In this case, a more efficient triangle smoothing routine, `geodesics()` is used. `geodesics()` smoothes the bounds matrix via the triangle inequality using a sparse matrix version of a shortest path algorithm.

The `embed` routine takes a bounds object as input, and returns a four-dimensional array of coordinates; (values of the 4-th coordinate may be nearly zero, depending on the value of *k4d*, see below.) Options for how the embed is done are passed in through the `dg_options` routine, whose option string has *name=value* pairs, separated by commas or whitespace. Allowed options are listed in the following table.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
<code>ddm</code>	none	Dump distance matrix to this file.
<code>rdm</code>	none	Instead of creating a distance matrix, read it from this file.
<code>dmm</code>	none	Dump the metric matrix to this file.
<code>rmm</code>	none	Instead of creating a metric matrix, read it from this file.
<code>gdist</code>	0	If set to non-zero value, use a Gaussian distribution for selecting distances; this will have a mean at the center of the allowed range, and a standard deviation equal to 1/4 of the range. If <code>gdist=0</code> , select distances from a uniform distribution in the allowed range.
<code>randpair</code>	0	Use random pair-wise metrization for this percentage of the distances, i.e., <code>randpair=10</code> . would metrize 10% of the distance pairs.
<code>eamax</code>	10	Maximum number of embed attempts before bailing out.
<code>seed</code>	-1	Initial seed for the random number generator.

17. NAB: Distance Geometry

keyword	default	meaning
pembed	0	If set to a non-zero value, use the "proximity embedding" scheme of de Groot <i>et al.</i> , [26] and Agrafiotis [27], rather than metric matrix embedding.
shuffle	1	Set to 1 to randomize coordinates inside a box of dimension <i>rbox</i> at the beginning of the <i>pembed</i> scheme; if 0, use whatever coordinates are fed to the routine.
rbox	20.0	Size, in angstroms, of each side of the cubic into which the coordinates are randomly created in the proximity-embed procedure, if <i>shuffle</i> is set.
riter	1000	Maximum number of cycles for random-embed procedure. Each cycle selects 1000 pairs for adjustment.
slearn	1.0	Starting value for the learning parameter in proximity embedding; see [27] for details.
kchi	1.0	Force constant for enforcement of chirality constraints.
k4d	1.0	Force constant for squeezing out the fourth dimensional coordinate. If this is non-zero, a penalty function will be added to the bounds-violation energy, which is equal to $0.5 * k4d * w * w$, where w is the value of the fourth dimensional coordinate.
sqviol	0	If set to non-zero value, use parabolas for the violation energy when upper or lower bounds are violated; otherwise use functions based on those in the <i>dgeom</i> program. See the code in <i>embed.c</i> for details.
lbpen	3.5	Weighting factor for lower-bounds violations, relative to upper-bounds violations. The default penalizes lower bounds 3.5 times as much as the equivalent upper-bounds violations, which is frequently appropriate distance geometry calculations on molecules.
ntrpr	10	Frequency at which the bounds matrix violations will be printed in subsequent refinements.
pencut	-1.0	If $pencut \geq 0.0$, individual distance and chirality violations greater than <i>pencut</i> will be printed out (along with the total energy) every <i>ntrpr</i> steps.

Typical calling sequences. The following segment shows some ways in which these routines can be put together to do some simple embeds:

```

1 molecule m;
2 bounds b;
3 float fret, xyz[ 10000 ];
4 int ier;
5
6 m = getpdb( argv[2] );
7 b = newbounds( m, "" );

```

```

8  tsmooth( b, 0.0005 );
9
10 dg_options( b, "gdist=1, ntp=50, k4d=2.0, randpair=10." );
11 embed( b, xyz );
12 ier = conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 200 );
13 printf( "conjgrad returns %d\\n", ier );
14
15 setmol_from_xyzw( m, NULL, xyz );
16 putpdb( "new.pdb", m );

```

In lines 6-8, the molecule is created by reading in a pdb file, then bounds are created and smoothed for it. The embed options (established in line 10) include 10% random pairwise metrization, use of Gaussian distance selection, squeezing out the 4-th dimension with a force constant of 2.0, and printing every 50 steps. The coordinates developed in the *embed* step (line 11) are passed to a conjugate gradient minimizer (see the description below), which will minimize for 200 steps, using the bounds-violation routine *db_viol* as the target function. Finally, in lines 15-16, the *setmol_from_xyzw* routine is used to put the coordinates from the *xyz* array back into the molecule, and a new pdb file is written.

More complex and representative examples of distance geometry are given in the **Examples** chapter below.

17.3. Distance geometry templates

The *useboundsfrom()* function can be used with structures supplied by the user, or by canonical structures supplied with the *nab* distribution called "templates". These templates include stacking schemes for all standard residues in a A-DNA, B-DNA, C-DNA, D-DNA, T-DNA, Z-DNA, A-RNA, or A'-RNA stack. Also included are the 28 possible basepairing schemes as described in Saenger.[226] The templates are in PDB format and are located in *\$NABHOME/dgdb/basepairs/* and *\$NABHOME/dgdb/stacking/*.

A typical use of these templates would be to set the bounds between two residues to some percentage of the idealized distance described by the template. In this case, the template would be the reference molecule (the second molecule passed to the function). A typical call might be:

```

useboundsfrom(b, m, "1:2,3:??,H?",
getpdb( PATH + "gc.bdna.pdb" ), "?:??,H?", 0.1 );

```

where *PATH* is *\$AMBERHOME/dat/dgdb/stacking/*. This call sets the bounds of all the base atoms in residues 2 (GUA) and 3 (CYT) of strand 1 to be within 10% of the distances found in the template.

The basepair templates are named so that the first field of the template name is the one-character initials of the two individual residues and the next field is the Roman numeral corresponding to same bonding scheme described by Sanger, p. 120. *Note: since no specific sugar or backbone conformation is assumed in the templates, the non-base atoms should not be referenced.* The base atoms of the templates are show in figures 17.1 and 17.2.

17. NAB: Distance Geometry

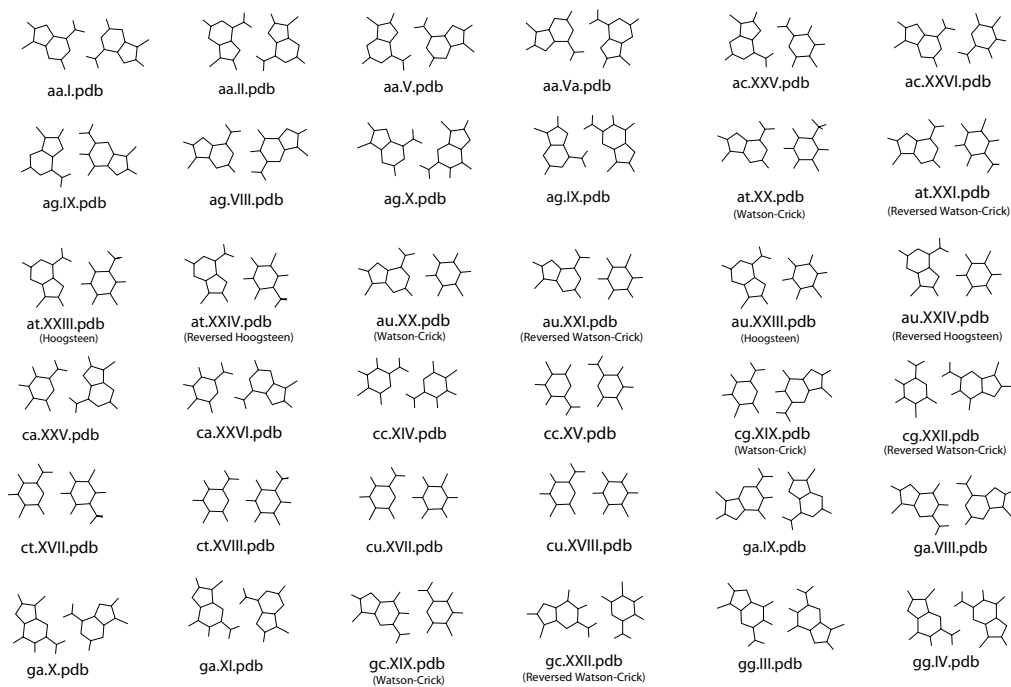


Figure 17.1.: Basepair templates for use with `useboundsfrom()`, (aa-gg)

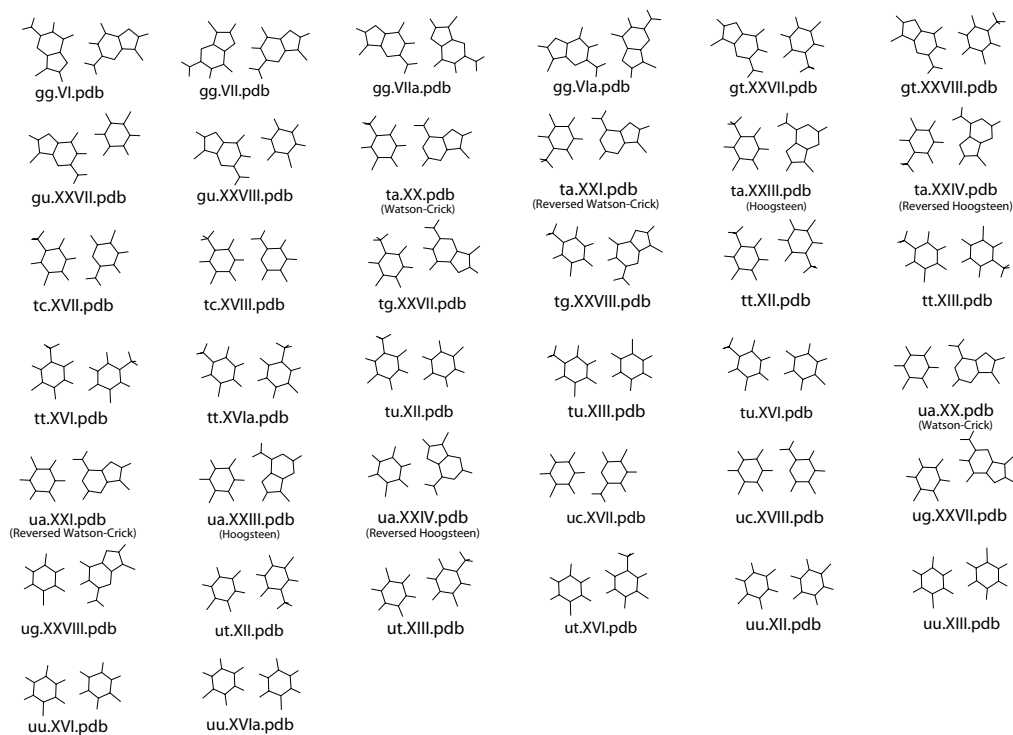


Figure 17.2.: Basepair templates for use with `useboundsfrom()`, (gg-uu)

The stacking templates are named in the same manner as the basepair templates. The first two letters of the template name are the one-character initials of the two residues involved in the stacking scheme (5' residue, then 3' residue) and the second field is the actual helical pattern (*note: a-rna represents the helical parameters of a'rna*). The stacking schemes can be found in the \$AMBERHOME/dat/dgdb/stacking directory.

17.4. Bounds databases

In addition to canonical templates, it is also possible to specify bounds information from a database of known molecular structures. This provides the option to use data obtained from actual structures, rather than from an idealized, canonical conformation.

The function `setboundsfromdb()` sets the bounds of all pairs of atoms between the two residues selected by `aex1` and `aex2` to a statistically averaged distance calculated from known structures plus or minus a multiple of the standard deviation. The statistical information is kept in database files. Currently, there are three types of database files - Those containing bounds information between Watson-Crick basepairs, those containing bounds information between helically stacked residues, and those containing intra-residue bounds information for residues in any conformation. The standard deviation is multiplied by the parameter *mul* and subtracted from the average distance to determine the lower bound and similarly added to the average distance to determine the upper bound of all base-base atom distances. Base-backbone bounds, that is, bounds between pairs of atoms in which one atom is a base atom and the other atom is a backbone atom, are set to be looser than base-base atoms. Specifically, the lower bound between a base-backbone atom pair is set to the smallest measured distance of all the structures considered in creating the database. Similarly, the upper bound between a base-backbone atom pair is set to the largest measured distance of all the structures considered. Base-base, and base-sugar bounds are set in a similar manner. This was done to avoid imposing false constraints on the atomic bounds, since Watson-Crick basepairing and stacking does not preclude any specific backbone and sugar conformation. `setboundsfromdb()` first searches the current directory for *dbase* before checking the default database location, \$AMBERHOME/dat/dgdb.

Each entry in the database file has six fields: The atoms whose bounds are to be set, the number of separate structures sampled in constructing these statistics, the average distance between the two atoms, the standard deviation, the minimum measured distance, and the maximum measured distance. For example, the database `bdna.basepair.db` has the following sample entries:

```
A:C2-T:C1' 424 6.167 0.198 5.687 6.673
A:C2-T:C2 424 3.986 0.175 3.554 4.505
A:C2-T:C2' 424 7.255 0.304 5.967 7.944
A:C2-T:C3' 424 8.349 0.216 7.456 8.897
A:C2-T:C4 424 4.680 0.182 4.122 5.138
A:C2-T:C4' 424 8.222 0.248 7.493 8.800
A:C2-T:C5 424 5.924 0.168 5.414 6.413
A:C2-T:C5' 424 9.385 0.306 8.273 10.104
A:C2-T:C6 424 6.161 0.163 5.689 6.679
A:C2-T:C7 424 7.205 0.184 6.547 7.658
```


The first column identifies the atoms from the adenosine C2 atom to various thymidine atoms in a Watson-Crick basepair. The second column indicates that 424 structures were sampled in determining the next four columns: the average distance, the standard deviation, and the minimum and maximum distances.

The databases were constructed using the coordinates from all the known nucleic acid structures from the Nucleic Acid Database (NDB - <http://www.ndbserver.ebi.ac.uk:5700/NDB/>). If one wishes to remake the databases, the coordinates of all the NDB structures should be downloaded and kept in the \$NABHOME/coords directory. The databases are made by issuing the command \$AMBERHOME/dat/dgdb/make_databases *dblist* where *dblist* is a list of nucleic acid types (i.e., *bdna*, *arna*, *etc.*). If one wants to add new structures to the structure repository at \$NABHOME/coords, it is necessary to make sure that the first two letters of the pdb file identify the nucleic acid type. That is, all *bdna* pdb files must begin with *bd*.

The *nab* functions used to create the databases are located in \$AMBERHOME/dat/dgdb/functions. The stacking databases were constructed as follows: If two residues stacked 5' to 3' in a helix have fewer than ten inter-residue atom distances closer than 2.0 Å or larger than 9.0 Å, and if the normals between the base planes are less than 20.0°, the residues were considered stacked. The base plane is calculated as the normal to the N1-C4 and midpoint of the C2-N3 and N1-C4 vectors. The first atom expression given to *setboundsfromdb()* specifies the 5' residue and the second atom expression specifies the 3' residue. The source for this function is *getstackdist.nab*.

Similarly, the basepair databases were constructed by measuring the heavy atom distances of corresponding residues in a helix to check for hydrogen bonding. Specifically, if an A-U basepair has an N1-N3 distance of between 2.3 and 3.2 Å and a N6-O4 distance of between 2.3 and 3.3 Å, then the A-U basepair is considered a Watson-Crick basepair and is used in the database. A C-G basepair is considered Watson-Crick paired if the N3-N1 distance is between 2.3 and 3.3 Å, the N4-O6 distance is between 2.3 and 3.2 Å, and the O2-N2 distance is between 2.3 and 3.2 Å.

The nucleotide databases contain all the distance information between atoms in the same residue. No residues in the coordinates directory are excluded from this database. The intent was to allow the residues of this database to assume all possible conformations and ensure that a nucleotide residue would not be biased to a particular conformation.

For the basepair and stacking databases, setting the parameter *mul* to 1.0 results in lower bounds being set from the average database distance minus one standard deviation, and upper bounds as the average database distance plus one standard deviation, between base-base atoms. Base-backbone and base-sugar upper and lower bounds are set to the maximum and minimum measured database values, respectively. *Note, however, that a stacking multiple of 0.0 may not correspond to consistent bounds. A stacking multiple of 0.0 will probably have conflicting bounds information as the bounds information is derived from many different structures.*

The database types are named *nucleic_acid_type.database_type.db*, and can be found in the \$AMBERHOME/dat/dgdb directory.

18. NAB: Molecular mechanics and dynamics

The initial models created by rigid-body transformations or distance geometry are often in need of further refinement, and molecular mechanics and dynamics can often be useful here. nab has facilities to allow molecular mechanics and molecular dynamics calculations to be carried out. At present, this uses the AMBER program LEaP to set up the parameters and topology; the force field calculations and manipulations like minimization and dynamics are done by routines in the nab suite. A version of LEaP is included in the NAB distribution, and is accessed by the leap() discussed below. A later chapter gives a more detailed description.

18.1. Basic molecular mechanics routines

```
molecule getpdb_prm( string pdb-
file, string leaprc, string leap_cmd2, int savef );
int readparm( molecule m, string parmfile );
int mme_init( molecule mol, string aexp, string aexp2, point xyz_ref[], file f );
int mm_options( string opts );
float mme( point xyz[], point grad[], int iter );
float mme_rattle( point xyz[], point grad[], int iter );
int conjgrad( float x[], int n, float fret, float func(), float rmsgrad,
              float dfpred, int maxiter );
int md( int n, int maxstep, point xyz[], point f[], float v[], float func );
int getxv( string filename, int natom, float start_time, float x[], float v[] );
int putxv( string filename, string title, int natom, float start_time,
          float x[], float v[] );
void mm_set_checkpoint( string filename );
```

The getpdb_prm() is a lot like getpdb() itself, except that it creates a molecule (and the associated force field parameters) that can be used in subsequent molecular mechanics calculations. It is often adequate to convert an input PDB file into a NAB molecule. (If this routine fails, you may be able to fix things up by editing your input pdb file, and/or by modifying the *leaprc* or *leap_cmd2* strings; if this doesn't work you will have to run tleap by hand, create a prmtop file, and use readparm() to input it.)

The leaprc string is passed to LEaP, and identifies which parameter and force field libraries to load. Sample leaprc files are in \$AmberHOME/dat/leap/cmd, and there is no default. The leap_cmd2 string is interpreted after the molecule has been read in to a unit called "X". Typically, leap_cmd2 would modify the molecule, say by adding or removing bonds, etc. The final

parameter, `savef` will save the intermediate files if non-zero; otherwise, all intermediate files created will be removed. `getpdb_prm()` returns a molecule whose force field parameters are already populated, and hence is ready for further force-field manipulation.

`readparm` reads an AMBER parameter-topology file, created by `tleap` or with other AMBER programs, and sets up a data structure which we call a "parmstruct". This is part of the molecule, but is not directly accessible (yet) to `nab` programs. You would use this command as an alternative to `getpdb_prm()`. You need to be sure that the molecule used in the `readparm()` call has been created by calling `getpdb()` with a PDB file that has been created by `tleap` itself (i.e., that has exactly the Amber atoms in the correct order). As noted above, the `readparm()` routine is primarily intended for cases where `getpdb_prm()` fails (i.e., when you need to run `tleap` by hand).

`setxyz_from_mol()` copies the atomic coordinates of `mol` to the array `xyz`. `setmol_from_xyz()` replaces the atomic coordinates of `mol` with the contents of `xyz`. Both return the number of atoms copied with a 0 indicating an error occurred.

The `getxv()` and `putxv()` routines read and write non-periodic Amber-style restart files. Velocities are read if present.

The `getxyz()` and `putxyz()` routines are used in conjunction with the `mm_set_checkpoint()` routine to write checkpoint or restart files. The coordinates are written at higher precision than to an AMBER restart file, i.e., with sufficiently high precision to restart even a Newton-Raphson minimization where the error in coordinates may be on the order of 10^{-12} . The checkpoint files are written at iteration intervals that are specified by the `nchk` or `nchk2` parameters to the `mm_options()` routine (see below). The checkpoint file names are determined by the filename string that is passed to `mm_set_checkpoint()`. If filename contains one or more `%d` format specifiers, then the file name will be a modification of filename wherein the leftmost `%d` of filename is replaced by the iteration count. If filename contains no `%d` format specifier, then the file name will be filename with the iteration count appended on the right.

The `mme_init()` function must be called after `mm_options()` and before calls to `mme()`. It sets up parameters for future force field evaluations, and takes as input an `nab` molecule. The string `aexp` is an atom expression that indicates which atoms are to be allowed to move in minimization or dynamics: atoms that do not match `aexp` will have their positions in the gradient vector set to zero. A NULL atom expression will allow all atoms to move. The second string, `aexp2` identifies atoms whose positions are to be restrained to the positions in the array `xyz_ref`. The strength of this restraint will be given by the `wcons` variable set in `mm_options()`. A NULL value for `aexp2` will cause all atoms to be constrained. The last parameter to `mme_init()` is a file pointer for the output trajectory file. This should be NULL if no output file is desired. NAB writes trajectories in the *binpos* format, which can be read by *ptraj*, and either analyzed, or converted to another format.

`mm_options()` is used to set parameters, and must be called before `mme_init()`; if you change options through a call to `mm_options()` without a subsequent call to `mme_init()` you may get incorrect calculations with no error messages. Beware. The `opts` string contains keyword/value pairs of the form `keyword=value` separated by white space or commas. Allowed values are shown in the following table.

18.1. Basic molecular mechanics routines

<i>keyword</i>	<i>default</i>	<i>meaning</i>
<code>ntpr</code>	10	Frequency of printing of the energy and its components.
<code>e_debug</code>	0	If non-zero printout additional components of the energy.
<code>gb_debug</code>	0	If non-zero printout information about Born first derivatives.
<code>gb2_debug</code>	0	If non-zero printout information about Born second derivatives.
<code>nchk</code>	10000	Frequency of writing checkpoint file during first derivative calculation, i.e., in the <code>mme()</code> routine.
<code>nchk2</code>	10000	Frequency of writing checkpoint file during second derivative calculation, i.e., in the <code>mme2()</code> routine.
<code>nsnb</code>	25	Frequency at which the non-bonded list is updated.
<code>nscm</code>	0	If > 0, remove translational and rotational center-of-mass (COM) motion after every <code>nscm</code> steps. For Langevin dynamics (<code>gamma_ln</code> >0) without HCP (<code>hcp</code> =0), the position of the COM is reset to zero every <code>nscm</code> steps, but the velocities are not affected. With HCP (<code>hcp</code> >0) COM translation and rotation are also removed, with or without Langevin dynamics. It is strongly recommended that this option be used whenever HCP is used.
<code>cut</code>	8.0	Non-bonded cutoff, in angstroms. This parameter is ignored if <code>hcp</code> > 0.
<code>wcons</code>	0.0	Restraint weight for keeping atoms close to their positions in <code>xyz_ref</code> (see <code>mme_init</code>).
<code>dim</code>	3	Number of spatial dimensions; supported values are 3 and 4.
<code>k4d</code>	1.0	Force constant for squeezing out the fourth dimensional coordinate, if <code>dim</code> =4. If this is non-zero, a penalty function will be added to the bounds-violation energy, which is equal to $0.5 * k4d * w * w$, where w is the value of the fourth dimensional coordinate.
<code>dt</code>	0.001	Time step, ps.
<code>t</code>	0.0	Initial time, ps.
<code>rattle</code>	0	If set to 1, bond lengths will be constrained to their equilibrium values, for dynamics; default is not to include such constraints. Note: if you want to use rattle (effectively "shake") for minimization, you do not need to set this parameter; rather, pass the <code>mme_rattle()</code> function to <code>conjgrad()</code> .
<code>tautp</code>	999999.	Temperature coupling parameter, in ps. The time constant determines the strength of the weak-coupling ("Berendsen") temperature bath.[227] Set <code>tautp</code> to a very large value (e.g. 9999999.) in order to turn off coupling and revert to Newtonian dynamics. This variable only has an effect if <code>gamma_ln</code> remains at its default value of zero; if <code>gamma_ln</code> is not zero, Langevin dynamics is assumed, as discussed below.

18. NAB: Molecular mechanics and dynamics

<i>keyword</i>	<i>default</i>	<i>meaning</i>
gamma_ln	0.0	Collision frequency for Langevin dynamics, in ps^{-1} . Values in the range $2-5ps^{-1}$ often give acceptable temperature control, while allowing transitions to take place.[228] Values near $50ps^{-1}$ correspond to the collision frequency for liquid water, and may be useful if rough physical time scales for motion are desired. The so-called BBK integrator is used here.[229]
temp0	300.0	Target temperature, K.
vlimit	20.0	Maximum absolute value of any component of the velocity vector.
ntr_md	10	Printing frequency for dynamics information to stdout.
ntwx	0	Frequency for dumping coordinates to traj_file.
zerov	0	If non-zero, then the initial velocities will be set to zero.
tempi	0.0	If <i>zerov</i> =0 and <i>tempi</i> >0, then the initial velocities will be randomly chosen for this temperature. If both <i>zerov</i> and <i>tempi</i> are zero, the velocities passed into the md() function will be used as the initial velocities; this combination is useful to continue an existing trajectory.
genmass	10.0	The general mass to use for MD if individual masses are not read from a prmtop file; value in amu.
diel	C	Code for the dielectric model. "C" gives a dielectric constant of 1; "R" makes the dielectric constant equal to distance in angstroms; "RL" uses the sigmoidal function of Ramstein & Lavery, PNAS 85 , 7231 (1988); "RL94" is the same thing, but speeded up assuming one is using the Cornell <i>et al</i> force field; "R94" is a distance-dependent dielectric, again with speedups that assume the Cornell <i>et al.</i> force field.
dielc	1.0	This is the dielectric constant used for <i>non-GB</i> simulations. It is implemented in routine mme_init() by scaling all of the charges by sqrt(dielc). This means that you need to set this (if desired) in mm_options() before calling mme_init().
gb	0	If set to 0 then GB is off. Setting gb=1 turns on the Hawkins, Cramer, Truhlar (HCT) form of pairwise generalized Born model for solvation. See ref [65] for details of the implementation; this is equivalent to the <i>igb=1</i> option in Amber. Set diel to "C" if you use this option. Setting gb=2 turns on the Onufriev, Bashford, Case (OBC) variant of GB,[66, 230] with $\alpha=0.8$, $\beta=0.0$ and $\gamma=2.909$. This is equivalent to the <i>igb=2</i> option in <i>sander</i> . Setting gb=5 just changes the values of α , β and γ to 1.0, 0.8, and 4.85, respectively, corresponding to the <i>igb=5</i> option in <i>sander</i> .

<i>keyword</i>	<i>default</i>	<i>meaning</i>
rgbmax	999.0	A maximum value for considering pairs of atoms to contribute to the calculation of the effective Born radii. The default value means that there is effectively no cutoff. Calculations will be sped up by using smaller values, say around 15. Å or so. This parameter is ignored if hcp > 0.
gbsa	0	If set to 1, add a surface-area dependent energy equal to surfen*SASA, where surfen is discussed below, and SASA is an approximate surface area term. NAB uses the "LCPO" approximation developed by Weiser, Shenkin, and Still.[129]
surften	0.005	Surface tension (see <i>gbsa</i> , above) in kcal/mol/Å ² .
epsext	78.5	Exterior dielectric for generalized Born; interior dielectric is always 1.
kappa	0.0	Inverse of the Debye-Hueckel length, if gb is turned on, in Å ⁻¹ . This parameter is related to the ionic strength as $\kappa = [8\pi\beta I/\epsilon]^{1/2}$, where <i>I</i> is the ionic strength (same as the salt concentration for a 1-1 salt). For <i>T</i> =298.15 and ϵ =78.5, $\kappa = (0.10806I)^{1/2}$, where <i>I</i> is in [M].
ipb	0	Switch to compute electrostatic solvation free energy. If set to 0 then PBSA is off. This is equivalent to the ipb option in <i>pbsa</i> . Possible values: 0 , 1 , 2 , and 4 . See PBSA chapter for more information.
inp	2	Option to select different methods to compute non-polar solvation free energy. This is equivalent to the inp option in <i>pbsa</i> . Possible values: 0 , 1 , and 2 . See PBSA chapter for more information.
epsin	1.0	Sets the dielectric constant of the solute region. The solute region is defined to be the solvent excluded volume.
epsout	80.0	Sets the implicit solvent dielectric constant. The solvent region is defined to be the space not occupied the solute region. Thus, only two dielectric regions are allowed in the current release.
smoothopt	1	Instructs PB how to set up dielectric values for finite-difference grid edges that are located across the solute/solvent dielectric boundary.
istrng	0.0	Sets the ionic strength (in mM) for the PB equation.
radiopt	1	Option to set up atomic radii. This is equivalent to the radiopt option in <i>pbsa</i> . Possible values: 0 , and 1 . See PBSA chapter for more information.
dprob	1.4	Solvent probe radius for molecular surface used to define the dielectric boundary between solute and solvent. If set 0.0, it would be later assigned to the value of sprob.
iprob	2.0	Mobile ion probe radius for ion accessible surface used to define the Stern layer.

18. NAB: Molecular mechanics and dynamics

<i>keyword</i>	<i>default</i>	<i>meaning</i>
npbopt	0	Option to select the linear or the full nonlinear PB equation. = 0 Linear PB equation is solved. = 1 Nonlinear PB equation is solved.
solvopt	1	Option to select iterative solvers. This is equivalent to the solvopt option in <i>pbsa</i> . Possible values: 1, 2, 3, 4, 5 , and 6 . See PBSA chapter for more information.
accept	0.001	Sets the iteration convergence criterion (relative to the initial residue).
maxitn	100	Sets the maximum number of iterations for the finite difference solvers, default to 100.
fillratio	2.0	The ratio between the longest dimension of the rectangular finite-difference grid and that of the solute.
space	0.5	Sets the grid spacing for the finite difference solver.
nfocus	2	Set how many successive FD calculations will be used to perform an electrostatic focussing calculation on a molecule. Possible values: 1 and 2 .
fscale	8	Set the ratio between the coarse and fine grid spacings in an electrostatic focussing calculation.
bcopt	5	Boundary condition options. This is equivalent to the bcopt option in <i>pbsa</i> . Possible values: 1, 5, 6 , and 10 . See PBSA chapter for more information.
eneopt	2	Option to compute total electrostatic energy and forces. This is equivalent to the eneopt option in <i>pbsa</i> . Possible values: 1 , and 2 . See PBSA chapter for more information.
dbfopt	n/a	This keyword is phased out in this release.
frcopt	0	Option to compute and output electrostatic forces to a file named force.dat in the working directory. This is equivalent to the frcopt option in <i>pbsa</i> . Possible values: 0, 1, 2 , and 3 . See PBSA chapter for more information.
cutnb	0.0	Atom-based cutoff distance for van der Waals interactions, and pairwise Coulombic interactions when ENEOPT = 2. When ENEOPT = 1, this is the cutoff distance used for van der Waals interactions only.
sprob	0.557	Solvent probe radius for solvent accessible surface area (SASA) used to compute the dispersion term.
npbverb	0	This turns on verbose mode in PB when set to 1.
arcres	0.25	gives the resolution (in the unit of Å) of dots used to represent solvent accessible arcs.
maxarcdot	1500	1500 actually means automatically determine number of arc dots required for solvent accessible surface, might grow too large to fit machines with less available memory. Please assign it to 4000~7000 and see if it fits into your computers.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
npbgrid	1	How many step do pbsa wait to re-calculate the geometry in a simulation, npbgrid = 1 is required to do trajectory evaluation. npbgrid is recommended to be 100 if “conjgrad” is used.
irism	0	Use 3D-RISM. = 0 Off. = 1 On.
xvfile	n/a	.xv file which describes bulk solvent properties. Required for 3D-RISM calculations. Produced by rism1d.
guvfile	n/a	Root name for solute-solvent 3D pair distribution function, $G^{UV}(\mathbf{R})$. This will produce one file for each solvent atom type for each frame requested.
huvfile	n/a	Rootname for solute-solvent 3D total correlation function, $H^{UV}(\mathbf{R})$. This will produce one file for each solvent atom type for each frame requested.
cuvfile	n/a	Rootname for solute-solvent 3D total correlation function, $C^{UV}(\mathbf{R})$. This will produce one file for each solvent atom type for each frame requested.
quvfile	n/a	Rootname for solvent 3D charge density distribution [$e/\text{\AA}$]. This will produce one file with contributions from each solvent atom type for each frame requested.
chgdist	n/a	Rootname for solvent 3D charge distribution [e]. This will produce one file with contributions from each solvent atom type for each frame requested.
uuvfile	n/a	Rootname for solute-solvent 3D potential energy, $U^{UV}(\mathbf{R})$. This will produce one file for each solvent atom type for each frame requested.
asymptfile	n/a	Rootname for solute-solvent 3D long range real-space asymptotics for C and H . This will produce one file for C and H for each frame requested.
volfmt	dx	Output format for volumetric data. = dx DX format (see §9.7.6). = xyzv XYZV format (see §9.7.7).
closure	KH	Select closure approximation. = HNC Hyper-netted chain equation (HNC). = KH Kovalenko-Hirata (KH). = PSEn Partial series expansion of order n where “n” is a positive integer.

18. NAB: Molecular mechanics and dynamics

<i>keyword</i>	<i>default</i>	<i>meaning</i>
closureorder	1	(Deprecated) Order for PSE-n closure if closure is specified as “PSE” or “PSEN” (no integers).
solvcut	buffer	Cut-off distance for solvent-solute potential and force calculations. <code>solvcut</code> must be explicitly set if <code>buffer</code> < 0. For minimization it is recommended to not use a cut-off (e.g. <code>solvcut=9999</code>).
buffer	14	Minimum distance in Å between the solute and the edge of the solvent box. < 0 Use fixed box size (<code>ng3</code> and <code>solvbox</code>). >= 0 Buffer distance.
grdspc	0.5	Linear grid spacing in x-, y- and z-dimensions [Å]. May be specified as single number if all dimensions have the same value. E.g., ‘ <code>grdspc=0.5</code> ’ is equivalent to ‘ <code>grdspc=0.5,0.5,0.5</code> ’.
ng	n/a	Sets the number of grid points for a fixed size solvation box. May be specified as single integer if all dimensions have the same value. E.g., ‘ <code>ng=64</code> ’ is equivalent to ‘ <code>ng=64,64,64</code> ’.
solvbox	n/a	Sets the size in Å of the fixed size solvation box. May be specified as single number if all dimensions have the same value. E.g., ‘ <code>solvbox=32.0</code> ’ is equivalent to ‘ <code>solvbox=32.0,32.0,32.0</code> ’.
tolerance	1e-5	Maximum residual tolerance required for convergence. For minimization a tolerance of 1e-11 or lower is recommended.
mdiis_del	0.7	“Step size” in MDIIS.
mdiis_nvec	5	Number of vectors used by the MDIIS method. Higher values for this parameter can greatly increase memory requirements but may also accelerate convergence.
mdiis_method	2	Specify implementation of the MDIIS routine. = 0 Original reference implementation. = 1 BLAS optimized. = 2 BLAS and memory optimized.
maxstep	10000	Maximum number of iterations allowed to converge on a solution.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
npropagate	5	<p>Number of previous solutions propagated forward to create an initial guess for this solute atom configuration.</p> <p>= 0 Do not use any previous solutions</p> <p>= 1..5 Values greater than 0 but less than 4 or 5 will use less system memory but may introduce artifacts to the solution (e.g., energy drift).</p>
centering	1	<p>Controls how the solute is centered/re-centered in the solvent box.</p> <p>= -2 Center of geometry. Center on first step only.</p> <p>= -1 Center of mass. Center on first step only.</p> <p>= 0 No centering. Dangerous.</p> <p>= 1 Center of mass. Center on every step. Recommended for molecular dynamics.</p> <p>= 2 Center of geometry. Center on every step. Recommended for minimization.</p>
zerofrc	1	<p>Redistribute solvent forces across the solute such that the net solvation force on the solute is zero.</p> <p>= 0 Unmodified forces.</p> <p>= 1 Zero net force.</p>
apply_rism_force	1	<p>Calculate and use solvation forces from 3D-RISM. Not calculating these forces can save computation time and is useful for trajectory post-processing.</p> <p>= 0 Do not calculate forces.</p> <p>= 1 Calculate forces.</p>
ntwrism	0	<p>Indicates that solvent density grid should be written to file every <code>ntwrism</code> iterations.</p> <p>= -1 Write output for the zero iteration.</p> <p>= 0 No files written.</p> <p>>= 1 Output every <code>ntwrism</code> time steps.</p>

18. NAB: Molecular mechanics and dynamics

<i>keyword</i>	<i>default</i>	<i>meaning</i>
ntprism	0	Indicates that 3D-RISM thermodynamic output should be written to file every <code>ntprism</code> iterations. = -1 Write output for the zero iteration. = 0 No files written. >= 1 Output every <code>ntwism</code> time steps.
polardecomp	0	Decompose the solvation free energy into polar and non-polar contributions. This is only useful if <code>ntprism</code> \neq 0 and adds about 80% to the total calculation time. = 0 No decomposition. = 1 Decomposition is performed.
verbose	0	Indicates level of diagnostic detail about the calculation written to the log file. = 0 No output. = 1 Print the number of iterations required to converge. = 2 Print details for each iteration and information about what FCE is doing every <code>progress</code> iterations.
progress	1	Display progress of the 3D-RISM solution every <code>progress</code> iterations. 0 indicates this information will not be displayed. Only used if <code>verbose</code> > 1.
static_arrays	1	If set to 1, do not allocate dynamic arrays for each call to the <code>mme()</code> and <code>mme2()</code> functions. The default value of 1 reduces computation time by avoiding array allocation.
blocksize	8	The granularity with which loop iterations are assigned to OpenMP threads or MPI processes. For MPI, a <code>blocksize</code> as small as 1 results in better load balancing during parallel execution. For OpenMP, <code>blocksize</code> should not be smaller than the number of floating-point numbers that fit into one cache line in order to avoid performance degradation through 'false sharing'. For ScaLAPACK, the optimum <code>blocksize</code> is not know, although a value of 1 is probably too small.
hcp	0	Use the Hierarchical Charge Partitioning (HCP) method. 1 = 1-charge approximation; 2 = 2-charge approximation. See Section 18.5 for detailed instructions on using the HCP. It is strongly recommended that the NSCM option above be used whenever HCP is used.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
hcp_h1	15	HCP level 1 threshold distance. The recommended level 1 threshold distance for amino acids is 15 Å. For structures with nucleic acids the recommended level 1 threshold distance is 21 Å.

The `mme()` function takes a coordinate set and returns the energy in the function value and the gradient of the energy in `grad`. The input parameter *iter* is used to control printing (see the *ntr* variable) and non-bonded updates (see *nsnb*). The `mme_rattle()` function has the same interface, but constrains the bond lengths and returns a corrected gradient. If you want to minimize with constrained bond lengths, pass `mme_rattle` and not `mme` to the *conjgrad* routine.

The `conjgrad()` function will carry out conjugate gradient minimization of the function `func` that depends upon *n* parameters, whose initial values are in the `x` array. The function `func` must be of the form `func(x[], g[], iter)`, where `x` contains the input values, and the function value is returned through the function call, and its gradient with respect to `x` through the `g` array. The iteration number is passed through *iter*, which `func` can use for whatever purpose it wants; a typical use would just be to determine when to print results. The input parameter *dfpred* is the expected drop in the function value on the first iteration; generally only a rough estimate is needed. The minimization will proceed until *maxiter* steps have been performed, or until the root-mean-square of the components of the gradient is less than *rmsgrad*. The value of the function at the end of the minimization is returned in the variable *fret*. `conjgrad` can return a variety of exit codes:

<i>Return codes for conjgrad routine</i>	
>0	minimization converged; gives number of final iteration
-1	bad line search; probably an error in the relation of the function to its gradient (perhaps from round-off if you push too hard on the minimization).
-2	search direction was uphill
-3	exceeded the maximum number of iterations
-4	could not further reduce function value

Finally, the `md` function will run *maxstep* steps of molecular dynamics, using `func` as the force field (this would typically be set to a function like `mme`.) The number of dynamical variables is given as input parameter *n*: this would be 3 times the number of atoms for ordinary cases, but might be different for other force fields or functions. The arrays `x[]`, `f[]` and `v[]` hold the coordinates, gradient of the potential, and velocities, respectively, and are updated as the simulation progresses. The method of temperature regulation (if any) is specified by the variables *tautp* and *gamma_ln* that are set in `mm_options()`.

Note: In versions of NAB up to 4.5.2, there was an additional input variable to `md()` called *minv* that reserved space for the inverse of the masses of the particles; this has now been removed. This change is not backwards compatible: you must modify existing NAB scripts that call `md()` to remove this variable.

18.2. Typical calling sequences

The following segment shows some ways in which these routines can be put together to do some molecular mechanics and dynamics:

```

1 // carry out molecular mechanics minimization and some simple dynamics
2 molecule m, mi;
3 int ier;
4 float m_xyz[ dynamic ], f_xyz[ dynamic ], v[ dynamic ];
5 float dgrad, fret, dummy[2];
6
7 mi = bdna( "gcgc" );
8 putpdb( "temp.pdb", mi );
9 m = getpdb_prm( "temp.pdb", "leaprc.ff99SB", "", 0 );
10
11 allocate m_xyz[ 3*m.natoms ]; allocate f_xyz[ 3*m.natoms ];
12 allocate v[ 3*m.natoms ];
13 setxyz_from_mol( m, NULL, m_xyz );
14
15 mm_options( "cut=25.0, ntp=10, nsnb=999, gamma_ln=5.0" );
16 mme_init( m, NULL, "::ZZZ", dummy, NULL );
17 fret = mme( m_xyz, f_xyz, 1 );
18 printf( "Initial energy is %8.3f\n", fret );
19
20 dgrad = 0.1;
21 ier = conjgrad( m_xyz, 3*m.natoms, fret, mme, dgrad, 10.0, 100 );
22 setmol_from_xyz( m, NULL, m_xyz );
23 putpdb( "gcgc.min.pdb", m );
24
25 mm_options( "tautp=0.4, temp0=100.0, ntp_md=10, tempi=50." );
26 md( 3*m.natoms, 1000, m_xyz, f_xyz, v, mme );
27 setmol_from_xyz( m, NULL, m_xyz );
28 putpdb( "gcgc.md.pdb", m );

```

Line 7 creates an nab molecule; any nab creation method could be used here. Then a temporary pdb file is created, and this is used to generate a NAB molecule that can be used for force-field calculations (line 9). Lines 11-13 allocate some memory, and fill the coordinate array with the molecular position. Lines 15-17 initialize the force field routine, and call it once to get the initial energy. The atom expression "::ZZZ" will match no atoms, so that there will be no restraints on the atoms; hence the fourth argument to mme_init can just be a place-holder, since there are no reference positions for this example. Minimization takes place at line 21, which will call mme repeatedly, and which also arranges for its own printout of results. Finally, in lines 25-28, a short (1000-step) molecular dynamics run is made. Note the the initialization routine mme_init *must* be called before calling the evaluation routines mme or md.

Elaboration of the the above scheme is generally straightforward. For example, a simulated annealing run in which the target temperature is slowly reduced to zero could be written as successive calls to mm_options (setting the temp0 parameter) and md (to run a certain number of steps with the new target temperature.) Note also that routines other than mme could be sent

to `conjgrad` and `md`: any routine that takes the same three arguments and returns a float function value could be used. In particular, the routines `db_viol` (to get violations of distance bounds from a bounds matrix) or `mme4` (to compute molecular mechanics energies in four spatial dimensions) could be used here. Or, you can write your own `nab` routine to do this as well. For some examples, see the `gbrna`, `gbrna_long` and `rattle_md` programs in the `$AMBERHOME/AmberTools/test/nab` directory.

18.3. Second derivatives and normal modes

Russ Brown has contributed new codes that compute analytically the second derivatives of the Amber functions, including the generalized Born terms. This capability resides in the three functions described here.

```
float newton( float x[], int n, float fret, float func1(), float func2(), float rms,  
             float nradd, int maxiter );  
float nmode( float x[], int n, float func(), int eigp, int ntrun, float eta, float hrmax, int iosen );
```

These routines construct and manipulate a Hessian (second derivative matrix), allowing one (for now) to carry out Newton-Raphson minimization and normal mode calculations. The `mme2()` routine takes as input a $3 \times n_{\text{atom}}$ vector of coordinates `x[]`, and returns a gradient vector `g[]`, a Hessian matrix, stored columnwise in a $3 \times n_{\text{atom}} \times 3 \times n_{\text{atom}}$ vector `h[]`, and the masses of the system, in a vector `m[]` of length `natom`. The iteration variable `iter` is just used to control printing. At present, these routines only work for `gb = 0` or `1`.

Users cannot call `mme2()` directly, but will pass this as an argument to one of the next two routines.

The `newton()` routine takes a input coordinates `x[]` and a size parameter `n` (must be set to $3 \times n_{\text{atom}}$). It performs Newton-Raphson optimization until the root-mean-square of the gradient vector is less than `rms`, or until `maxiter` steps have been taken. For now, the input function `func1()` must be `mme()` and `func2()` must be `mme2()`. The value `nradd` will be added to the diagonal of the Hessian before the step equations are solved; this is generally set to zero, but can be set something else under particular circumstances, which we do not discuss here.^[231]

Generally, you only want to try Newton-Raphson minimization (which can be very expensive) after you have optimized structures with `conjgrad()` to an rms gradient of 10^{-3} or so. In most cases, it should only take a small number of iterations then to go down to an rms gradient of about 10^{-12} or so, which is somewhere near the precision limit.

Once a good minimum has been found, you can use the `nmode()` function to compute normal/Langevin modes and thermochemical parameters. The first three arguments are the same as for `newton()`, the next two integers give the number of eigenvectors to compute and the type of run, respectively. The last three arguments (only used for Langevin modes) are the viscosity in centipoise, the value for the hydrodynamic radius, and the type of hydrodynamic interactions. Several techniques are available for diagonalizing the Hessian depending on the number of modes required and the amount of memory available.

In all cases the modes are written to an Amber-compatible "vecs" file for normal modes or "lmodevecs" file for Langevin modes. There are currently no `nab` routines that use this format.

18. NAB: Molecular mechanics and dynamics

The Langevin modes will also generate an output file called "lmode" that can be read by the Amber module *lmanal*.

ntrun

- 0: The dsyev routine is used to diagonalize the Hessian
- 1: The dsyevd routine is used to diagonalize the Hessian
- 2: The ARPACK package (shift invert technique) is used to obtain a small number of eigenvalues
- 3: The Langevin modes are computed with the viscosity and hydrodynamic radius provided

hrrmax Hydrodynamic radius for the atom with largest area exposed to solvent. If a file named "expfile" is provided then the relative exposed areas are read from this file. If "expfile" is not present all atoms are assigned a hydrodynamic radius of hrrmax or 0.2 for the hydrogen atoms. The "expfile" can be generated with the ms (molecular surface) program.

ioseen

- 0: Stokes Law is used for the hydrodynamic interaction
- 1: Oseen interaction included
- 2: Rotne-Prager correction included

Here is a typical calling sequence:

```
1 molecule m;
2 float x[4000], fret;
3
4 m = getpdb_prm( "mymolecule.pdb" );
5 mm_options( "cut=999., ntp=50, nsnb=99999, diel=C, gb=1, dielc=1.0" );
6 mme_init( m, NULL, "Z", x, NULL );
7 setxyz_from_mol( m, NULL, x );
8
9 // conjugate gradient minimization
10 conjgrad(x, 3*m.natoms, fret, mme, 0.1, 0.001, 2000 );
11
12 // Newton-Raphson minimization\fp
13 mm_options( "ntp=1" );
14 newton( x, 3*m.natoms, fret, mme, mme2, 0.00000001, 0.0, 6 );
15
16 // get the normal modes:
17 nmode( x, 3*m.natoms, mme2, 0, 0, 0.0, 0.0, 0 );
```


18.4. Low-MODE (LMOD) optimization methods

István Kolossváry has contributed new functions, which implement the LMOD methods for minimization, conformational searching, and flexible docking.[232–235] The centerpiece of LMOD is a conformational search algorithm based on eigenvector following of low-frequency vibrational modes. It has been applied to a spectrum of computational chemistry domains including protein loop optimization and flexible active site docking. The search method is implemented without explicit computation of a Hessian matrix and utilizes the Arnoldi package (ARPACK, <http://www.caam.rice.edu/software/ARPACK/>) for computing the low-frequency modes. LMOD optimization can be thought of as an advanced minimization method. LMOD can not only energy minimize a molecular structure in the local sense, but can generate a series of very low energy conformations. The LMOD capability resides in a single, top-level calling function *lmod()*, which uses fast local minimization techniques, collectively termed XMIN that can also be accessed directly through the function *xmin()*.

18.4.1. LMOD conformational searching

The LMOD conformational search procedure is based on gentle, but very effective structural perturbations applied to molecular systems in order to explore their conformational space. LMOD perturbations are derived from low-frequency vibrational modes representing large-amplitude, concerted atomic movements. Unlike essential dynamics where such low modes are derived from long molecular dynamics simulations, LMOD calculates the modes directly and utilizes them to improve Monte Carlo sampling.

LMOD has been developed primarily for macromolecules, with its main focus on protein loop optimization. However, it can be applied to any kind of molecular system,s including complexes and flexible docking where it has found widespread use. The LMOD procedure starts with an initial molecular model, which is energy minimized. The minimized structure is then subjected to an ARPACK calculation to find a user-specified number of low-mode eigenvectors of the Hessian matrix. The Hessian matrix is never computed; ARPACK makes only implicit reference to it through its product with a series of vectors. Hv , where v is an arbitrary unit vector, is calculated via a finite-difference formula as follows,

$$Hv = [\nabla(x_{min} + h) - \nabla(x_{min})] / h \quad (18.1)$$

where x_{min} is the coordinate vector at the energy minimized conformation and h denotes machine precision. The computational cost of Eq. 1 requires a single gradient calculation at the energy minimum point and one additional gradient calculation for each new vector. Note that ∇x is never 0, because minimization is stopped at a finite gradient RMS, which is typically set to 0.1-1.0 kcal/mol-Å in most calculations.

The low-mode eigenvectors of the Hessian matrix are stored and can be re-used throughout the LMOD search. Note that although ARPACK is very fast in relative terms, a single ARPACK calculation may take up to a few hours on an absolute CPU time scale with a large protein structure. Therefore, it would be impractical to recalculate the low-mode eigenvectors for each new structure. Visual inspection of the low-frequency vibrational modes of different, randomly generated conformations of protein molecules showed very similar, collective motions clearly

suggesting that low-modes of one particular conformation were transferable to other conformations for LMOD use. This important finding implies that the time limiting factor in LMOD optimization, even for relatively small molecules, is energy minimization, not the eigenvector calculation. This is the reason for employing XMIN for local minimization instead of NAB's standard minimization techniques.

18.4.2. LMOD Procedure

Given the energy-minimized structure of an initial protein model, protein- ligand complex, or any other molecular system and its low-mode Hessian eigenvectors, LMOD proceeds as follows. For each of the first n low-modes repeat steps 1-3 until convergence:

1. Perturb the energy-minimized starting structure by moving along the i th ($i = 1-n$) Hessian eigenvector in either of the two opposite directions to a certain distance. The $3N$ -dimensional (N is equal to the number of atoms) travel distance along the eigenvector is scaled to move the fastest moving atom of the selected mode in 3-dimensional space to a randomly chosen distance between a user-specified minimum and maximum value.

Note: A single LMOD move inherently involves excessive bond stretching and bond angle bending in Cartesian space. Therefore the primarily torsional trajectory drawn by the low-modes of vibration on the PES is severely contaminated by this naive, linear approximation and, therefore, the actual Cartesian LMOD trajectory often misses its target by climbing walls rather than crossing over into neighboring valleys at not too high altitudes. The current implementation of LMOD employs a so-called ZIG-ZAG algorithm, which consists of a series of alternating short LMOD moves along the low-mode eigenvector (ZIG) followed by a few steps of minimization (ZAG), which has been found to relax excessive stretches and bends more than reversing the torsional move. Therefore, it is expected that such a ZIG- ZAG trajectory will eventually be dominated by concerted torsional movements and will carry the molecule over the energy barrier in a way that is not too different from finding a saddle point and crossing over into the next valley like passing through a mountain pass.

Barrier crossing check: The LMOD algorithm checks barrier crossing by evaluating the following criterion: IF the current endpoint of the zigzag trajectory is lower than the energy of the starting structure, OR, the endpoint is at least lower than it was in the previous ZIG-ZAG iteration step AND the molecule has also moved farther away from the starting structure in terms of all-atom superposition RMS than at the previous position THEN it is assumed that the LMOD ZIG-ZAG trajectory has crossed an energy barrier.

2. Energy-minimize the perturbed structure at the endpoint of the ZIG- ZAG trajectory.
3. Save the new minimum-energy structure and return to step 1. Note that LMOD saves only low-energy structures within a user-specified energy window above the then current global minimum of the ongoing search.

After exploring the modes of a single structure, LMOD goes on to the next starting structure, which is selected from the set of previously found low- energy structures. The selection is based on either the Metropolis criterion, or simply the than lowest energy structure is used. LMOD

<i>Parameter list for xmin()</i>		
<i>keyword</i>	<i>default</i>	<i>meaning</i>
func	N/A	The name of the function that computes the function value and gradient of the objective function to be minimized. <i>func()</i> must have the following argument list: <code>float func(float x[], float g[], int i)</code> where <code>x[]</code> is the vector of the iterate, <code>g[]</code> is the gradient and <code>i</code> is currently ignored except when <code>func = mme</code> where <code>i</code> is handled internally.
natm	N/A	Number of atoms. NOTE: if <code>func</code> is other than <code>mme</code> , <code>natm</code> is used to pass the total number of variables of the objective function to be minimized. However, <code>natm</code> retains its original meaning in case <code>func</code> is a user-defined energy function for 3-dimensional (molecular) structure optimization. Make sure that the meaning of <code>natm</code> is compatible with the setting of <code>mol_struct_opt</code> below.
x[]	N/A	Coordinate vector. User has to allocate memory in calling program and fill <code>x[]</code> with initial coordinates using, e.g., the <code>setxyz_from_mol</code> function (see sample program below). Array size = <code>3*natm</code> .
g[]	N/A	Gradient vector. User has to allocate memory in calling program. Array size = <code>3*natm</code> .
ene	N/A	On output, <code>ene</code> stores the minimized energy.
grms_out	N/A	On output, <code>grms_out</code> stores the gradient RMS achieved by XMIN.

Table 18.2.: Arguments for *xmin()*.

terminates when the user-defined number of steps has been completed or when the user-defined number of low-energy conformations has been collected.

Note that for flexible docking calculations LMOD applies explicit translations and rotations of the ligand(s) on top of the low-mode perturbations.

18.4.3. XMIN

```
float xmin( float func(), int natm, float x[], float g[],
           float ene, float grms_out, struct xmod_opt xo);
```

At a glance: The *xmin()* function minimizes the energy of a molecular structure with initial coordinates given in the `x[]` array. On output, *xmin()* returns the minimized energy as the function value and the coordinates in `x[]` will be updated to the minimum-energy conformation. The arguments to *xmin()* are described in Table 18.2; the parameters in the `xmin_opt` structure are described in Table 18.3; these should be preceded by "`xo.`", since they are members of an *xmod_opt* struct with that name; see the sample program below to see how this works.

There are three types of minimizers that can be used, specified by the *method* parameter:

method

- 1: PRCG Polak-Ribiere conjugate gradient method, similar to the *conjgrad()* function [237].

<i>Parameter list for xmin_opt</i>		
<i>keyword</i>	<i>default</i>	<i>meaning</i>
mol_struct_opt	1	1= 3-dimensional molecular structure optimization. Any other value means general function optimization.
maxiter	1000	Maximum number of iteration steps allowed for XMIN. A value of zero means single point energy calculation, no minimization.
grms_tol	0.05	Gradient RMS threshold below which XMIN should minimize the input structure.
method	3	Minimization algorithm. See text for description.
numdiff	1	Finite difference method used in TNCG for approximating the product of the Hessian matrix and some vector in the conjugate gradient iteration (the same approximation is used in LMOD, see Eq. 18.1 in section 18.4.1). 1= Forward difference. 2=Central difference.
m_lbfgs	3	Size of the L-BFGS memory used in either L-BFGS minimization or L-BFGS preconditioning for TNCG. The value zero turns off preconditioning. It usually makes little sense to set the value >10.
print_level	0	Amount of debugging printout. 0= No output. 1= Minimization details. 2= Minimization (including conjugate gradient iteration in case of TNCG) and line search details.
iter	N/A	Output parameter. The total number of iteration steps completed by XMIN.
xmin_time	N/A	Output parameter. CPU time in seconds used by XMIN.
ls_method	2	1= modified Armijo [236](not recommended, primarily used for testing). 2= Wolfe (after J. J. More' and D. J. Thuente).
ls_maxiter	20	Maximum number of line search steps per single minimization step.
ls_maxatmov	0.5	Maximum (co-ordinate) movement per degree of freedom allowed in line search, range > 0.
beta_armijo	0.5	Armijo beta parameter, range (0, 1). <i>Only change it if you know what you are doing.</i>
c_armijo	0.4	Armijo c parameter, range (0, 0.5). <i>Only change it if you know what you are doing.</i>
mu_armijo	1.0	Armijo mu parameter, range [0, 2). <i>Only change it if you know what you are doing.</i>
ftol_wolfe	0.0001	Wolfe ftol parameter, range (0, 0.5). <i>Only change it if you know what you are doing.</i>
gtol_wolfe	0.9	Wolfe gtol parameter, range (ftol_wolfe, 1). <i>Only change it if you know what you are doing.</i>
ls_iter	N/A	Output parameter. The total number of line search steps completed by XMIN.
error_flag	N/A	Output parameter. A non-zero value indicates an error. In case of an error XMIN will always print a descriptive error message.

Table 18.3.: Options for xmin_opt.

- 2: L-BFGS Limited-memory Broyden-Fletcher-Goldfarb-Shanno quasi-Newton algorithm [238]. L-BFGS is 2-3 times faster than PRCG mainly, because it requires significantly fewer line search steps than PRCG.
- 3: lbfgs-TNCG L-BFGS preconditioned truncated Newton conjugate gradient algorithm [237, 239]. Sophisticated technique that can minimize molecular structures to lower energy and gradient than PRCG and L-BFGS and requires an order of magnitude fewer minimization steps, but L-BFGS can sometimes be faster in terms of total CPU time.

NOTE: The *xmin* routine can be utilized for minimizing arbitrary, user-defined objective functions. The function must be defined in a user NAB program or in any other user library that is linked in. The name of the function is passed to *xmin()* via the *func* argument.

18.4.4. Sample XMIN program

The following sample program, which is based on the test program *txmin.nab*, reads a molecular structure from a PDB file, minimizes it, and saves the minimized structure in another PDB file.

```

1 // XMIN reverse communication external minimization package.
2 // Written by Istvan Kolosvary.
3
4 #include "xmin_opt.h"
5
6 // M A I N P R O G R A M to carry out XMIN minimization on a molecule:
7
8 struct xmin_opt xo;
9
10 molecule mol;
11 int natm;
12 float xyz[ dynamic ], grad[ dynamic ];
13 float energy, grms;
14 point dummy;
15
16 xmin_opt_init( xo ); // set up defaults (shown here)
17
18 // xo.mol_struct_opt = 1;
19 // xo.maxiter      = 1000;
20 // xo.grms_tol     = 0.05;
21 // xo.method       = 3;
22 // xo.numdiff      = 1;
23 // xo.m_lbfgs      = 3;
24 // xo.ls_method    = 2;
25 // xo.ls_maxiter   = 20;
26 // xo.maxatmov     = 0.5;
27 // xo.beta_armijo  = 0.5;
28 // xo.c_armijo     = 0.4;
29 // xo.mu_armijo    = 1.0;

```

18. NAB: Molecular mechanics and dynamics

```

30 //    xo.ftol_wolfe  = 0.0001;
31 //    xo.gtol_wolfe  = 0.9;
32 // xo.print_level    = 0;
33
34 xo.maxiter          = 10; // non-defaults are here
35 xo.grms_tol         = 0.001;
36 xo.method           = 3;
37 xo.ls_maxatmov      = 0.15;
38 xo.print_level      = 2;
39
40 mol = getpdb( "gbrna.pdb" );
41 readparm( mol, "gbrna.prmtop" );
42 natm = mol.natoms;
43 allocate xyz[ 3*natm ]; allocate grad[ 3*natm ];
44 setxyz_from_mol( mol, NULL, xyz );
45
46 mm_options( "ntpr=1, gb=1, kappa=0.10395, rgbmax=99., cut=99.0, diel=C ");
47 mme_init( mol, NULL, "::ZZZ", dummy, NULL );
48
49 energy = mme( xyz, grad, 0 );
50 energy = xmin( mme, natm, xyz, grad, energy, grms, xo );
51
52 // E N D   M A I N

```

The corresponding screen output should look similar to this. Note that this is fairly technical, debugging information; normally `print_level` is set to zero.

```

Reading parm file (gbrna.prmtop)
title:
PDB 5DNB, Dickerson decamer
old prmtop format => using old algorithm for GB parms
    mm_options:  ntp=99
    mm_options:  gb=1
    mm_options:  kappa=0.10395
    mm_options:  rgbmax=99.
    mm_options:  cut=99.0
    mm_options:  diel=C
  iter   Total    bad      vdW    elect.    cons.   genBorn   frms
ff:    0  -4107.50   906.22   -192.79   -137.96     0.00   -4682.97  1.93e+01

MIN:                                It=    0  E=   -4107.50 ( 19.289)
CG:   It=    3 ( 0.310) :-)
LS: step= 0.94735  it= 1  info= 1
MIN:                                It=    1  E=   -4423.34 ( 5.719)
CG:   It=    4 ( 0.499) :-)
LS: step= 0.91413  it= 1  info= 1

```

```

MIN:                                It=    2  E=   -4499.43 (  2.674)
CG:   It=    9 (  0.498)  :-)
LS: step= 0.86829  it= 1  info= 1
MIN:                                It=    3  E=   -4531.20 (  1.543)
CG:   It=    8 (  0.499)  :-)
LS: step= 0.95556  it= 1  info= 1
MIN:                                It=    4  E=   -4547.59 (  1.111)
CG:   It=    9 (  0.491)  :-)
LS: step= 0.77247  it= 1  info= 1
MIN:                                It=    5  E=   -4556.35 (  1.068)
CG:   It=    8 (  0.361)  :-)
LS: step= 0.75150  it= 1  info= 1
MIN:                                It=    6  E=   -4562.95 (  1.042)
CG:   It=    8 (  0.273)  :-)
LS: step= 0.79565  it= 1  info= 1
MIN:                                It=    7  E=   -4568.59 (  0.997)
CG:   It=    5 (  0.401)  :-)
LS: step= 0.86051  it= 1  info= 1
MIN:                                It=    8  E=   -4572.93 (  0.786)
CG:   It=    4 (  0.335)  :-)
LS: step= 0.88096  it= 1  info= 1
MIN:                                It=    9  E=   -4575.25 (  0.551)
CG:   It=   64 (  0.475)  :-)
LS: step= 0.95860  it= 1  info= 1
MIN:                                It=   10  E=   -4579.19 (  0.515)
-----
FIN:                                :-)                                E=   -4579.19 (  0.515)

```

The first few lines are typical NAB output from `mm_init()` and `mme()`. The output below the horizontal line comes from XMIN. The MIN/CG/LS blocks contain the following pieces of information. The MIN: line shows the current iteration count, energy and gradient RMS (in parentheses). The CG: line shows the CG iteration count and the residual in parentheses. The happy face :-) means convergence whereas :-(indicates that CG iteration encountered negative curvature and had to abort. The latter situation is not a serious problem, minimization can continue. This is just a safeguard against uphill moves. The LS: line shows line search information. "step" is the relative step with respect to the initial guess of the line search step. "it" tells the number of line search steps taken and "info" is an error code. "info" = 1 means that line searching converged with respect to sufficient decrease and curvature criteria whereas a non- zero value indicates an error condition. Again, an error in line searching doesn't mean that minimization necessarily failed, it just cannot proceed any further because of some numerical dead end. The FIN: line shows the final result with a happy face :-) if either the `grms_tol` criterion has been met or when the number of iteration steps reached the `maxiter` value.

18.4.5. LMOD

```
float lmod( int natm, float x[], float g[], float ene, float conflib[],
           float lmod_traj[], int lig_start[], int lig_end[], int lig_cent[],
           float tr_min[], float tr_max[], float rot_min[], float rot_max[],
           struct xmin_opt, struct xmin_opt, struct lmod_opt);
```

At a glance: The *lmod()* function is similar to *xmin()* in that it optimizes the energy of a molecular structure with initial coordinates given in the *x[]* array. However, the optimization goes beyond local minimization, it is a sophisticated conformational search procedure. On output, *lmod()* returns the global minimum energy of the LMOD conformational search as the function value and the coordinates in *x[]* will be updated to the global minimum-energy conformation. Moreover, a set of the best low-energy conformations is also returned in the array *conflib[]*. Coordinates, energy, and gradient are in NAB units. The parameters are given in the table below; items above the line are passed as parameters; the rest of the parameters are all preceded by "lo.", because they are members of an *lmod_opt* struct with that name; see the sample program below to see how this works.

Also note that *xmin()*'s *xmin_opt* struct is passed to *lmod()* as well. *lmod()* changes the default values of some of the "xo." parameters via the call to *lmod_opt_init()* relative to a call to *xmin_opt_init()*, which means that in a more complex NAB program with multiple calls to *xmin()* and *lmod()*; make sure to always initialize and set user parameters for each and every XMIN and LMOD search via, respectively calling *xmin_opt_init()* and *lmod_opt_init()* just before the calls to *xmin()* and *lmod()*.

keyword	default	meaning
natm	N/A	Number of atoms.
x[]		Coordinate vector. User has to allocate memory in calling program and fill x[] with initial coordinates using, e.g., the <i>setxyz_from_mol</i> function (see sample program below). Array size = 3*natm.
g[]		Gradient vector. User has to allocate memory in calling program. Array size = 3*natm.
ene		On output, ene stores the global minimum energy.
conflib[]		User allocated storage array where LMOD stores low-energy conformations. Array size = 3*natm*nconf.
lmod_traj[]		User allocated storage array where LMOD stores snapshots of the pseudo trajectory drawn by LMOD on the potential energy surface. Array size = 3*natom * (nconf + 1).
lig_start[]	N/A	The serial number(s) of the first/last atom(s) of the ligand(s). The number(s) should correspond to the numbering in the NAB input files. Note that the ligand(s) can be anywhere in the atom list, however, a single ligand must have continuous numbering between the corresponding <i>lig_start</i> and <i>lig_end</i> values. The arrays should be allocated in the calling program. Array size = nlig, but in case nlig=0 there is no need for allocating memory.

18.4. Low-MODE (LMOD) optimization methods

<i>keyword</i>	<i>default</i>	<i>meaning</i>
lig_end[] lig_cent[]	N/A N/A	See above. Similar array in all respects to lig_start/end, but the serial number(s) define the center of rotation. The value zero means that the center of rotation will be the geometric center of gravity of the ligand.
tr_min[]	N/A	The range of random translation/rotation applied to individual ligand(s). Rotation is carried out about the origin defined by the corresponding lig_cent value(s). The angle is given in +/- degrees and the distance in angstroms. The particular angles and distances are randomly chosen from their respective ranges. The arrays should be allocated in the calling program. Array size = <i>nlig</i> , but in case <i>nlig</i> =0 there is no need to allocate memory.
tr_max[] rot_min[] rot_max[]		See tr_min[], above. See tr_min[], above. See tr_min[], above.
niter	10	The number of LMOD iterations. Note that a single LMOD iteration involves a number of different computations (see section 18.4.2.). A value of zero results in a single local minimization; like a call to xmin.
nmod	5	The total number of low-frequency modes computed by LMOD every time such computation is requested.
minim_grms	0.1	The gradient RMS convergence criterion of structure minimization.
kmod	3	The definite number of randomly selected low-modes used to drive LMOD moves at each LMOD iteration step.
nrotran_dof	6	The number of rotational and translational degrees of freedom. This is related to the number of frozen or tethered atoms in the system: 0 atoms dof=6, 1 atom dof=3, 2 atoms dof=1, >=3 atoms dof=0. Default is 6, no frozen or tethered atoms. See section 18.4.7, note (5).
nconf	10	The maximum number of low-energy conformations stored in confliib[]. Note that the calling program is responsible for allocating memory for confliib[].
energy_window	50.0	The energy window for conformation storage; the energy of a stored structure will be in the interval [global_min, global_min + energy_window].
eig_recalc	5	The frequency, measured in LMOD iterations, of the recalculation of eigenvectors.
ndim_arnoldi	0	The dimension of the ARPACK Arnoldi factorization. The default, zero, specifies the whole space, that is, three times the number of atoms. See note below.

18. NAB: Molecular mechanics and dynamics

<i>keyword</i>	<i>default</i>	<i>meaning</i>
<code>lmod_restart</code>	10	The frequency, in LMOD iterations, of updating the conflib storage, that is, discarding structures outside the energy window, and restarting LMOD with a randomly chosen structure from the low-energy pool defined by <code>n_best_struct</code> below. A value $> \text{maxiter}$ will prevent LMOD from doing any restarts.
<code>n_best_struct</code>	10	Number of the lowest-energy structures found so far at a particular LMOD restart point. The structure to be used for the restart will be chosen randomly from this pool. <code>n_best_struct</code> = 1 allows the user to explore the neighborhood of the then current global minimum.
<code>mc_option</code>	1	The Monte Carlo method. 1= Metropolis Monte Carlo (see <code>rtemp</code> below). 2= "Total_Quench", which means that the LMOD trajectory always proceeds towards the lowest lying neighbor of a particular energy well found after exhaustive search along all of the randomly selected <code>kmod</code> low-modes. 3= "Quick_Quench", which means that the LMOD trajectory proceeds towards the first neighbor found, which is lower in energy than the current point on the path, without exploring the remaining modes.
<code>rtemp</code>	1.5	The value of RT in NAB energy units. This is utilized in the Metropolis criterion.
<code>lmod_step_size_min</code>	2.0	The minimum length of a single LMOD ZIG move in Å. See section 18.4.2 .
<code>lmod_step_size_max</code>	5.0	The maximum length of a single LMOD ZIG move in Å. See section 18.4.2 .
<code>nof_lmod_steps</code>	0	The number of LMOD ZIG-ZAG moves. The default, zero, means that the number of ZIG-ZAG moves is not pre-defined, instead LMOD will attempt to cross the barrier in as many ZIG-ZAG moves as it is necessary. The criterion of crossing an energy barrier is stated above in section 18.4.2 . <code>nof_lmod_steps</code> > 0 means that multiple barriers may be crossed and LMOD can carry the molecule to a large distance on the potential energy surface without severely distorting the geometry.
<code>lmod_relax_grms</code>	1.0	The gradient RMS convergence criterion of structure relaxation, see ZAG move in section 18.4.2 .
<code>nlig</code>	0	Number of ligands considered for flexible docking. The default, zero, means no docking.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
<code>apply_rigdock</code>	2	The frequency, measured in LMOD iterations, of the application of rigid-body rotational and translational motions to the ligand(s). At each <code>apply_rigdock</code> -th LMOD iteration <code>nof_pose_to_try</code> rotations and translations are applied to the ligand(s).
<code>nof_poses_to_try</code>	10	The number of rigid-body rotational and translational motions applied to the ligand(s). Such applications occur at each <code>apply_rigdock</code> -th LMOD iteration. In case <code>nof_pose_to_try</code> > 1, it is always the lowest energy pose that is kept, all other poses are discarded.
<code>random_seed</code>	314159	The seed of the random number generator. A value of zero requests hardware seeding based on the system clock.
<code>print_level</code>	0	Amount of debugging printout. 0= No output. 1= Basic output. 2= Detailed output. 3= Copious debugging output including ARPACK details.
<code>lmod_time</code>	N/A	CPU time in seconds used by LMOD itself.
<code>aux_time</code>	N/A	CPU time in seconds used by auxiliary routines.
<code>error_flag</code>	N/A	A non-zero value indicates an error. In case of an error LMOD will always print a descriptive error message.

Notes on the `ndim_arnoldi` parameter: Basically, the ARPACK package used for the eigen-vector calculations solves multiple "small" eigenvalue problems instead of a single "large" problem, which is the diagonalization of the three times the number of atoms by three times the number of atoms Hessian matrix. This parameter is the user specified dimension of the "small" problem. The allowed range is $nmod + 1 \leq ndim_arnoldi \leq 3 \cdot natm$. The default means that the "small" problem and the "large" problem are identical. This is the preferred, i.e., fastest, calculation for small to medium size systems, because ARPACK is guaranteed to converge in a single iteration. The ARPACK calculation scales with three times the number of atoms times the Arnoldi dimension squared and, therefore, for larger molecules there is an optimal `ndim_arnoldi` much less than three times the number of atoms that converges much faster in multiple iterations (possibly thousands or tens of thousands of iterations). The key to good performance is to select `ndim_arnoldi` such that all the ARPACK storage fits in memory. For proteins, `ndim_arnoldi` = 1000 is generally a good value, but often a very small ~50-100 Arnoldi dimension provides the fastest net computational cost with very many iterations.

18.4.6. Sample LMOD program

The following sample program, which is based on the test program `tlmod.nab`, reads a molecular structure from a PDB file, runs a short LMOD search, and saves the low-energy conformations in PDB files.

```

1 //  LMOD reverse communication external minimization package.
2 //  Written by Istvan Kolossvary.
3

```

18. NAB: Molecular mechanics and dynamics

```
4  #include "xmin_opt.h"
5  #include "lmod_opt.h"
6
7  // M A I N P R O G R A M to carry out LMOD simulation on a molecule/complex:
8
9  struct xmin_opt xo;
10 struct lmod_opt lo;
11
12 molecule mol;
13 int  natm;
14 float energy;
15 int lig_start[ dynamic ], lig_end[ dynamic ], lig_cent[ dynamic ];
16 float xyz[ dynamic ], grad[ dynamic ], conflib[ dynamic ], lmod_trajectory[ dynamic ];
17 float tr_min[ dynamic ], tr_max[ dynamic ], rot_min[ dynamic ], rot_max[ dynamic ];
18 float glob_min_energy;
19 point dummy;
20
21     lmod_opt_init( lo, xo );    // set up defaults
22
23     lo.niter          = 3;      // non-default options are here
24     lo.mc_option      = 2;
25     lo.nof_lmod_steps = 5;
26     lo.random_seed    = 99;
27     lo.print_level    = 2;
28
29     xo.ls_maxatmov    = 0.15;
30
31     mol = getpdb( "trpcage.pdb" );
32     readparm( mol, "trpcage.top" );
33     natm = mol.natoms;
34
35     allocate xyz[ 3*natm ]; allocate grad[ 3*natm ];
36     allocate conflib[ lo.nconf * 3*natm ];
37     allocate lmod_trajectory[ (lo.niter+1) * 3*natm ];
38     setxyz_from_mol( mol, NULL, xyz );
39
40     mm_options( "ntpr=5000, gb=0, cut=999.0, nsnb=9999, diel=R " );
41     mme_init( mol, NULL, "::ZZZ", dummy, NULL );
42
43     mme( xyz, grad, 1 );
44     glob_min_energy = lmod( natm, xyz, grad, energy,
45         conflib, lmod_trajectory, lig_start, lig_end, lig_cent,
46         tr_min, tr_max, rot_min, rot_max, xo, lo );
47
48     printf( "\nGlob. min. E          = %12.31f kcal/mol\n", glob_min_energy );
49
50
51 // E N D M A I N
```

The corresponding screen output should look similar to this.

Reading parm file (trpcage.top)

title:

```
mm_options: ntp=5000
mm_options: gb=0
mm_options: cut=999.0
mm_options: nsnb=9999
mm_options: diel=R
```

Low-Mode Simulation

```

1   E =   -118.117 ( 0.054)  Rg =    5.440
1 / 6 E =   -89.2057 ( 0.090)  Rg =    2.625  rmsd=  8.240  p= 0.0000
1 / 8 E =   -51.682 ( 0.097)  Rg =    5.399  rmsd=  8.217  p= 0.0000
3 /12 E =  -120.978 ( 0.091)  Rg =    3.410  rmsd=  7.248  p= 1.0000
3 /10 E =  -106.292 ( 0.099)  Rg =    5.916  rmsd=  4.829  p= 0.0004
4 / 6 E =  -106.788 ( 0.095)  Rg =    4.802  rmsd=  3.391  p= 0.0005
4 / 3 E =  -111.501 ( 0.097)  Rg =    5.238  rmsd=  2.553  p= 0.0121
```

```

2   E =  -120.978 ( 0.091)  Rg =    3.410
1 / 4 E =  -137.867 ( 0.097)  Rg =    2.842  rmsd=  5.581  p= 1.0000
1 / 9 E =  -130.025 ( 0.100)  Rg =    4.282  rmsd=  5.342  p= 1.0000
4 / 3 E =  -123.559 ( 0.089)  Rg =    3.451  rmsd=  1.285  p= 1.0000
4 / 4 E =  -107.253 ( 0.095)  Rg =    3.437  rmsd=  2.680  p= 0.0001
5 / 5 E =  -113.119 ( 0.096)  Rg =    3.136  rmsd=  2.074  p= 0.0053
5 / 4 E =   -134.1 ( 0.091)  Rg =    3.141  rmsd=  2.820  p= 1.0000
```

```

3   E =  -130.025 ( 0.100)  Rg =    4.282
1 / 8 E =  -150.556 ( 0.093)  Rg =    3.347  rmsd=  5.287  p= 1.0000
1 / 4 E =  -123.738 ( 0.079)  Rg =    4.218  rmsd=  1.487  p= 0.0151
2 / 8 E =  -118.254 ( 0.095)  Rg =    3.093  rmsd=  5.296  p= 0.0004
2 / 7 E =  -115.027 ( 0.090)  Rg =    4.871  rmsd=  4.234  p= 0.0000
4 / 7 E =  -128.905 ( 0.099)  Rg =    4.171  rmsd=  2.113  p= 0.4739
4 /11 E =  -133.85 ( 0.099)  Rg =    3.290  rmsd=  4.464  p= 1.0000
```

Full list:

```

1 E =   -150.556 / 1  Rg =    3.347
2 E =   -137.867 / 1  Rg =    2.842
3 E =    -134.1 / 1  Rg =    3.141
4 E =   -133.85 / 1  Rg =    3.290
5 E =   -130.025 / 1  Rg =    4.282
6 E =   -128.905 / 1  Rg =    4.171
7 E =   -123.738 / 1  Rg =    4.218
8 E =   -123.559 / 1  Rg =    3.451
9 E =   -120.978 / 1  Rg =    3.410
10 E =  -118.254 / 1  Rg =    3.093
```

Glob. min. E	=	-150.556 kcal/mol
Time in libLMOD	=	13.880 CPU sec
Time in NAB and libs	=	63.760 CPU sec

The first few lines come from *mm_init()* and *mme()*. The screen output below the horizontal line originates from LMOD. Each LMOD-iteration is represented by a multi-line block of data numbered in the upper left corner by the iteration count. Within each block, the first line displays the energy and, in parentheses, the gradient RMS as well as the radius of gyration (assigning unit mass to each atom), of the current structure along the LMOD pseudo simulation-path. The successive lines within the block provide information about the LMOD ZIG-ZAG moves (see section 18.4.2). The number of lines is equal to 2 times *kmodes* (2x3 in this example). Each selected mode is explored in both directions, shown in two separate lines. The leftmost number is the serial number of the mode (randomly selected from the set of *nmodes* modes) and the number after the slash character gives the number of ZIG-ZAG moves taken. This is followed by, respectively, the minimized energy and gradient RMS, the radius of gyration, the RMSD distance from the base structure, and the Boltzmann probability with respect to the energy of the base structure and *rtemp*, of the minimized structure at the end of the ZIG-ZAG path. Note that exploring the same mode along both directions can result in two quite different structures. Also note that the number of ZIG-ZAG moves required to cross the energy barrier (see section 18.4.2) in different directions can vary quite a bit, too. Occasionally, an exclamation mark next to the energy (!E = ...) denotes a structure that could not be fully minimized.

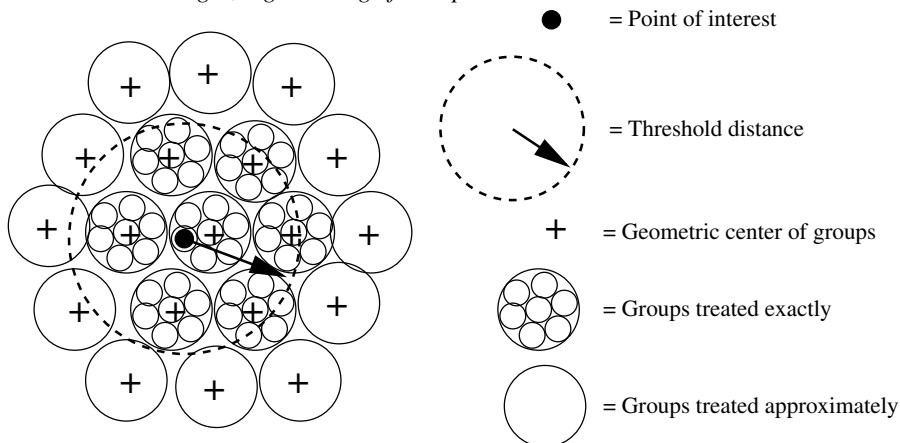
After finishing all the computation within a block, the corresponding LMOD step is completed by selecting one of the ZIG-ZAG endpoint structures as the base structure of the next LMOD iteration. The selection is based on the *mc_option* and the Boltzmann probability. The LMOD pseudo simulation-path is defined by the series of these *mc_option*-selected structures and it is stored in *lmod_traj[]*. Note that the sample program saves these structures in a multi-PDB disk file called *lmod_trajectory.pdb*. The final section of the screen output lists the *nconf* lowest energy structures found during the LMOD search. Note that some of the lowest energy structures are not necessarily included in the *lmod_traj[]* list, as it depends on the *mc_option* selection. The list displays the energy, the number of times a particular conformation was found (increasing numbers are somewhat indicative of a more complete search), and the radius of gyration. The sample program writes the top ten low-energy structures in separate, numbered PDB files. The glob. min. energy and the timing results are printed from the sample NAB program, not from LMOD.

As a final note, it is instructive to be aware of a simple safeguard that LMOD applies. A copy of the *conflib[]* array is saved periodically in a binary disk file called *conflib.dat*. Since LMOD searches might run for a long time, in case of a crash low-energy structures can be recovered from this file. The format of *conflib.dat* is as follows. Each conformation is represented by 3 numbers (double energy, double radius of gyration, and int number of times found), followed by the double (x, y, z) coordinates of the atoms.

18.4.7. Tricks of the trade of running LMOD searches

1. The AMBER atom types HO, HW, and ho all have zero van der Waals parameters in all of the AMBER (and some other) force fields. Corresponding Aij and Bij coefficients in the PRMTOP file are set to zero. This means there is no repulsive wall to prevent two oppositely charged atoms, one being of type HO, HW or ho, to fuse as a result of the ever decreasing electrostatic energy as they come closer and closer to each other. This potential problem is rarely manifest in molecular dynamics simulations, but it presents a nuisance when running LMOD searches. The problem is local minimization, especially "aggressive" TNCG minimization (XMIN xo.method=3) that can easily result in atom fusion. Therefore, before running an LMOD simulation, the PRMTOP file (let's call it prmtop.in) must be processed by running the script "lmodprmtop prmtop.in prmtop.out". This script will replace all the repulsive Aij coefficients set to zero in prmtop.in with a high value of 1e03 in prmtop.out in order to re-create the van der Waals wall. It is understood that this procedure is parameter fudging; however, note that the primary goal of using LMOD is the quick generation of approximate, low-energy structures that can be further refined by high-accuracy MD.
2. LMOD requires that the potential energy surface is continuous everywhere to a great degree. Therefore, always use a distance dependent dielectric constant in mm_options when running searches in vacuo, or use GB solvation (note that GB calculations will be slow), and always apply a large cut-off. It does make sense to run quick and dirty LMOD searches in vacuo to generate low-energy starting structures for MD runs. Note that the most likely symptom of discontinuities causing a problem is when your NAB program utilizing LMOD is grabbing CPU time, but the LMOD search does not seem to progress. This is the result of NaN's that often can be seen when print_level is set to > 0.
3. LMOD is NOT INTENDED to be used with explicit water models and periodic boundary conditions. Although explicit-water solvation representation is not recommended, LMOD docking can be readily used with crystallographic water molecules as ligands.
4. Conformations in the confflib and lmod_trajectory files can have very different orientations. One trick to keep them in a common orientation is to restrain the position of, e.g., a single benzene ring. This will ensure that the molecule cannot be translated or rotated as a whole. However, when applying this trick you should set nrotran_dof = 0.
5. A subset of the atoms of a molecular system can be frozen or tethered/restrained in NAB by two different methods. Atoms can either be frozen by using the first atom expression argument in *mme_init()* or restrained by using the second atom expression argument and the reference coordinate array in *mme_init()* along with the *wcons* option in mm_options. LMOD searches, especially docking calculations can be run much faster if parts of the molecular system can be frozen, because the effective degrees of freedom is determined by the size of the flexible part of the system. Application of frozen atoms means that a much smaller number of moving atoms are moving in the fixed, external potential of the frozen atoms. The tethered atom model is expected to give similar results to the frozen atom model, but note that the number of degrees of freedom and, therefore, the computational cost of a tethered calculation is comparable to that of a fully unrestrained system.

Figure 18.1.: *The HCP threshold distance. For the level 1 approximation shown here, groups within the threshold distance are treated exactly using atomic charges, while groups beyond the threshold distance are approximated by a small number of charges, e.g. 1 charge for $hcp=1$ shown here.*



However, the eigenvector calculations are likely to converge faster with the tethered systems.

18.5. Using the Hierarchical Charge Partitioning (HCP) method

The HCP is an $N \log N$ approximation for computing long range electrostatic interactions [240, 241]. This method uses the natural organization of biomolecular structures to partition the structure into multiple hierarchical levels of components - atoms, groups, chains, and complexes. The charge distribution for each of these components is approximated by 1 ($hcp=1$) or 2 ($hcp=2$) charges. The approximate charges are then used for computing electrostatic interactions with distant components while the full set of atomic charges are used for nearby components (Figure 17.1). The HCP can be used for gas phase ($dielec=C$), distant dependent dielectric ($dielec=R/RL$), and generalized Born ($gb=1-5$) simulations with or without Langevin dynamics ($\gamma_{ln}>0$). The speedup from using the HCP for MD can be up to 3 orders of magnitude, depending on structure size.

18.5.1. Level 1 HCP approximation

The HCP option can now be used with one level of approximation (groups) using the NAB molecular dynamics scripts described in Section 18.2 above. No additional manipulation of the input structure files is required for one level of approximation. For an example see `AmberTools/examples/hcp/2trx.nab`. The level 1 approximation is recommended for single domain and

small (< 10,000 atoms) multi-domain structures. Speedups of 2x-10x can be realized using the level 1 approximation, depending on structure size.

18.5.2. Level 2 and 3 HCP approximation

For larger multi-domain structures higher levels of approximations (chains and complexes) can be used to achieve up to 3 orders of magnitude speedups, depending on structure size. The following additional steps are required to include information about these higher level components in the prmtop file. For an example see AmberTools/examples/hcp/1kx5.nab.

1. Ensure the pdb file identifies the higher level structures: Chains (level 2) separated by TER, and Complexes (level 3) separated by REMARK END-OF-COMPLEX:
...
ATOM ...
TER (end of chain)
ATOM ...
...
ATOM ...
TER (end of chain)
REMARK END-OF-COMPLEX
ATOM ...
2. Execute hcp_getpdb to generate prmtop entries for HCP: hcp_getpdb pdb-filename > hcp-prmtop
3. Concatenate the HCP prmtop entries to the end of the standard prmtop file generated by Leap: cat prmtop-file hcp-prmtop > new-prmtop
4. Use this new prmtop file in the NAB molecular dynamics scripts instead of the prmtop file generated by Leap

19. NAB: Sample programs

This chapter provides a variety of examples that use the basic NAB functionality described in earlier chapters to solve interesting molecular manipulation problems. Our hope is that the ideas and approaches illustrated here will facilitate construction of similar programs to solve other problems.

19.1. Duplex Creation Functions

nab provides a variety of functions for creating Watson/Crick duplexes. A short description of four of them is given in this section. All four of these functions are written in nab and the details of their implementation is covered in the section **Creating Watson/Crick Duplexes** of the **User Manual**. You should also look at the function `fd_helix()` to see how to create duplex helices that correspond to fibre-diffraction models. As with the PERL language, "there is more than one way to do it."

```
molecule bdna( string seq );
string wc_complement( string seq, string rlib, string rlt );
molecule wc_helix( string seq, string rlib, string natype, string cseq, string crlib,
    string cnatype, float xoffset, float incl, float twist, float rise, string options );
molecule dg_helix( string seq, string rlib, string natype,
    string cseq, string crlib, string cnatype, float xoffset, float incl, float twist, float rise,
    string options );
molecule wc_basepair( residue res, residue cres );
```

`bdna()` converts the character string `seq` containing one or more A, C, G or Ts (or their lower case equivalents) into a uniform ideal Watson/Crick B-form DNA duplex. Each basepair has an X-offset of 2.25 Å, an inclination of -4.96 Å and a helical step of 3.38 Å rise and 36.00 twist. The first character of `seq` is the 5' base of the strand "sense" of the molecule returned by `bdna()`. The other strand is called "anti". The phosphates of the two 5' bases have been replaced by hydrogens and hydrogens have been added to the two O3' atoms of the three prime bases. `bdna()` returns NULL if it can not create the molecule.

`wc_complement()` returns a string that is the Watson/Crick complement of its argument `seq`. Each C, G, T (U) in `seq` is replaced by G, C and A. The replacements for A depends if `rlt` is DNA or RNA. If it is DNA, A is replaced by T. If it is RNA A is replaced by U. `wc_complement()` considers lower case and upper case letters to be the same and always returns upper case letters. `wc_complement()` returns NULL on error. Note that while the orientations of the argument

string and the returned string are opposite, their absolute orientations are *undefined* until they are used to create a molecule.

`wc_helix()` creates a uniform duplex from its arguments. The two strands of the returned molecule are called "sense" and "anti". The two sequences, `seq` and `cseq` must specify Watson/Crick base pairs. Note that must be specified as *lower-case* strings, such as "ggact". The nucleic acid type (DNA or RNA) of the sense strand is specified by `natype` and of the complementary strand `cseq` by `cnatype`. Two residue libraries—`rlib` and `crlib`— permit creation of DNA:RNA heteroduplexes. If either `seq` or `cseq` (but not both) is NULL only the specified strand of what would have been a uniform duplex is created. The `options` string contains some combination of the strings "s5", "s3", "a5" and "a3"; these indicate which (if any) of the ends of the helices should be "capped" with hydrogens attached to the O5' atom (in place of a phosphate) if "s5" or "a5" is specified, and a proton added to the O3' position if "s3" or "a3" is specified. A blank string indicates no capping, which would be appropriate if this section of helix were to be inserted into a larger molecule. The string "s5a5s3a3" would cap the 5' and 3' ends of both the "sense" and "anti" strands, leading to a chemically complete molecule. `wc_helix()` returns NULL on error.

`dg_helix()` is the functional equivalent of `wc_helix()` but with the backbone geometry minimized via a distance constraint error function. `dg_helix()` takes the same arguments as `wc_helix()`.

`wc_basepair()` assembles two nucleic acid residues (assumed to be in a standard orientation) into a two stranded molecule containing one Watson/Crick base pair. The two strands of the new molecule are "sense" and "anti". It returns NULL on error.

19.2. nab and Distance Geometry

Distance geometry is a method which converts a molecule represented as a set of interatomic distances and related information into a 3-D structure. `nab` has several builtin functions that are used together to provide metric matrix distance geometry. `nab` also provides the `bounds` type for holding a molecule's distance geometry information. A `bounds` object contains the molecule's interatomic distance bounds matrix and a list of its chiral centers and their volumes. `nab` uses chiral centers with a volume of 0 to enforce planarity.

Distance geometry has several advantages. It is unique in its power to create structures from very incomplete descriptions. It easily incorporates "low resolution structural data" such as that derived from chemical probing since these kinds of experiments generally return only distance bounds. And it also provides an elegant method by which structures may be described functionally.

The `nab` distance geometry package is described more fully in the section **NAB Language Reference**. Generally, the function `newbounds()` creates and returns a `bounds` object corresponding to the molecule `mol`. This object contains two things—a distance bounds matrix containing initial upper and lower bounds for every pair of atoms in `mol` and a initial list of the molecules chiral centers and their volumes. Once a `bounds` object has been initialized, the modeller uses functions from the middle of the distance geometry function list to tighten, loosen or set other distance bounds and chiralities that correspond to experimental measurements or parts of the model's hypothesis. The four functions `andbounds()`, `orbounds()`, `setbounds` and `use-`

`boundsfrom()` work in similar fashion. Each uses two atom expressions to select pairs of atoms from `mol`. In `andbounds()`, the current distance bounds of each pair are compared against `lb` and `ub` and are replaced by `lb`, `ub` if they represent tighter bounds. `orbounds()` replaces the current bounds of each selected pair, if `lb`, `ub` represent looser bounds. `setbounds()` sets the bounds of all selected pairs to `lb`, `ub`. `useboundsfrom()` sets the bounds between each atom selected in the first expression to a percentage of the distance between the atoms selected in the second atom expression. If the two atom expressions select the same atoms from the same molecule, the bounds between all the atoms selected will be constrained to the current geometry. `setchivol()` takes four atom expressions that must select exactly four atoms and sets the volume of the tetrahedron enclosed by those atoms to `vol`. Setting `vol` to 0 forces those atoms to be planar. `getchivol()` returns the chiral volume of the tetrahedron described by the four points.

After all experimental and model constraints have been entered into the bounds object, the function `tsmooth()` applies a process called “triangle smoothing” to them. This tests each triple of distance bounds to see if they can form a triangle. If they can not form a triangle then the distance bounds do not even represent a Euclidean object let alone a 3-D one. If this occurs, `tsmooth()` quits and returns a 1 indicating failure. If all triples can form triangles, `tsmooth()` returns a 0. Triangle smoothing pulls in the large upper bounds. After all, the maximum distance between two atoms can not exceed the sum of the upper bounds of the shortest path between them. Triangle smoothing can also increase lower bounds, but this process is much less effective as it requires one or more large lower bounds to begin with.

The function `embed()` takes the smoothed bounds and converts them into a 3-D object. This process is called “embedding”. It does this by choosing a random distance for each pair of atoms within the bounds of that pair. Sometimes the bounds simply do not represent a 3-D object and `embed()` fails, returning the value 1. This is rare and usually indicates the that the distance bounds matrix part of the bounds object contains errors. If the distance set does embed, `conjugrad()` can subject newly embedded coordinates to conjugate gradient refinement against the distance and chirality information contained in bounds. The refined coordinates can replace the current coordinates of the molecule in `mol`. `embed()` returns a 0 on success and `conjugrad()` returns an exit code explained further in the **Language Reference** section of this manual. The call to `embed()` is usually placed in a loop with each new structure saved after each call to see the diversity of the structures the bounds represent.

In addition to the explicit bounds manipulation functions, *nab* provides an implicit way of setting bounds between interacting residues. The function `setboundsfromdb()` is for use in creating distance and chirality bounds for nucleic acids. `setboundsfromdb()` takes as an argument two atom expressions selecting two residues, the name of a database containing bounds information, and a number which dictates the tightness of the bounds. For instance, if the database *bdna.stack.db* is specified, `setboundsfromdb()` sets the bounds between the two residues to what they would be if they were stacked in strand in a typical Watson-Crick B-form duplex. Similarly, if the database *arna.basepair.db* is specified, `setboundsfromdb()` sets the bounds between the two residues to what they would be if the two residues form a typical Watson-Crick basepair in an A-form helix.

19.2.1. Refine DNA Backbone Geometry

As mentioned previously, `wc_helix()` performs rigid body transformations on residues and does not correct for poor backbone geometry. Using distance geometry, several techniques are available to correct the backbone geometry. In program 7, an 8-basepair dna sequence is created using `wc_helix()`. A new bounds object is created on line 14, which automatically sets all the 1-2, 1-3, and 1-4 distance bounds information according to the geometry of the model. Since this molecule was created using `wc_helix()`, the O3'-P distance between adjacent stacked residues is often not the optimal 1.595 Å, and hence, the 1-2, 1-3, and 1-4, distance bounds set by `newbounds()` are incorrect. We want to preserve the position of the nucleotide bases, however, since this is the helix whose backbone we wish to minimize. Hence the call to `useboundsfrom()` on line 17 which sets the bounds from every atom in each nucleotide base to the actual distance to every other atom in every other nucleotide base. *In general, the likelihood of a distance geometry refinement to satisfy a given bounds criteria is proportional to the number of (consistent) bounds set supporting that criteria.* In other words, the more bounds that are set supporting a given conformation, the greater the chance that conformation will resolve after the refinement. An example of this concept is the use of `useboundsfrom()` in line 17, which works to preserve our rigid helix conformation of all the nucleotide base atoms.

We can correct the backbone geometry by overwriting the erroneous bounds with more appropriate bounds. In lines 19-29, all the 1-2, 1-3, and 1-4 bounds involving the O3'-P connection between strand 1 residues are set to that which would be appropriate for an idealized phosphate linkage. Similarly, in lines 31-41, all the 1-2, 1-3, and 1-4 bounds involving the O3'-P connection among strand 2 residues are set to an idealized conformation. This technique is effective since all the 1-2, 1-3, and 1-4 distance bounds created by `newbounds()` include those of the idealized nucleotides in the nucleic acid libraries `dna.amber94.rlb`, `rna.amber94.rlb`, etc. contained in `reslib`. Hence, by setting these bounds and refining against the distance energy function, we are spreading the 'error' across the backbone, where the 'error' is the departure from the idealized sugar conformation and idealized phosphate linkage.

On line 43, we smooth the bounds matrix, and on line 44 we give a substantial penalty for deviating from a 3-D refinement by setting `k4d=4.0`. Notice that there is no need to embed the molecule in this program, as the actual coordinates are sufficient for any refinement.

```

1 // Program 7 - refine backbone geometry using distance function
2 molecule m;
3 bounds b;
4 string seq, cseq;
5 int i;
6 float xyz[ dynamic ], fret;
7
8 seq = "acgtacgt";
9 cseq = wc_complement( "acgtacgt", "", "dna" );
10
11 m = wc_helix( seq, "", "dna", cseq, "",
12              "dna", 2.25, -4.96, 36.0, 3.38, "" );
13
14 b = newbounds(m, "");
15 allocate xyz[ 4*m.natoms ];

```

```

16
17 useboundsfrom(b, m, " :??,H?[^T' ]", m, " :??,H?[^T' ]", 0.0 );
18 for ( i = 1; i < m.nresidues/2 ; i = i + 1 ){
19     setbounds(b,m, sprintf("1:%d:O3'",i),
20               sprintf("1:%d:P",i+1), 1.595,1.595);
21     setbounds(b,m, sprintf("1:%d:O3'",i),
22               sprintf("1:%d:O5'",i+1), 2.469,2.469);
23     setbounds(b,m, sprintf("1:%d:C3'",i),
24               sprintf("1:%d:P",i+1), 2.609,2.609);
25     setbounds(b,m, sprintf("1:%d:O3'",i),
26               sprintf("1:%d:O1P",i+1), 2.513,2.513);
27     setbounds(b,m, sprintf("1:%d:O3'",i),
28               sprintf("1:%d:O2P",i+1), 2.515,2.515);
29     setbounds(b,m, sprintf("1:%d:C4'",i),
30               sprintf("1:%d:P",i+1), 3.550,4.107);
31     setbounds(b,m, sprintf("1:%d:C2'",i),
32               sprintf("1:%d:P",i+1), 3.550,4.071);
33     setbounds(b,m, sprintf("1:%d:C3'",i),
34               sprintf("1:%d:O1P",i+1), 3.050,3.935);
35     setbounds(b,m, sprintf("1:%d:C3'",i),
36               sprintf("1:%d:O2P",i+1), 3.050,4.004);
37     setbounds(b,m, sprintf("1:%d:C3'",i),
38               sprintf("1:%d:O5'",i+1), 3.050,3.859);
39     setbounds(b,m, sprintf("1:%d:O3'",i),
40               sprintf("1:%d:C5'",i+1), 3.050,3.943);
41
42     setbounds(b,m, sprintf("2:%d:P",i+1),
43               sprintf("2:%d:O3'",i), 1.595,1.595);
44     setbounds(b,m, sprintf("2:%d:O5'",i+1),
45               sprintf("2:%d:O3'",i), 2.469,2.469);
46     setbounds(b,m, sprintf("2:%d:P",i+1),
47               sprintf("2:%d:C3'",i), 2.609,2.609);
48     setbounds(b,m, sprintf("2:%d:O1P",i+1),
49               sprintf("2:%d:O3'",i), 2.513,2.513);
50     setbounds(b,m, sprintf("2:%d:O2P",i+1),
51               sprintf("2:%d:O3'",i), 2.515,2.515);
52     setbounds(b,m, sprintf("2:%d:P",i+1),
53               sprintf("2:%d:C4'",i), 3.550,4.107);
54     setbounds(b,m, sprintf("2:%d:P",i+1),
55               sprintf("2:%d:C2'",i), 3.550,4.071);
56     setbounds(b,m, sprintf("2:%d:O1P",i+1),
57               sprintf("2:%d:C3'",i), 3.050,3.935);
58     setbounds(b,m, sprintf("2:%d:O2P",i+1),
59               sprintf("2:%d:C3'",i), 3.050,4.004);
60     setbounds(b,m, sprintf("2:%d:O5'",i+1),
61               sprintf("2:%d:C3'",i), 3.050,3.859);
62     setbounds(b,m, sprintf("2:%d:C5'",i+1),
63               sprintf("2:%d:O3'",i), 3.050,3.943);
64 }

```

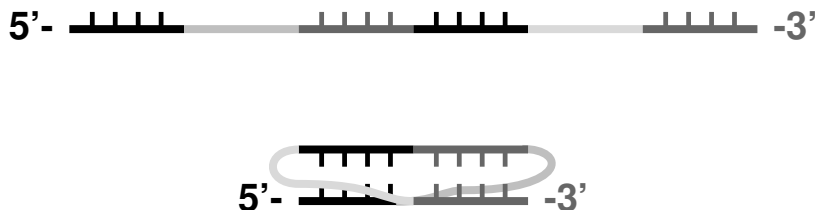


Figure 19.1.: Single-stranded RNA (top) folded into a pseudoknot (bottom). The black and dark grey base pairs can be stacked.

```

65 tsmooth( b, 0.0005 );
66 dg_options( b, "seed=33333, gdist=0, ntp=100, k4d=4.0" );
67 setxyzw_from_mol( m, NULL, xyz );
68 conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 500 );
69 setmol_from_xyzw( m, NULL, xyz );
70 putpdb( "acgtacgt.pdb", m );

```

The approach of Program 7 is effective but has a disadvantage in that it does not scale linearly with the number of atoms in the molecule. In particular, `tsmooth()` and `conjgrad()` require extensive CPU cycles for large numbers of residues. For this reason, the function `dg_helix()` was created. `dg_helix()` takes uses the same method of Program 7, but employs a 3-basepair helix template which traverses the new helix as it is being constructed. In this way, the helix is built in a piecewise manner and the maximum number of residues considered in each refinement is less than or equal to six. This is the preferred method of helix construction for large, idealized canonical duplexes.

19.2.2. RNA Pseudoknots

In addition to the standard helix generating functions, `nab` provides extensive support for generating initial structures from low structural information. As an example, we will describe the construction of a model of an RNA pseudoknot based on a small number of secondary and tertiary structure descriptions. Shen and Tinoco (*J. Mol. Biol.* **247**, 963-978, 1995) used the molecular mechanics program X-PLOR to determine the three dimensional structure of a 34 nucleotide RNA sequence that folds into a pseudoknot. This pseudoknot promotes frame shifting in Mouse Mammary Tumor Virus. A pseudoknot is a single stranded nucleic acid molecule that contains two improperly nested hairpin loops as shown in Figure 19.1. NMR distance and angle constraints were converted into a three dimensional structure using a two stage restrained molecular dynamics protocol. Here we show how a three-dimensional model can be constructed using just a few key features derived from the NMR investigation.

```

1 // Program 8 - create a pseudoknot using distance geometry
2 molecule m;
3 float xyz[ dynamic ],f[ dynamic ],v[ dynamic ];
4 bounds b;
5 int i, seqlen;
6 float fret;

```



```

7  string seq, opt;
8
9  seq = "GCGGAAACGCCGCGUAAGCG";
10
11 seqlen = length(seq);
12
13 m = link_na("1", seq, "", "RNA", "35");
14
15 allocate xyz[ 4*m.natoms ];
16 allocate f[ 4*m.natoms ];
17 allocate v[ 4*m.natoms ];
18
19 b = newbounds(m, "");
20
21 for ( i = 1; i <= seqlen; i = i + 1 ) {
22     useboundsfrom(b, m, sprintf("1:%d:??,H?[^'T]", i), m,
23     sprintf("1:%d:??,H?[^'T]", i), 0.0 );
24 }
25
26 setboundsfromdb(b, m, "1:1:", "1:2:", "arna.stack.db", 1.0);
27 setboundsfromdb(b, m, "1:2:", "1:3:", "arna.stack.db", 1.0);
28 setboundsfromdb(b, m, "1:3:", "1:18:", "arna.stack.db", 1.0);
29 setboundsfromdb(b, m, "1:18:", "1:19:", "arna.stack.db", 1.0);
30 setboundsfromdb(b, m, "1:19:", "1:20:", "arna.stack.db", 1.0);
31
32 setboundsfromdb(b, m, "1:8:", "1:9:", "arna.stack.db", 1.0);
33 setboundsfromdb(b, m, "1:9:", "1:10:", "arna.stack.db", 1.0);
34 setboundsfromdb(b, m, "1:10:", "1:11:", "arna.stack.db", 1.0);
35 setboundsfromdb(b, m, "1:11:", "1:12:", "arna.stack.db", 1.0);
36 setboundsfromdb(b, m, "1:12:", "1:13:", "arna.stack.db", 1.0);
37
38 setboundsfromdb(b, m, "1:1:", "1:13:", "arna.basepair.db", 1.0);
39 setboundsfromdb(b, m, "1:2:", "1:12:", "arna.basepair.db", 1.0);
40 setboundsfromdb(b, m, "1:3:", "1:11:", "arna.basepair.db", 1.0);
41
42 setboundsfromdb(b, m, "1:8:", "1:20:", "arna.basepair.db", 1.0);
43 setboundsfromdb(b, m, "1:9:", "1:19:", "arna.basepair.db", 1.0);
44 setboundsfromdb(b, m, "1:10:", "1:18:", "arna.basepair.db", 1.0);
45
46 tsmooth(b, 0.0005);
47
48 opt = "seed=571, gdist=0, ntp=50, k4d=2.0, randpair=5., sqviol=1";
49 dg_options( b, opt );
50 embed(b, xyz );
51
52 conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 500 );
53
54 setmol_from_xyzw( m, NULL, xyz );
55 putpdb( "rna_pseudoknot.pdb", m );

```

Program 8 uses distance geometry followed by minimization and simulated annealing to create a model of a pseudoknot. Distance geometry code begins in line 20 with the call to `newbounds()` and ends on line 53 with the call to `embed()`. The structure created with distance geometry is further refined with molecular dynamics in lines 58-74. Note that very little structural information is given - only connectivity and general base-base interactions. The stacking and base-pair interactions here are derived from NMR evidence, but in other cases might arise from other sorts of experiments, or as a model hypothesis to be tested.

The 20-base RNA sequence is defined on line 9. The molecule itself is created with the `link_na()` function call which creates an extended conformation of the RNA sequence and caps the 5' and 3' ends. Lines 15-17 define arrays that will be used in the simulated annealing of the structure. The bounds object is created in line 19 which automatically sets the 1-2, 1-3, and 1-4 distance bounds in the molecule. The loop in lines 21-24 sets the bounds of each atom in each residue base to the actual distance to every other atom in the same base. This has the effect of enforcing the planarity of the base by treating the base somewhat like a rigid body. In lines 26-44, bounds are set according to information stored in a database. The `setboundsfromdb()` call sets the bounds from all the atoms in the two specified residues to a 1.0 multiple of the standard deviation of the bounds distances in the specified database. Specifically, line 26 sets the bounds between the base atoms of the first and second residues of strand 1 to be within one standard deviation of a *typical* aRNA stacked pair. Similarly, line 38 sets the bounds between residues 1 and 13 to be that of *typical* Watson-Crick basepairs. For a description of the `setboundsfromdb()` function, see Chapter 1.

Line 46 smooths the bounds matrix, by attempting to adjust any sets of bounds that violate the triangle equality. Lines 48-49 initialize some distance geometry variables by setting the random number generator seed, declaring the type of distance distribution, how often to print the energy refinement process, declaring the penalty for using a 4th dimension in refinement, and which atoms to use to form the initial metric matrix. The coordinates are calculated and embedded into a 3D coordinate array, `xyz` by the `embed()` function call on line 50.

The coordinates `xyz` are subject to conjugate gradient refinements in line 52. Line 54 replaces the old molecular coordinates with the new refined ones, and lastly, on line 55, the molecule is saved as `"rna_pseudoknot.pdb"`.

The resulting structure of Program 8 is shown in Figure 19.2. This structure had an final total energy of 46 units. The helical region, shown as polytubes, shows stacking and wc-pairing interactions and a well-defined right-handed helical twist. Of course, good modeling of a "real" pseudoknot would require putting in more constraints, but this example should illustrate how to get started on problems like this.

19.2.3. NMR refinement for a protein

Distance geometry techniques are often used to create starting structures in NMR refinement. Here, in addition to the covalent connections, one makes use of a set of distance and torsional restraints derived from NMR data. While NAB is not (yet?) a fully-functional NMR refinement package, it has enough capabilities to illustrate the basic ideas, and could be the starting point for a flexible procedure. Here we give an illustration of how the rough structure of a protein can

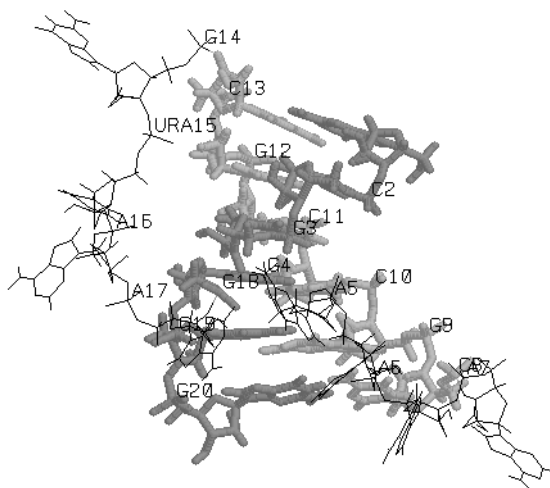


Figure 19.2.: *Folded RNA pseudoknot.*

be determined using distance geometry and NMR distance constraints; the structures obtained here would then be candidates for further refinement in programs like X-plor or Amber.

The program below illustrates a general procedure for a primarily helical DNA binding domain. Lines 15-22 just construct the sequence in an extended conformation, such that bond lengths and angles are correct, but none of the torsions are correct. The bond lengths and angles are used by `newbounds()` to construct the "covalent" part of the bounds matrix.

```

1 // Program 8a. General driver routine to do distance geometry \fc
2 //    on proteins, with DYANA-like distance restraints.\fc
3
4 #define MAXCOORDS 12000
5
6 molecule m;
7 atom      a;
8 bounds    b;
9
10 int      ier,i, numstrand, ires,jres;
11
12 float     fret, rms, ub;
13
14 float      xyz[ MAXCOORDS ], f[ MAXCOORDS ], v[ MAXCOORDS ];
15
16 file       boundsf;
17
18 string     iresname,jresname,iat,jat,aex1,aex2,aex3,aex4,line,dgopts,seq;
19
20 // sequence of the mrf2 protein:
21
22 seq = "RADEQAFVLVALYKYMKERKTPIERIPYLGFKQINLWTFQAAQKLGGYETITARRQWKHIY"
23      + "DELGGNPGSTSAATCTRRHYERLILPYERFIKGEEDKPLPPIKPRK";
24
25 // build this sequence in an extended conformation, and construct a bounds

```

19. NAB: Sample programs

```

20 //      matrix just based on the covalent structure:
21 m = linkprot( "A", seq, "" );
22 b = newbounds( m, "" );
23
24 //  read in constraints, updating the bounds matrix using "andbounds":
25
26 //  distance constraints are basically those from Y.-C. Chen, R.H. Whitson
27 //  Q. Liu, K. Itakura and Y. Chen, "A novel DNA-binding motif shares
28 //  structural homology to DNA replication and repair nucleases and
29 //  polymerases," Nature Struct. Biol. 5:959-964 (1998).
30
31 boundsf = fopen( "mrf2.7col", "r" );
32 while( line = getline( boundsf ) ){
33     sscanf( line, "%d %s %s %d %s %s %lf", ires, iresname, iat,
34             jres, jresname, jat, ub );
35
36 //      translations for DYANA-style pseudoatoms:
37     if( iat == "HN" ){ iat = "H"; }
38     if( jat == "HN" ){ jat = "H"; }
39
40     if( iat == "QA" ){ iat = "CA"; ub += 1.0; }
41     if( jat == "QA" ){ jat = "CA"; ub += 1.0; }
42     if( iat == "QB" ){ iat = "CB"; ub += 1.0; }
43     if( jat == "QB" ){ jat = "CB"; ub += 1.0; }
44     if( iat == "QG" ){ iat = "CG"; ub += 1.0; }
45     if( jat == "QG" ){ jat = "CG"; ub += 1.0; }
46     if( iat == "QD" ){ iat = "CD"; ub += 1.0; }
47     if( jat == "QD" ){ jat = "CD"; ub += 1.0; }
48     if( iat == "QE" ){ iat = "CE"; ub += 1.0; }
49     if( jat == "QE" ){ jat = "CE"; ub += 1.0; }
50     if( iat == "QQG" ){ iat = "CB"; ub += 1.8; }
51     if( jat == "QQG" ){ jat = "CB"; ub += 1.8; }
52     if( iat == "QQD" ){ iat = "CG"; ub += 1.8; }
53     if( jat == "QQD" ){ jat = "CG"; ub += 1.8; }
54     if( iat == "QG1" ){ iat = "CG1"; ub += 1.0; }
55     if( jat == "QG1" ){ jat = "CG1"; ub += 1.0; }
56     if( iat == "QG2" ){ iat = "CG2"; ub += 1.0; }
57     if( jat == "QG2" ){ jat = "CG2"; ub += 1.0; }
58     if( iat == "QD1" ){ iat = "CD1"; ub += 1.0; }
59     if( jat == "QD1" ){ jat = "CD1"; ub += 1.0; }
60     if( iat == "QD2" ){ iat = "ND2"; ub += 1.0; }
61     if( jat == "QD2" ){ jat = "ND2"; ub += 1.0; }
62     if( iat == "QE2" ){ iat = "NE2"; ub += 1.0; }
63     if( jat == "QE2" ){ jat = "NE2"; ub += 1.0; }
64
65     aex1 = ":" + sprintf( "%d", ires ) + ":" + iat;
66     aex2 = ":" + sprintf( "%d", jres ) + ":" + jat;
67     andbounds( b, m, aex1, aex2, 0.0, ub );
68 }

```

```

69 fclose( boundsf );
70
71 // add in helical chirality constraints to force right-handed helices:
72 // (hardwire in locations 1-16, 36-43, 88-92)
73 for( i=1; i<=12; i++){
74     aex1 = ":" + sprintf( "%d", i ) + ":CA";
75     aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
76     aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
77     aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
78     setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
79 }
80 for( i=36; i<=39; i++){
81     aex1 = ":" + sprintf( "%d", i ) + ":CA";
82     aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
83     aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
84     aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
85     setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
86 }
87 for( i=88; i<=89; i++){
88     aex1 = ":" + sprintf( "%d", i ) + ":CA";
89     aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
90     aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
91     aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
92     setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
93 }
94
95 // set up some options for the distance geometry calculation
96 // here use the random embed method:
97 dgopts = "ntpr=10000,rembed=1,rbox=300.,riter=250000,seed=8511135";
98 dg_options( b, dgopts );
99
100 // do triangle-smoothing on the bounds matrix, then embed:
101 geodesics( b ); embed( b, xyz );
102
103 // now do conjugate-gradient minimization on the resulting structures:
104
105 // first, weight the chirality constraints heavily:
106 dg_options( b, "ntpr=20, k4d=5.0, sqviol=0, kchi=50." );
107 conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.02, 1000., 300 );
108
109 // next, squeeze out the fourth dimension, and increase penalties for
110 // distance violations:
111 dg_options( b, "k4d=10.0, sqviol=1, kchi=50." );
112 conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.02, 100., 400 );
113
114 // transfer the coordinates from the "xyz" array to the molecule
115 // itself, and print out the violations:
116 setmol_from_xyzw( m, NULL, xyz );
117 dumpboundsviolations( stdout, b, 0.5 );

```

```

118
119 // do a final short molecular-mechanics "clean-up":
120 putpdb( m, "temp.pdb" );
121 m = getpdb_prm( "temp.pdb", "leaprc.ff99SB", "", 0 );
122 setxyz_from_mol( m, NULL, xyz );
123
124 mm_options( "cut=10.0" );
125 mme_init( m, NULL, "::ZZZ", xyz, NULL );
126 conjgrad( xyz, 3*m.natoms, fret, mme, 0.02, 100., 200 );
127 setmol_from_xyz( m, NULL, xyz );
128 putpdb( argv[3] + ".mm.pdb", m );

```

Once the covalent bounds are created, the the bounds matrix is modified by constraints constructed from an NMR analysis program. This particular example uses the format of the DYANA program, but NAB could be easily modified to read in other formats as well. Here are a few lines from the *mrf2.7col* file:

```

1 ARG+ QB 2 ALA QB 7.0
4 GLU- HA 93 LYS+ QB 7.0
5 GLN QB 8 LEU QQD 9.9
5 GLN HA 9 VAL QQG 6.4
85 ILE HA 92 ILE QD1 6.0
5 GLN HN 1 ARG+ O 2.0
5 GLN N 1 ARG+ O 3.0
6 ALA HN 2 ALA O 2.0
6 ALA N 2 ALA O 3.0

```

The format should be self-explanatory, with the final number giving the upper bound. Code in lines 31-69 reads these in, and translates pseudo-atom codes like "QQD" into atom names. Lines 71-93 add in chirality constraints to ensure right-handed alpha-helices: distance constraints alone do not distinguish chirality, so additions like this are often necessary. The "actual" distance geometry steps take place in line 101, first by triangle-smoothing the bounds, then by embedding them into a three-dimensional object. The structures at this point are actually generally quite bad, so "real-space" refinement is carried out in lines 103-112, and a final short molecular mechanics minimization in lines 119-126.

It is important to realize that many of the structures for the above scheme will get "stuck", and not lead to good structures for the complex. Helical proteins are especially difficult for this sort of distance geometry, since helices (or even parts of helices) start out left-handed, and it is not always possible to easily convert these to right-handed structures. For this particular example, (using different values for the *seed* in line 97), we find that about 30-40% of the structures are "acceptable", in the sense that further refinement in Amber yields good structures.

19.3. Building Larger Structures

While the DNA duplex is locally rather stiff, many DNA molecules are sufficiently long that they can be bent into a wide variety of both open and closed curves. Some examples would be

simple closed circles, supercoiled closed circles that have relaxed into circles with twists and the nucleosome core fragment where the duplex itself is wound into a short helix. This section shows how nab can be used to “wrap” DNA around a curve. Three examples are provided: the first produces closed circles with or without supercoiling, the second creates a simple model of the nucleosome core fragment and the third shows how to wind a duplex around a more arbitrary open curve specified as a set of points. The examples are fairly general but do require that the curves be relatively smooth so that the deformation from a linear duplex at each step is small.

Before discussing the examples and the general approach they use, it will be helpful to define some terminology. The helical axis of a base pair is the helical axis defined by an ideal B-DNA duplex that contains that base pair. The base pair plane is the mean plane of both bases. The origin of a base pair is at the intersection the base pair’s helical axis and its mean plane. Finally the rise is the distance between the origins of adjacent base pairs.

The overall strategy for wrapping DNA around a curve is to create the curve, find the points on the curve that contain the base pair origins, place the base pairs at these points, oriented so that their helical axes are tangent to the curve and finally rotate the base pairs so that they have the correct helical twist. In all the examples below, the points are chosen so that the rise is constant. This is by no means an absolute requirement, but it does simplify the calculations needed to locate base pairs, and is generally true for the gently bending curves these examples are designed for. In examples 1 and 2, the curve is simple, either a circle or a helix, so the points that locate the base pairs are computed directly. In addition, the bases are rotated about their original helical axes so that they have the correct helical orientation before being placed on the curve.

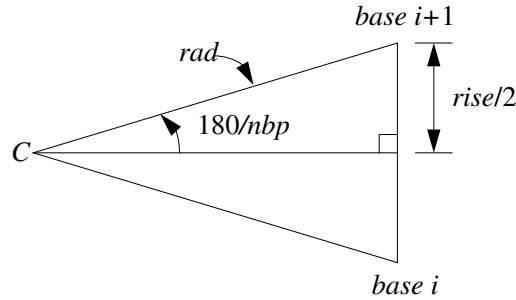
However, this method is inadequate for the more complicated curves that can be handled by example 3. Here each base is placed on the curve so that its helical axis is aligned correctly, but its helical orientation with respect to the previous base is arbitrary. It is then rotated about its helical axis so that it has the correct twist with respect to the previous base.

19.3.1. Closed Circular DNA

This section describes how to use nab to make closed circular duplex DNA with a uniform rise of 3.38 Å. Since the distance between adjacent base pairs is fixed, the radius of the circle that forms the axis of the duplex depends only on the number of base pairs and is given by this rule:

$$rad = rise / (2 \sin(180/nbp))$$

where *nbp* is the number of base pairs. To see why this is so, consider the triangle below formed by the center of the circle and the centers of two adjacent base pairs. The two long sides are radii of the circle and the third side is the rise. Since the base pairs are uniformly distributed about the circle the angle between the two radii is $360/nbp$. Now consider the right triangle in the top half of the original triangle. The angle at the center is $180/nbp$, the opposite side is $rise/2$ and *rad* follows from the definition of sin.



In addition to the radius, the helical twist which is a function of the amount of supercoiling must also be computed. In a closed circular DNA molecule, the last base of the duplex must be oriented in such a way that a single helical step will superimpose it on the first base. In circles based on ideal B-DNA, with 10 bases/turn, this requires that the number of base pairs in the duplex be a multiple of 10. Supercoiling adds or subtracts one or more whole turns. The amount of supercoiling is specified by the $\Delta linkingnumber$ which is the number of extra turns to add or subtract. If the original circle had $nbp/10$ turns, the supercoiled circle will have $nbp/10 + \Delta lk$ turns. As each turn represents 360° of twist and there are nbp base pairs, the twist between base pairs is

$$(nbp/10 + \Delta lk) \times 360/nbp$$

At this point, we are ready to create models of circular DNA. Bases are added to model in three stages. Each base pair is created using the nab builtin `wc_helix()`. It is originally in the XY plane with its center at the origin. This makes it convenient to create the DNA circle in the XZ plane. After the base pair has been created, it is rotated around its own helical axis to give it the proper twist, translated along the global X axis to the point where its center intersects the circle and finally rotated about the Y axis to move it to its final location. Since the first base pair would be both twisted about Z and rotated about Y 0°, those steps are skipped for base one. A detailed description follows the code.

```

1 // Program 9 - Create closed circular DNA.
2 #define RISE    3.38
3
4 int    b, nbp, dlk;
5 float    rad, twist, ttw;
6 molecule    m, ml;
7 matrix    matdx, mattw, matry;
8 string    sbase, abase;
9 int    getbase();
10
11 if( argc != 3 ){
12     fprintf( stderr, "usage: %s nbp dlk\n", argv[ 1 ] );
13     exit( 1 );
14 }
15
16 nbp = atoi( argv[ 2 ] );

```



```

17 if( !nbp || nbp % 10 ){
18     fprintf( stderr,
19         "%s: Num. of base pairs must be multiple of 10\\n",
20         argv[ 1 ] );
21     exit( 1 );
22 }
23
24 dlk = atoi( argv[ 3 ] );
25
26 twist = ( nbp / 10 + dlk ) * 360.0 / nbp;
27 rad = 0.5 * RISE / sin( 180.0 / nbp );
28
29 matdx = newtransform( rad, 0.0, 0.0, 0.0, 0.0, 0.0 );
30
31 m = newmolecule();
32 addstrand( m, "A" );
33 addstrand( m, "B" );
34 ttw = 0.0;
35 for( b = 1; b <= nbp; b = b + 1 ){
36
37     getbase( b, sbase, abase );
38
39     m1 = wc_helix(
40         sbase, "", "dna", abase, "",
41         "dna", 2.25, -4.96, 0.0, 0.0 );
42
43     if( b > 1 ){
44         mattw = newtransform( 0.,0.,0.,0.,0.,ttw );
45         transformmol( mattw, m1, NULL );
46     }
47
48     transformmol( matdx, m1, NULL );
49
50     if( b > 1 ){
51         matry = newtransform(
52             0.,0.,0.,0.,-360.*(b-1)/nbp,0. );
53         transformmol( matry, m1, NULL );
54     }
55
56     mergestr( m, "A", "last", m1, "sense", "first" );
57     mergestr( m, "B", "first", m1, "anti", "last" );
58     if( b > 1 ){
59         connectres( m, "A", b - 1, "O3'", b, "P" );
60         connectres( m, "B", 1, "O3'", 2, "P" );
61     }
62
63     ttw = ttw + twist;
64     if( ttw >= 360.0 )
65         ttw = ttw - 360.0;

```

19. NAB: Sample programs

```
66 }  
67  
68 connectres( m, "A", nbp, "O3'", 1, "P" );  
69 connectres( m, "B", nbp, "O3'", 1, "P" );  
70  
71 putpdb( "circ.pdb", m );  
72 putbnd( "circ.bnd", m );
```

The code requires two integer arguments which specify the number of base pairs and the Δ linkingnumber or the amount of supercoiling. Lines 11-24 process the arguments making sure that they conform to the model's assumptions. In lines 11-14, the code checks that there are exactly three arguments (the nab program's name is argument one), and exits with an error message if the number of arguments is different. Next lines 16-22 set the number of base pairs (nbp) and test to make certain it is a nonzero multiple of 10, again exiting with an error message if it is not. Finally the Δ linkingnumber(dlk) is set in line 24. The helical twist and circle radius are computed in lines 26 and 27 in accordance with the formulas developed above. Line 29 creates a transformation matrix, matdx, that is used to move each base from the global origin along the X-axis to the point where its center intersects the circle.

The circular DNA is built in the molecule variable m, which is initialized and given two strands, "A" and "B" in lines 30-32. The variable ttw in line 34 holds the total twist applied to each base pair. The molecule is created in the loop from lines 35-66. The base pair number (b) is converted to the appropriate strings specifying the two nucleotides in this pair. This is done by the function getbase(). This source of this function must be provided by the user who is creating the circles as only he or she will know the actual DNA sequence of the circle. Once the two bases are specified they are passed to the nab builtin wc_helix() which returns a single base pair in the XY plane with its center at the origin. The helical axis of this base pair is on the Z-axis with the 5'-3' direction oriented in the positive Z-direction.

One or three transformations is required to position this base in its correct place in the circle. It must be rotated about the Z-axis (its helical axis) so that it is one additional unit of twist beyond the previous base. This twist is done in lines 43-46. Since the first base needs 0 twist, this step is skipped for it. In line 48, the base pair is moved in the positive direction along the X-axis to place the base pair's origin on the circle. Finally, the base pair is rotated about the Y-axis in lines 50-54 to bring it to its proper position on the circle. Again, since this rotation is 0 for base 1, this step is also skipped for the first base.

In lines 56-57, the newly positioned base pair in m1 is added to the growing molecule in m. Note that since the two strands of DNA are antiparallel, the "sense" strand of m1 is added after the last base of the "A" strand of m and the "anti" strand of m1 is added before the first base of the "B" strand of m. For all but the first base, the newly added residues are bonded to the residues they follow (or precede). This is done by the two calls to connectres() in lines 59-60. Again, due to the antiparallel nature of DNA, the new residue in the "A" strand is residue b, but is residue 1 in the "B" strand. In line 63-65, the total twist (ttw) is updated and adjusted to keep in in the range [0,360). After all base pairs have been added the loop exits.

After the loop exit, since this is a *closed* circular molecule the first and last bases of each strand must be bonded and this is done with the two calls to connectres() in lines 67-68. The last step is to save the molecule's coordinates and connectivity in lines 71-72. The nab builtin putpdb() writes the coordinate information in PDB format to the file "circ.pdb" and the nab

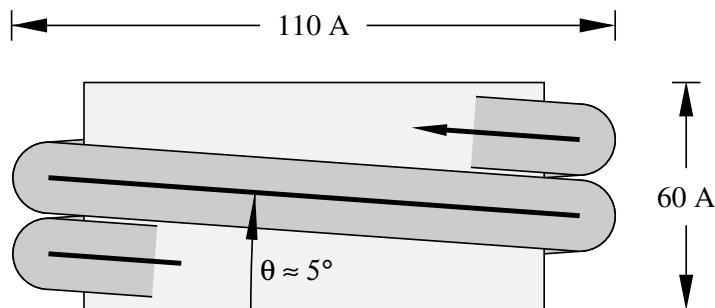
builtin `putbnd()` saves the bonding as pairs of integers, one pair/line in the file "circ.bnd", where each integer in a pair refers to an ATOM record in the previously written PDB file.

19.3.2. Nucleosome Model

While the DNA duplex is locally rather stiff, many DNA molecules are sufficiently long that they can be bent into a wide variety of both open and closed curves. Some examples would be simple closed circles, supercoiled closed circles that have relaxed into circles with twists, and the nucleosome core fragment, where the duplex itself is wound into a short helix.

The overall strategy for wrapping DNA around a curve is to create the curve, find the points on the curve that contain the base pair origins, place the base pairs at these points, oriented so that their helical axes are tangent to the curve, and finally rotate the base pairs so that they have the correct helical twist. In the example below, the simplifying assumption is made that the rise is constant at 3.38 Å.

The nucleosome core fragment is composed of duplex DNA wound in a left handed helix around a central protein core. A typical core fragment has about 145 base pairs of duplex DNA forming about 1.75 superhelical turns. Measurements of the overall dimensions of the core fragment indicate that there is very little space between adjacent wraps of the duplex. A side view of a schematic of core particle is shown below.



Computing the points at which to place the base pairs on a helix requires us to spiral an inelastic wire (representing the helical axis of the bent duplex) around a cylinder (representing the protein core). The system is described by four numbers of which only three are independent. They are the number of base pairs n , the number of turns it makes around the protein core t , the “winding” angle θ (which controls how quickly the the helix advances along the axis of the core) and the helix radius r . Both the the number of base pairs and the number of turns around the core can be measured. The leaves two choices for the third parameter. Since the relationship of the winding angle to the overall particle geometry seems more clear than that of the radius, this code lets the user specify the number of turns, the number of base pairs and the winding angle, then computes the helical radius and the displacement along the helix axis for each base pair:

$$d = 3.38 \sin(\theta); \phi = 360t/(n-1) \quad (19.1)$$

$$r = \frac{3.38(n-1) \cos(\theta)}{2\pi t} \quad (19.2)$$

where d and ϕ are the displacement along and rotation about the protein core axis for each base pair.

These relationships are easily derived. Let the nucleosome core particle be oriented so that its helical axis is along the global Y-axis and the lower cap of the protein core is in the XZ plane. Consider the circle that is the projection of the helical axis of the DNA duplex onto the XZ plane. As the duplex spirals along the core particle it will go around the circle t times, for a total rotation of $360t$. The duplex contains $(n-1)$ steps, resulting in $360t/(n-1)$ of rotation between successive base pairs.

```

1 // Program 10. Create simple nucleosome model.
2 #define PI 3.141593
3 #define RISE 3.38
4 #define TWIST 36.0
5 int      b, nbp; int getbase();
6 float    nt, theta, phi, rad, dy, ttw, len, plen, side;
7 molecule m, m1;
8 matrix   matdx, matrx, maty, matry, mattw;
9 string    sbase, abase;
10
11 nt = atof( argv[ 2 ] ); // number of turns
12 nbp = atoi( argv[ 3 ] ); // number of base pairs
13 theta = atof( argv[ 4 ] ); // winding angle
14
15 dy = RISE * sin( theta );
16 phi = 360.0 * nt / ( nbp-1 );
17 rad = (( nbp-1 ) * RISE * cos( theta )) / ( 2 * PI * nt );
18
19 matdx = newtransform( rad, 0.0, 0.0, 0.0, 0.0, 0.0 );
20 matrx = newtransform( 0.0, 0.0, 0.0, -theta, 0.0, 0.0 );
21
22 m = newmolecule();
23 addstrand( m, "A" ); addstrand( m, "B" );
24 ttw = 0.0;
25 for( b = 1; b <= nbp; b = b + 1 ){
26     getbase( b, sbase, abase );
27     m1 = wc_helix( sbase, "", "dna", abase, "", "dna",
28         2.25, -4.96, 0.0, 0.0 );
29     mattw = newtransform( 0., 0., 0., 0., 0., ttw );
30     transformmol( mattw, m1, NULL );
31     transformmol( matrx, m1, NULL );
32     transformmol( matdx, m1, NULL );
33     maty = newtransform( 0., dy*(b-1), 0., 0., -phi*(b-1), 0. );
34     transformmol( maty, m1, NULL );
35
36     mergestr( m, "A", "last", m1, "sense", "first" );
37     mergestr( m, "B", "first", m1, "anti", "last" );
38     if( b > 1 ){
39         connectres( m, "A", b - 1, "O3'", b, "P" );
40         connectres( m, "B", 1, "O3'", 2, "P" );

```

```

41     }
42     ttw += TWIST; if( ttw >= 360.0 ) ttw -= 360.0;
43 }
44 putpdb( "nuc.pdb", m );

```

Finding the radius of the superhelix is a little tricky. In general a single turn of the helix will not contain an integral number of base pairs. For example, using typical numbers of 1.75 turns and 145 base pairs requires 82.9 base pairs to make one turn. An approximate solution can be found by considering the ideal superhelix that the DNA duplex is wrapped around. Let L be the arc length of this helix. Then $L\cos(\theta)$ is the arc length of its projection into the XZ plane. Since this projection is an overwound circle, L is also equal to $2\pi rt$, where t is the number of turns and r is the unknown radius. Now L is not known but is approximately $3.38(n - 1)$. Substituting and solving for r gives Eq. 19.2.

The resulting nab code is shown in Program 2. This code requires three arguments—the number of turns, the number of base pairs and the winding angle. In lines 15-17, the helical rise (dy), twist (phi) and radius (rad) are computed according to the formulas developed above.

Two constant transformation matrices, matdx and matrix are created in lines 19-20. matdx is used to move the newly created base pair along the X-axis to the circle that is the helix's projection onto the XZ plane. matrix is used to rotate the new base pair about the X-axis so it will be tangent to the local helix of spirally wound duplex. The model of the nucleosome will be built in the molecule m which is created and given two strands "A" and "B" in line 23. The variable ttw will hold the total local helical twist for each base pair.

The molecule is created in the loop in lines 25-43. The user specified function getbase() takes the number of the current base pair (b) and returns two strings that specify the actual nucleotides to use at this position. These two strings are converted into a single base pair using the nab builtin wc_helix(). The new base pair is in the XY plane with its origin at the global origin and its helical axis along Z oriented so that the 5'-3' direction is positive.

Each base pair must be rotated about its Z-axis so that when it is added to the global helix it has the correct amount of helical twist with respect to the previous base. This rotation is performed in lines 29-30. Once the base pair has the correct helical twist it must be rotated about the X-axis so that its local origin will be tangent to the global helical axes (line 31).

The properly-oriented base is next moved into place on the global helix in two stages in lines 32-34. It is first moved along the X-axis (line 32) so it intersects the circle in the XZ plane that is projection of the duplex's helical axis. Then it is simultaneously rotated about and displaced along the global Y-axis to move it to final place in the nucleosome. Since both these movements are with respect to the same axis, they can be combined into a single transformation.

The newly positioned base pair in m1 is added to the growing molecule in m using two calls to the nab builtin mergestr(). Note that since the two strands of a DNA duplex are antiparallel, the base of the "sense" strand of molecule m1 is added *after* the last base of the "A" strand of molecule m and the base of the "anti" strand of molecule m1 is *before* the first base of the "B" strand of molecule m. For all base pairs except the first one, the new base pair must be bonded to its predecessor. Finally, the total twist (ttw) is updated and adjusted to remain in the interval [0,360) in line 42. After all base pairs have been created, the loop exits, and the molecule is written out. The coordinates are saved in PDB format using the nab builtin putpdb().

19.4. Wrapping DNA Around a Path

This last code develops two nab programs that are used together to wrap B-DNA around a more general open curve specified as a cubic spline through a set of points. The first program takes the initial set of points defining the curve and interpolates them to produce a new set of points with one point at the location of each base pair. The new set of points always includes the first point of the original set but may or may not include the last point. These new points are read by the second program which actually bends the DNA.

The overall strategy used in this example is slightly different from the one used in both the circular DNA and nucleosome codes. In those codes it was possible to directly compute both the orientation and position of each base pair. This is not possible in this case. Here only the location of the base pair's origin can be computed directly. When the base pair is placed at that point its helical axis will be tangent to the curve and point in the right direction, but its rotation about this axis will be arbitrary. It will have to be rotated about its new helical axis to give the proper amount of helical twist to stack it properly on the previous base. Now if the helical twist of a base pair is determined with respect to the previous base pair, either the first base pair is left in an arbitrary orientation, or some other way must be devised to define the helical of it. Since this orientation will depend both on the curve and its ultimate use, this code leaves this task to the user with the result that the helical orientation of the first base pair is undefined.

19.4.1. Interpolating the Curve

This section describes the code that finds the base pair origins along the curve. This program takes an ordered set of points

$$p_1, p_2, \dots, p_n$$

and interpolates it to produce a new set of points

$$np_1, np_2, \dots, np_m$$

such that the distance between each np_i and np_{i+1} is constant, in this case equal to 3.38 which is the rise of an ideal B-DNA duplex. The interpolation begins by setting np_1 to p_1 and continues through the p_i until a new point np_m has been found that is within the constant distance to p_n without having gone beyond it.

The interpolation is done via `spline()` [45] and `splint()`, two routines that perform a cubic spline interpolation on a tabulated function

$$y_i = f(x_i)$$

In order for `spline()/splint()` to work on this problem, two things must be done. These functions work on a table of (x_i, y_i) pairs, of which we have only the y_i . However, since the only requirement imposed on the x_i is that they be monotonically increasing we can simply use the sequence $1, 2, \dots, n$ for the x_i , producing the producing the table (i, y_i) . The second difficulty is that `spline()/splint()` interpolate along a one dimensional curve but we need an interpolation along a three dimensional curve. This is solved by creating three different splines, one for each of the three dimensions.

spline()/splint() perform the interpolation in two steps. The function spline() is called first with the original table and computes the value of the second derivative at each point. In order to do this, the values of the second derivative at two points must be specified. In this code these points are the first and last points of the table, and the values chosen are 0 (signified by the unlikely value of 1e30 in the calls to spline()). After the second derivatives have been computed, the interpolated values are computed using one or more calls to splint().

What is unusual about this interpolation is that the points at which the interpolation is to be performed are unknown. Instead, these points are chosen so that the distance between each point and its successor is the constant value RISE, set here to 3.38 which is the rise of an ideal B-DNA duplex. Thus, we have to search for the points and most of the code is devoted to doing this search. The details follow the listing.

```

1 // Program 11 - Build DNA along a curve
2 #define RISE      3.38
3
4 #define EPS 1e-3
5 #define APPROX(a,b) (fabs((a)-(b))<=EPS)
6 #define MAXI      20
7
8 #define MAXPTS 150
9 int npts;
10 float a[ MAXPTS ];
11 float x[ MAXPTS ], y[ MAXPTS ], z[ MAXPTS ];
12 float x2[ MAXPTS ], y2[ MAXPTS ], z2[ MAXPTS ];
13 float tmp[ MAXPTS ];
14
15 string line;
16
17 int i, li, ni;
18 float dx, dy, dz;
19 float la, lx, ly, lz, na, nx, ny, nz;
20 float d, tfrac, frac;
21
22 int spline();
23 int splint();
24
25 for( npts = 0; line = getline( stdin ); ){
26     npts = npts + 1;
27     a[ npts ] = npts;
28     sscanf( line, "%lf %lf %lf",
29         x[ npts ], y[ npts ], z[ npts ] );
30 }
31
32 spline( a, x, npts, 1e30, 1e30, x2, tmp );
33 spline( a, y, npts, 1e30, 1e30, y2, tmp );
34 spline( a, z, npts, 1e30, 1e30, z2, tmp );
35
36 li = 1; la = 1.0; lx = x[1]; ly = y[1]; lz = z[1];

```

19. NAB: Sample programs

```

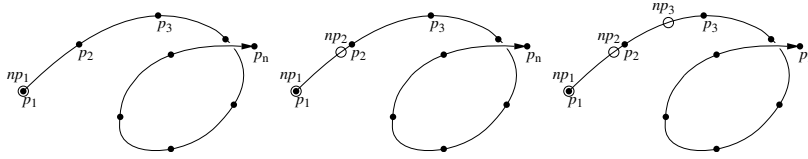
37 printf( "%8.3f %8.3f %8.3f\\n", lx, ly, lz );
38
39 while( li < npts ){
40     ni = li + 1;
41     na = a[ ni ];
42     nx = x[ ni ]; ny = y[ ni ]; nz = z[ ni ];
43     dx = nx - lx; dy = ny - ly; dz = nz - lz;
44     d = sqrt( dx*dx + dy*dy + dz*dz );
45     if( d > RISE ){
46         tfrac = frac = .5;
47         for( i = 1; i <= MAXI; i = i + 1 ){
48             na = la + tfrac * ( a[ni] - la );
49             splint( a, x, x2, npts, na, nx );
50             splint( a, y, y2, npts, na, ny );
51             splint( a, z, z2, npts, na, nz );
52             dx = nx - lx; dy = ny - ly; dz = nz - lz;
53             d = sqrt( dx*dx + dy*dy + dz*dz );
54             frac = 0.5 * frac;
55             if( APPROX( d, RISE ) )
56                 break;
57             else if( d > RISE )
58                 tfrac = tfrac - frac;
59             else if( d < RISE )
60                 tfrac = tfrac + frac;
61         }
62         printf( "%8.3f %8.3f %8.3f\\n", nx, ny, nz );
63     }else if( d < RISE ){
64         li = ni;
65         continue;
66     }else if( d == RISE ){
67         printf( "%8.3f %8.3f %8.3f\\n", nx, ny, nz );
68         li = ni;
69     }
70     la = na;
71     lx = nx; ly = ny; lz = nz;
72 }

```

Execution begins in line 25 where the points are read from stdin one point or three number-s/line and stored in the three arrays x, y and z. The independent variable for each spline, stored in the array a is created at this time holding the numbers 1 to npts. The second derivatives for the three splines, one each for interpolation along the X, Y and Z directions are computed in lines 32-34. Each call to spline() has two arguments set to 1e30 which indicates that the second derivative values should be 0 at the first and last points of the table. The first point of the interpolated set is set to the first point of the original set and written to stdout in lines 36-37.

The search that finds the new points is lines 39-72. To see how it works consider the figure below. The dots marked p_1, p_2, \dots, p_n correspond to the original points that define the spline. The circles marked np_1, np_2, np_3 represent the new points at which base pairs will be placed. The curve is a function of the parameter a , which as it ranges from 1 to $npts$ sweeps out the

curve from (x_1, y_1, z_1) to $(x_{npts}, y_{npts}, z_{npts})$. Since the original points will in general not be the correct distance apart we have to find new points by interpolating between the original points.



The search works by first finding a point of the original table that is at least RISE distance from the last point found. If the last point of the original table is not far enough from the last point found, the search loop exits and the program ends. However, if the search does find a point in the original table that is at least RISE distance from the last point found, it starts an interpolation loop in lines 47-61 to zero on the best value of a that will produce a new point that is the correct distance from the previous point. After this point is found, the new point becomes the last point and the loop is repeated until the original table is exhausted.

The main search loop uses `li` to hold the index of the point in the original table that is closest to, but does not pass, the last point found. The loop begins its search for the next point by assuming it will be before the next point in the original table (lines 40-42). It computes the distance between this point (nx, ny, nz) and the last point (lx, ly, lz) in lines 43-44 and then takes one of three actions depending if the distance is greater than RISE (lines 46-62), less than RISE (lines 64-65) or equal to RISE (lines 67-68).

If this distance is greater than RISE, then the desired point is between the last point found which is the point generated by `la` and the point corresponding to `a[ni]`. Lines 46-61 perform a bisection of the interval $[la, a[ni]]$, a process that splits this interval in half, determines which half contains the desired point, then splits that half and continues in this fashion until the either the distance between the last and new points is close enough as determined by the macro `APPROX()` or `MAXI` subdivisions have been at made, in which case the new point is taken to be the point computed after the last subdivision. After the bisection the new point is written to `stdout` (line 62) and execution skips to line 70-71 where the new values `na` and (nx, ny, nz) become the last values `la` and (lx, ly, lz) and then back to the top of the loop to continue the interpolation. The macro `APPROX()` defined in line 4, tests to see if the absolute value of the difference between the current distance and RISE is less than `EPS`, defined in line 3 as 10^{-3} . This more complicated test is used instead of simply testing for equality because floating point arithmetic is inexact, which means that while it will get close to the target distance, it may never actually reach it.

If the distance between the last and candidate points is less than RISE, the desired point lies beyond the point at `a[ni]`. In this case the action in lines 64-65 is performed which advances the candidate point to `li+2` then goes back to the top of the loop (line 38) and tests to see that this index is still in the table and if so, repeats the entire process using the point corresponding to `a[li+2]`. If the points are close together, this step may be taken more than once to look for the next candidate at `a[li+2]`, `a[li+3]`, etc. Eventually, it will find a point that is RISE beyond the last point at which case it interpolates or it runs out points, indicating that the next point lies beyond the last point in the table. If this happens, the last point found, becomes the last point of the new set and the process ends.

The last case is if the distance between the last point found and the point at `a[ni]` is exactly equal to RISE. If it is, the point at `a[ni]` becomes the new point and `li` is updated to `ni`. (lines

67-68). Then lines 70-71 are executed to update *la* and (*lx,ly,lz*) and then back to the top of the loop to continue the process.

19.4.2. Driver Code

This section describes the main routine or driver of the second program which is the actual DNA bender. This routine reads in the points, then calls `putdna()` (described in the next section) to place base pairs at each point. The points are either read from `stdin` or from the file whose name is the second command line argument. The source of the points is determined in lines 8-18, being `stdin` if the command line contained a single arguments or in the second argument if it was present. If the argument count was greater than two, the program prints an error message and exits. The points are read in the loop in lines 20-26. Any line with a `#` in column 1 is a comment and is ignored. All other lines are assumed to contain three numbers which are extracted from the string, line and stored in the point array `pts` by the `nab` builtin `sscanf()` (lines 23-24). The number of points is kept in `npts`. Once all points have been read, the loop exits and the point file is closed if it is not `stdin`. Finally, the points are passed to the function `putdna()` which will place a base pair at each point and save the coordinates and connectivity of the resulting molecule in the pair of files `dna.path.pdb` and `dna.path.bnd`.

```

1 // Program 12 - DNA bender main program
2 string      line;
3 file        pf;
4 int          npts;
5 point        pts[ 5000 ];
6 int          putdna();
7
8 if( argc == 1 )
9     pf = stdin;
10 else if( argc > 2 ){
11     fprintf( stderr, "usage: %s [ path-file ]\\n",
12         argv[ 1 ], argv[ 2 ] );
13     exit( 1 );
14 }else if( !( pf = fopen( argv[ 2 ], "r" ) ) ){
15     fprintf( stderr, "%s: can't open %s\\n",
16         argv[ 1 ], argv[ 2 ] );
17     exit( 1 );
18 }
19
20 for( npts = 0; line = getline( pf ); ){
21     if( substr( line, 1, 1 ) != "#" ){
22         npts = npts + 1;
23         sscanf( line, "%lf %lf %lf",
24             pts[ npts ].x, pts[ npts ].y, pts[ npts ].z );
25     }
26 }
27
28 if( pf != stdin )
29     fclose( pf );

```

```

30
31 putdna( "dna.path", pts, npts );

```

19.4.3. Wrap DNA

Every nab molecule contains a frame, a movable handle that can be used to position the molecule. A frame consists of three orthogonal unit vectors and an origin that can be placed in an arbitrary position and orientation with respect to its associated molecule. When the molecule is created its frame is initialized to the unit vectors along the global X, Y and Z axes with the origin at (0,0,0).

nab provides three operations on frames. They can be defined by atom expressions or absolute points (`setframe()` and `setframep()`), one frame can be aligned or superimposed on another (`alignframe()`) and a frame can be placed at a point on an axis (`axis2frame()`). A frame is defined by specifying its origin, two points that define its X direction and two points that define its Y direction. The Z direction is $X \times Y$. Since it is convenient to not require the original X and Y be orthogonal, both frame creation builtins allow the user to specify which of the original X or Y directions is to be the true X or Y direction. If X is chosen then Y is recreated from $Z \times X$; if Y is chosen then X is recreated from $Y \times Z$.

When the frame of one molecule is aligned on the frame of another, the frame of the first molecule is transformed to superimpose it on the frame of the second. At the same time the coordinates of the first molecule are also transformed to maintain their original position and orientation with respect to their own frame. In this way frames provide a way to precisely position one molecule with respect to another. The frame of a molecule can also be positioned on an axis defined by two points. This is done by placing the frame's origin at the first point of the axis and aligning the frame's Z-axis to point from the first point of the axis to the second. After this is done, the orientation of the frame's X and Y vectors about this axis is undefined.

Frames have two other properties that need to be discussed. Although the builtin `alignframe()` is normally used to position two molecules by superimposing their frames, if the second molecule (represented by the second argument to `alignframe()`) has the special value NULL, the first molecule is positioned so that its frame is superimposed on the global X, Y and Z axes with its origin at (0,0,0). The second property is that when nab applies a transformation to a molecule (or just a subset of its atoms), only the atomic coordinates are transformed. The frame's origin and its orthogonal unit vectors remain untouched. While this may at first glance seem odd, it makes possible the following three stage process of setting the molecule's frame, aligning that frame on the *global* frame, then transforming the molecule with respect to the global axes and origin which provides a convenient way to position and orient a molecule's frame at arbitrary points in space. With all this in mind, here is the source to `putdna()` which bends a B-DNA duplex about an open space curve.

```

1 // Program 13 - place base pairs on a curve.
2 point      s_ax[ 4 ];
3 int        getbase();
4
5 int putdna( string mname, point pts[ 1 ], int npts )
6 {

```

19. NAB: Sample programs

```

7   int p;
8   float tw;
9   residue r;
10  molecule m, m_path, m_ax, m_bp;
11  point p1, p2, p3, p4;
12  string sbase, abase;
13  string aex;
14  matrix mat;
15
16  m_ax = newmolecule();
17  addstrand( m_ax, "A" );
18  r = getresidue( "AXS", "axes.rlb" );
19  addresidue( m_ax, "A", r );
20  setxyz_from_mol( m_ax, NULL, s_ax );
21
22  m_path = newmolecule();
23  addstrand( m_path, "A" );
24
25  m = newmolecule();
26  addstrand( m, "A" );
27  addstrand( m, "B" );
28
29  for( p = 1; p < npts; p = p + 1 ){
30      setmol_from_xyz( m_ax, NULL, s_ax );
31      setframe( 1, m_ax,
32              "::ORG", "::ORG", "::SXT", "::ORG", "::CYT" );
33      axis2frame( m_path, pts[ p ], pts[ p + 1 ] );
34      alignframe( m_ax, m_path );
35      mergestr( m_path, "A", "last", m_ax, "A", "first" );
36      if( p > 1 ){
37          setpoint( m_path, sprintf( "A:%d:CYT",p-1 ), p1 );
38          setpoint( m_path, sprintf( "A:%d:ORG",p-1 ), p2 );
39          setpoint( m_path, sprintf( "A:%d:ORG",p ), p3 );
40          setpoint( m_path, sprintf( "A:%d:CYT",p ), p4 );
41          tw = 36.0 - torsionp( p1, p2, p3, p4 );
42          mat = rot4p( p2, p3, tw );
43          aex = sprintf( ":%d:", p );
44          transformmol( mat, m_path, aex );
45          setpoint( m_path, sprintf( "A:%d:ORG",p ), p1 );
46          setpoint( m_path, sprintf( "A:%d:SXT",p ), p2 );
47          setpoint( m_path, sprintf( "A:%d:CYT",p ), p3 );
48          setframep( 1, m_path, p1, p1, p2, p1, p3 );
49      }
50
51      getbase( p, sbase, abase );
52      m_bp = wc_helix( sbase, "", "dna",
53                      abase, "", "dna",
54                      2.25, -5.0, 0.0, 0.0 );
55      alignframe( m_bp, m_path );

```

```

56     mergestr( m, "A", "last", m_bp, "sense", "first" );
57     mergestr( m, "B", "first", m_bp, "anti", "last" );
58     if( p > 1 ){
59         connectres( m, "A", p - 1, "O3'", p, "P" );
60         connectres( m, "B", 1, "P", 1, "O3'" );
61     }
62 }
63
64 putpdb( mname + ".pdb", m );
65 putbnd( mname + ".bnd", m );
66 };

```

putdna() takes three arguments—name, a string that will be used to name the PDB and bond files that hold the bent duplex, pts an array of points containing the origin of each base pair and npts the number of points in the array. putdna() uses four molecules. m_ax holds a small artificial molecule containing four atoms that is a proxy for the some of the frame's used placing the base pairs. The molecule m_path will eventually hold one copy of m_ax for each point in the input array. The molecule m_bp holds each base pair after it is created by wc_helix() and m will eventually hold the bent dna. Once again the function getbase() (to be defined by the user) provides the mapping between the current point (p) and the nucleotides required in the base pair at that point.

Execution of putdna() begins in line 16 with the creation of m_ax. This molecule is given one strand "A", into which is added one copy of the special residue AXS from the standard nab residue library "axes.rlb" (lines 17-19). This residue contains four atoms named ORG, SXT, CYT and NZT. These atoms are placed so that ORG is at (0,0,0) and SXT, CYT and NZT are 1o along the X, Y and Z axes respectively. Thus the residue AXS has the exact geometry as the molecules initial frame—three unit vectors along the standard axes centered on the origin. The initial coordinates of m_ax are saved in the point array s_ax. The molecules m_path and m are created in lines 22-23 and 25-27 respectively.

The actual DNA bending occurs in the loop in lines 29-62. Each base pair is added in a two stage process that uses m_ax to properly orient the frame of m_path, so that when the frame of new the base pair in m_bp is aligned on the frame of m_path, the new base pair will be correctly positioned on the curve.

Setting up the frame is done in lines 30-49. The process begins by restoring the original coordinates of m_ax (line 30), so that the the atom ORG is at (0,0,0) and SXT, CYT and NZT are each 1o along the global X, Y and Z axes. These atoms are then used to redefine the frame of m_ax (line 32-33) so that it is equal to the three standard unit vectors at the global origin. Next the frame of m_path is aligned so that its origin is at pts[p] and its Z-axis points from pts[p] to pts[p+1] (line 34). The call to alignframe() in line 34 transforms m_ax to align its frame on the frame of m_path, which has the effect of moving m_ax so that the atom ORG is at pts[p] and the ORG—NZT vector points towards pts[p+1]. A copy of the newly positioned m_ax is merged into m_path in line 35. The result of this process is that each time around the loop, m_path gets a new residue that resembles a coordinate frame located at the point the new base pair is to be added.

When nab sets a frame from an axis, the orientation of its X and Y vectors is arbitrary. While this does not matter for the first base pair for which any orientation is acceptable, it does matter

for the second and subsequent base pairs which must be rotated about their Z axis so that they have the proper helical twist with respect to the previous base pair. This rotation is done by the code in lines 37-48. It does this by considering the torsion angle formed by the four atoms—CYT and ORG of the previous AXS residue and ORG and CYT of the current AXS residue. The coordinates of these points are determined in lines 37-40. Since this torsion angle is a marker for the helical twist between pairs of the bent duplex, it must be 36.0°. The amount of rotation required to give it the correct twist is computed in line 41. A transformation matrix that will rotate the new AXS residue about the ORG—ORG axis by this amount is created in line 42, the atom expression that names the AXS residue is created in line 43 and the residue rotated in line 44. Once the new residue is given the correct twist the frame `m_path` is moved to the new residue in lines 45-48.

The base pair is added in lines 51-60. The user defined function `getbase()` converts the point number (`p`) into the names of the nucleotides needed for this base pair which is created by the `nab` builtin `wc_helix()`. It is then placed on the curve in the correct orientation by aligning its frame on the frame of `m_path` that we have just created (line 55). The new pair is merged into `m` and bonded with the previous base pair if it exists. After the loop exits, the bend DNA duplex coordinates are saved as a PDB file and the connectivity as a `bnd` file in the calls to `putpdb()` and `putbnd()` in lines 64-65, whereupon `putdna()` returns to the caller.

19.5. Other examples

There are several additional pedagogical (and useful!) examples in `$AMBERHOME/examples`. These can be consulted to gain ideas of how some useful molecular manipulation programs can be constructed.

- The *peptides* example was created by Paul Beroza to construct peptides with given backbone torsion angles. The idea is to call `linkprot` to create a peptide in an extended conformation, then to set frames and do rotations to construct the proper torsions. This can be used as just a stand-alone program to perform this task, or as a source for ideas for constructing similar functionality in other `nab` programs.
- The *suppose* example was created by Jarrod Smith to provide a driver to carry out rms-superpositions. It has a man page that shows how to use it.

Bibliography

- [1] Pettersen, E.F.; Goddard, T.D.; Huang, C.C.; Couch, G.S.; Greenblatt, D.M.; Meng, E.C.; Ferrin, T.E. UCSF Chimera - A visualization system for exploratory research and analysis. *J. Comput. Chem.*, **2004**, 25, 1605–1612.
- [2] Harvey, S.; McCammon, J.A. *Dynamics of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, 1987.
- [3] Leach, A.R. *Molecular Modelling. Principles and Applications, Second Edition*. Prentice-Hall, Harlow, England, 2001.
- [4] Allen, M.P.; Tildesley, D.J. *Computer Simulation of Liquids*. Clarendon Press, Oxford, 1987.
- [5] Frenkel, D.; Smit, B. *Understanding Molecular Simulation: From Algorithms to Applications. Second edition*. Academic Press, San Diego, 2002.
- [6] van Gunsteren, W.F.; Weiner, P.K.; Wilkinson, A.J. eds. *Computer Simulations of Biomolecular Systems, Vol. 3*. ESCOM Science Publishers, Leiden, 1997.
- [7] Pratt, L.R.; Hummer, G. eds. *Simulation and Theory of Electrostatic Interactions in Solution*. American Institute of Physics, Melville, NY, 1999.
- [8] Becker, O.; MacKerell, A.D.; Roux, B.; Watanabe, M. eds. *Computational Biochemistry and Biophysics*. Marcel Dekker, New York, 2001.
- [9] Skjevik, Åge A.; Madej, Benjamin D.; Walker, R.C.; Teigen, K. Lipid11: A comprehensive modular lipid force field for amber. *In press*, **2012**.
- [10] Hornak, V.; Abel, R.; Okur, A.; Strockbine, B.; Roitberg, A.; Simmerling, C. Comparison of multiple Amber force fields and development of improved. *Proteins*, **2006**, 65, 712–725.
- [11] Perez, A.; Marchan, I.; Svozil, D.; Sponer, J.; Cheatham, T.E.; Laughton, C.A.; Orozco, M. Refinement of the AMBER Force Field for Nucleic Acids: Improving the Description of alpha/gamma Conformers. *Biophys. J.*, **2007**, 92, 3817–3829.
- [12] Banáš, P.; Hollas, D.; Zgarbová, M.; Jurecka, P.; Orozco, M.; Cheatham, T.E. III; Šponer, J.; Otyepka, M. Performance of molecular mechanics force fields for RNA simulations: Stability of UUCG and GNRA hairpins. *J. Chem. Theory. Comput.*, **2010**, 6, 3836–3849.

BIBLIOGRAPHY

- [13] Zgarbova, M.; Otyepka, M.; Sponer, J.; Mladek, A.; Banas, P.; Cheatham, T.E.; Jurecka, P. Refinement of the Cornell et al. Nucleic Acids Force Field Based on Reference Quantum Chemical Calculations of Glycosidic Torsion Profiles. *J. Chem. Theory Comput.*, **2011**, 7, 2886–2902.
- [14] Joung, S.; Cheatham, T.E. III. Determination of alkali and halide monovalent ion parameters for use in explicitly solvated biomolecular simulations. *J. Phys. Chem. B*, **2008**, 112, 9020–9041.
- [15] Joung, I.S.; Cheatham, T.E. III. Molecular dynamics simulations of the dynamic and energetic properties of alkali and halide ions using water-model-specific ion parameters. *J. Phys. Chem. B*, **2009**, 113, 13279–13290.
- [16] Lindorff-Larsen, K.; Piana, S.; Palmo, K.; Maragakis, P.; Klepeis, J.; Dror, R.O.; Shaw, D.E. Improved side-chain torsion potentials for the Amber ff99SB protein force field. *Proteins*, **2010**, 78, 1950–1958.
- [17] Li, D.W.; Brüschweiler, R. NMR-based protein potentials. *Angew. Chem. Int. Ed.*, **2010**, 49, 6778–6780.
- [18] Yildirim, I.; Stern, H.A.; Kennedy, S.D.; Tubbs, J.D.; Turner, D.H. Reparameterization of RNA chi Torsion Parameters for the AMBER Force Field and Comparison to NMR Spectra for Cytidine and Uridine. *J. Chem. Theory Comput.*, **2010**, 6, 1520–1531.
- [19] Yildirim, I.; Stern, H.A.; Tubbs, J.D.; Kennedy, S.D.; Turner, D.H. Benchmarking AMBER Force Fields for RNA: Comparisons to NMR Spectra for Single-Stranded r(GACC) Are Improved by Revised chi Torsions. *J. Phys. Chem. B*, **2011**, 115, 9261–9270.
- [20] Ren, P.; Ponder, J.W. Consistent treatment of inter- and intramolecular polarization in molecular mechanics calculations. *J. Comput. Chem.*, **2002**, 23, 1497–1506.
- [21] Ren, P.Y.; Ponder, J.W. Polarizable atomic multipole water model for molecular mechanics simulation. *J. Phys. Chem. B*, **2003**, 107, 5933–5947.
- [22] Ren, P.; Ponder, J.W. Temperature and pressure dependence of the AMOEBA water model. *J. Phys. Chem. B*, **2004**, 108, 13427–13437.
- [23] Ren, P.Y.; Ponder, J.W. Tinker polarizable atomic multipole force field for proteins. *to be published.*, **2006**.
- [24] Ponder, J.W.; Wu, C.; Ren, P.; Pande, V.S.; Chodera, J.D.; Schieders, M.J.; Haque, I.; Mobley, D.L.; Lambrecht, D.S.; DiStasio, R.A. Jr.; Head-Gordon, M.; Clark, G.N.I.; Johnson, M.E.; Head-Gordon, T. Current status of the AMOEBA polarizable force field. *J. Phys. Chem. B*, **2010**, 114, 2549–2564.
- [25] Duan, Y.; Wu, C.; Chowdhury, S.; Lee, M.C.; Xiong, G.; Zhang, W.; Yang, R.; Cieplak, P.; Luo, R.; Lee, T. A point-charge force field for molecular mechanics simulations of proteins based on condensed-phase quantum mechanical calculations. *J. Comput. Chem.*, **2003**, 24, 1999–2012.

- [26] Lee, M.C.; Duan, Y. Distinguish protein decoys by using a scoring function based on a new Amber force field, short molecular dynamics simulations, and the generalized Born solvent model. *Proteins*, **2004**, *55*, 620–634.
- [27] Yang, L.; Tan, C.; Hsieh, M.-J.; Wang, J.; Duan, Y.; Cieplak, P.; Caldwell, J.; Kollman, P.A.; Luo, R. New-generation Amber united-atom force field. *J. Phys. Chem. B*, **2006**, *110*, 13166–13176.
- [28] Cieplak, P.; Dupradeau, F.-Y.; Duan, Y.; Wang, J. Polarization effects in molecular mechanical force fields. *J. Phys.: Condens. Matter*, **2009**, *21*, 333102.
- [29] Cieplak, P.; Caldwell, J.; Kollman, P. Molecular mechanical models for organic and biological systems going beyond the atom centered two body additive approximation: Aqueous solution free energies of methanol and N-methyl acetamide, nucleic acid base, and amide hydrogen bonding and chloroform/water partition coefficients of the nucleic acid bases. *J. Comput. Chem.*, **2001**, *22*, 1048–1057.
- [30] Wang, Z.-X.; Zhang, W.; Wu, C.; Lei, H.; Cieplak, P.; Duan, Y. Strike a Balance: Optimization of backbone torsion parameters of AMBER polarizable force field for simulations of proteins and peptides. *J. Comput. Chem.*, **2006**, *27*, 781–790.
- [31] Dixon, R.W.; Kollman, P.A. Advancing beyond the atom-centered model in additive and nonadditive molecular mechanics. *J. Comput. Chem.*, **1997**, *18*, 1632–1646.
- [32] Meng, E.; Cieplak, P.; Caldwell, J.W.; Kollman, P.A. Accurate solvation free energies of acetate and methylammonium ions calculated with a polarizable water model. *J. Am. Chem. Soc.*, **1994**, *116*, 12061–12062.
- [33] Wollacott, A.M.; Merz, K.M. Jr. Development of a parameterized force field to reproduce semiempirical geometries. *J. Chem. Theory Comput.*, **2006**, *2*, 1070–1077.
- [34] Tessier, M.B.; DeMarco, M.L.; Yongye, A.B.; Woods, R.J. Extension of the GLYCAM06 biomolecular force field to lipids, lipid bilayers and glycolipids. *Mol. Simul.*, **2008**, *34*, 349–363.
- [35] Kirschner, K.N.; Yongye, A.B.; Tschampel, S.M.; González-Outeiriño, J.; Daniels, C.R.; Foley, B.L.; Woods, R.J. GLYCAM06: A generalizable biomolecular force field. Carbohydrates. *J. Comput. Chem.*, **2008**, *29*, 622–655.
- [36] Case, D.A.; Cheatham, T.; Darden, T.; Gohlke, H.; Luo, R.; Merz, K.M. Jr.; Onufriev, A.; Simmerling, C.; Wang, B.; Woods, R. The Amber biomolecular simulation programs. *J. Computat. Chem.*, **2005**, *26*, 1668–1688.
- [37] Kirschner, K.N.; Woods, R.J. Solvent interactions determine carbohydrate conformation. *Proc. Natl. Acad. Sci. USA*, **2001**, *98*, 10541–10545.
- [38] Woods, R.J. Restrained electrostatic potential charges for condensed phase simulations of carbohydrates. *J. Mol. Struct (Theochem)*, **2000**, *527*, 149–156.

BIBLIOGRAPHY

- [39] Woods, R.J. Derivation of net atomic charges from molecular electrostatic potentials. *J. Comput. Chem.*, **1990**, *11*, 29–310.
- [40] Basma, M.; Sundara, S.; Calgan, D.; Venali, T.; Woods, R.J. Solvated ensemble averaging in the calculation of partial atomic charges. *J. Comput. Chem.*, **2001**, *22*, 1125–1137.
- [41] Tschampel, S.M.; Kennerty, M.R.; Woods, R.J. TIP5P-consistent treatment of electrostatics for biomolecular simulations. *J. Chem. Theory Comput.*, **2007**, *3*, 1721–1733.
- [42] DeMarco, M.L.; Woods, R.J. Bridging computational biology and glycobiology: A game of snakes and ladders. *Glycobiology*, *in press*, **2008**.
- [43] Åqvist, J. Ion-water interaction potentials derived from free energy perturbation simulations. *J. Phys. Chem.*, **1990**, *94*, 8021–8024.
- [44] Dang, L. Mechanism and thermodynamics of ion selectivity in aqueous solutions of 18-crown-6 ether: A molecular dynamics study. *J. Am. Chem. Soc.*, **1995**, *117*, 6954–6960.
- [45] Auffinger, P.; Cheatham, T.E. III; Vaiana, A.C. Spontaneous formation of KCl aggregates in biomolecular simulations: a force field issue? *J. Chem. Theory Comput.*, **2007**, *3*, 1851–1859.
- [46] Jorgensen, W.L.; Chandrasekhar, J.; Madura, J.; Klein, M.L. Comparison of simple potential functions for simulating liquid water. *J. Chem. Phys.*, **1983**, *79*, 926–935.
- [47] Price, D.J.; Brooks, C.L. A modified TIP3P water potential for simulation with Ewald summation. *J. Chem. Phys.*, **2004**, *121*, 10096–10103.
- [48] Jorgensen, W.L.; Madura, J.D. Temperature and size dependence for Monte Carlo simulations of TIP4P water. *Mol. Phys.*, **1985**, *56*, 1381–1392.
- [49] Horn, H.W.; Swope, W.C.; Pitara, J.W.; Madura, J.D.; Dick, T.J.; Hura, G.L.; Head-Gordon, T. Development of an improved four-site water model for biomolecular simulations: TIP4P-Ew. *J. Chem. Phys.*, **2004**, *120*, 9665–9678.
- [50] Horn, H.W.; Swope, W.C.; Pitara, J.W. Characterization of the TIP4P-Ew water model: Vapor pressure and boiling point. *J. Chem. Phys.*, **2005**, *123*, 194504.
- [51] Mahoney, M.W.; Jorgensen, W.L. A five-site model for liquid water and the reproduction of the density anomaly by rigid, nonpolarizable potential functions. *J. Chem. Phys.*, **2000**, *112*, 8910–8922.
- [52] Caldwell, J.W.; Kollman, P.A. Structure and properties of neat liquids using nonadditive molecular dynamics: Water, methanol and N-methylacetamide. *J. Phys. Chem.*, **1995**, *99*, 6208–6219.
- [53] Berendsen, H.J.C.; Grigera, J.R.; Straatsma, T.P. The missing term in effective pair potentials. *J. Phys. Chem.*, **1987**, *91*, 6269–6271.
- [54] Wu, Y.; Tepper, H.L.; Voth, G.A. Flexible simple point-charge water model with improved liquid-state properties. *J. Chem. Phys.*, **2006**, *124*, 024503.

- [55] Paesani, F.; Zhang, W.; Case, D.A.; Cheatham, T.E.; Voth, G.A. An accurate and simple quantum model for liquid water. *J. Chem. Phys.*, **2006**, *125*, 184507.
- [56] Crowley, M.F.; Williamson, M.J.; Walker, R.C. CHAMBER: Comprehensive support for CHARMM force fields within the AMBER software. *Int. J. Quant. Chem.*, **2009**, *109*, 3767.
- [57] Cornell, W.D.; Cieplak, P.; Bayly, C.I.; Gould, I.R.; Merz, K.M. Jr.; Ferguson, D.M.; Spellmeyer, D.C.; Fox, T.; Caldwell, J.W.; Kollman, P.A. A second generation force field for the simulation of proteins, nucleic acids, and organic molecules. *J. Am. Chem. Soc.*, **1995**, *117*, 5179–5197.
- [58] MacKerell Jr., A.D.; Bashford, D.; Bellott, M.; Dunbrack, R.L.; Evanseck, J.D.; Field, M.J.; Fischer, S.; Gao, J.; Guo, H.; Ha, S.; Joseph-McCarthy, D.; Kuchnir, L.; Kuczera, K.; Lau, F.T.K.; Mattos, C.; Michnick, S.; Ngo, T.; Nguyen, D.T.; Prodhom, B.; Reiher, W.E.; Roux, B.; Schlenkrich, M.; Smith, J.C.; Stote, R.; Straub, J.; Watanabe, M.; Wiorkiewicz-Kuczera, J.; Yin, D.; Karplus, M. All-Atom Empirical Potential for Molecular Modeling and Dynamics Studies of Proteins. *J. Phys. Chem. B*, **1998**, *102*, 3586–3616.
- [59] MacKerell Jr., A.D.; Banavali, N.; Foloppe, N. Development and current status of the CHARMM force field for nucleic acids. *Biopolymers*, **2000**, *56*, 257–265.
- [60] Brooks, B.R.; Bruccoleri, R.E.; Olafson, D.J.; States, D.J.; Swaminathan, S.; Karplus, M. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *J. Computat. Chem.*, **1983**, *4*, 187–217.
- [61] Brooks, B. R.; Brooks, C. L.; Mackerell, A. D.; Nilsson, L.; Petrella, R. J.; Roux, B.; Won, Y.; Archontis, G.; Bartels, C.; Boresch, S.; Caffisch, A.; Caves, L.; Cui, Q.; Dinner, A. R.; Feig, M.; Fischer, S.; Gao, J.; Hodosecek, M.; Im, W.; Kuczera, K.; Lazaridis, T.; Ma, J.; Ovchinnikov, V.; Paci, E.; Pastor, R. W.; Post, C. B.; Pu, J. Z.; Schaefer, M.; Tidor, B.; Venable, R. M.; Woodcock, H. L.; Wu, X.; Yang, W.; York, D. M.; Karplus, M. CHARMM: the biomolecular simulation program. *J. Comput. Chem.*, **2009**, *30*, 1545–1614.
- [62] MacKerell, A.D. Jr.; Feig, M.; Brooks III, C.L. Improved Treatment of the Protein Backbone in Empirical Force Fields. *J. Am. Chem. Soc.*, **2004**, *126*, 698–699.
- [63] MacKerell, A.D. Jr.; Feig, M.; Brooks III, C.L. Extending the treatment of backbone energetics in protein force fields: Limitations of gas-phase quantum mechanics in reproducing protein conformational distributions in molecular dynamics simulations. *J. Computat. Chem.*, **2004**, *25*, 1400–1415.
- [64] Bondi, A. van der Waals volumes and radii. *J. Phys. Chem.*, **1964**, *68*, 441–451.
- [65] Tsui, V.; Case, D.A. Theory and applications of the generalized Born solvation model in macromolecular simulations. *Biopolymers (Nucl. Acid. Sci.)*, **2001**, *56*, 275–291.

BIBLIOGRAPHY

- [66] Onufriev, A.; Bashford, D.; Case, D.A. Exploring protein native states and large-scale conformational changes with a modified generalized Born model. *Proteins*, **2004**, *55*, 383–394.
- [67] Tsui, V.; Case, D.A. Molecular dynamics simulations of nucleic acids using a generalized Born solvation model. *J. Am. Chem. Soc.*, **2000**, *122*, 2489–2498.
- [68] Wang, J.; Wolf, R.M.; Caldwell, J.W.; Kollman, P.A.; Case, D.A. Development and testing of a general Amber force field. *J. Comput. Chem.*, **2004**, *25*, 1157–1174.
- [69] Wang, B.; Merz, K.M. Jr. A fast QM/MM (quantum mechanical/molecular mechanical) approach to calculate nuclear magnetic resonance chemical shifts for macromolecules. *J. Chem. Theory Comput.*, **2006**, *2*, 209–215.
- [70] Peters, M.B.; Yang, Y.; Wang, B.; Fusti-Molnar, L.; Weaver, M.N.; Merz, K.M. Structural Survey of Zinc-Containing Proteins and Development of the Zinc AMBER Force Field (ZAFF). *J. Chem. Theor. Comput.*, **2010**, *6*, 2935–2947.
- [71] Jakalian, A.; Bush, B.L.; Jack, D.B.; Bayly, C.I. Fast, efficient generation of high-quality atomic charges. AM1-BCC model: I. Method. *J. Comput. Chem.*, **2000**, *21*, 132–146.
- [72] Jakalian, A.; Jack, D.B.; Bayly, C.I. Fast, efficient generation of high-quality atomic charges. AM1-BCC model: II. Parameterization and Validation. *J. Comput. Chem.*, **2002**, *23*, 1623–1641.
- [73] Wang, J.; Kollman, P.A. Automatic parameterization of force field by systematic search and genetic algorithms. *J. Comput. Chem.*, **2001**, *22*, 1219–1228.
- [74] Graves, A.P.; Shivakumar, D.M.; Boyce, S.E.; Jacobson, M.P.; Case, D.A.; Shoichet, B.K. Rescoring docking hit lists for model cavity sites: Predictions and experimental testing. *J. Mol. Biol.*, **2008**, *377*, 914–934.
- [75] Jojart, B.; Martinek, T.A. Performance of the general amber force field in modeling aqueous POPC membrane bilayers. *J. Comput. Chem.*, **2007**, *28*, 2051–2058.
- [76] Rosso, L.; Gould, I.R. Structure and dynamics of phospholipid bilayers using recently developed general all-atom force fields. *J. Comput. Chem.*, **2008**, *29*, 24–37.
- [77] Wang, J.; Hou, T. Application of Molecular Dynamics Simulations in Molecular Property Prediction. 1. Density and Heat of Vaporization. *J. Chem. Theory Comput.*, **2011**, *7*, 2151–2165.
- [78] Mobley, David L.; Bayly, Christopher I.; Cooper, Matthew D.; Shirts, Michael R.; Dill, Ken A. Small Molecule Hydration Free Energies in Explicit Solvent: An Extensive Test of Fixed-Charge Atomistic Simulations. *J. Chem. Theory Comput.*, **2009**, *5*, 350–358.
- [79] Wang, J.; Wang, W.; Kollman, P.A.; Case, D.A. Automatic atom type and bond type perception in molecular mechanical. *J. Mol. Graphics Model.*, **2006**, *25*, 247–260.

- [80] Srinivasan, J.; Cheatham, T.E. III; Cieplak, P.; Kollman, P.; Case, D.A. Continuum solvent studies of the stability of DNA, RNA, and phosphoramidate–DNA helices. *J. Am. Chem. Soc.*, **1998**, *120*, 9401–9409.
- [81] Chong, Lillian T.; Duan, Y.; Wang, L.; Massova, I.; Kollman, P.A. Molecular dynamics and free-energy calculations applied to affinity maturation in antibody 48G7. *Proc. Natl. Acad. Sci. USA*, **1999**, *96*, 14330–14335.
- [82] Kollman, P.A.; Massova, I.; Reyes, C.; Kuhn, B.; Huo, S.; Chong, L.; Lee, M.; Lee, T.; Duan, Y.; Wang, W.; Donini, O.; Cieplak, P.; Srinivasan, J.; Case, D.A.; Cheatham, T.E. III. Calculating structures and free energies of complex molecules: Combining molecular mechanics and continuum models. *Accts. Chem. Res.*, **2000**, *33*, 889–897.
- [83] Huo, S.; Massova, I.; Kollman, P.A. Computational Alanine Scanning of the 1:1 Human Growth Hormone-Receptor Complex. *J. Comput. Chem.*, **2002**, *23*, 15–27.
- [84] Homeyer, N.; Gohlke, H. Free energy calculations by the molecular mechanics poisson-boltzmann surface area method. *Mol. Informatics*, **2012**, DOI: 10.1002/minf.201100135.
- [85] Wang, W.; Kollman, P. Free energy calculations on dimer stability of the HIV protease using molecular dynamics and a continuum solvent model. *J. Mol. Biol.*, **2000**, *303*, 567.
- [86] Reyes, C.; Kollman, P. Structure and thermodynamics of RNA-protein binding: Using molecular dynamics and free energy analyses to calculate the free energies of binding and conformational change. *J. Mol. Biol.*, **2000**, *297*, 1145–1158.
- [87] Lee, M.R.; Duan, Y.; Kollman, P.A. Use of MM-PB/SA in estimating the free energies of proteins: Application to native, intermediates, and unfolded villin headpiece. *Proteins*, **2000**, *39*, 309–316.
- [88] Wang, J.; Morin, P.; Wang, W.; Kollman, P.A. Use of MM-PBSA in reproducing the binding free energies to HIV-1 RT of TIBO derivatives and predicting the binding mode to HIV-1 RT of efavirenz by docking and MM-PBSA. *J. Am. Chem. Soc.*, **2001**, *123*, 5221–5230.
- [89] Marinelli, L.; Cosconati, S.; Steinbrecher, T.; Limongelli, V.; Bertamino, A.; Novellino, E.; Case, D.A. Homology Modeling of NR2B Modulatory Domain of NMDA Receptor and Analysis of Ifenprodil Binding. *ChemMedChem*, **2007**, *2*, 1498–1510.
- [90] Tan, C. H.; Tan, Y. H.; Luo, R. Implicit nonpolar solvent models. *J. Phys. Chem. B*, **2007**, *111*, 12263–12274.
- [91] Walker, R.C.; Crowley, M.F.; Case, D.A. The implementation of a fast and efficient hybrid QM/MM potential method within The Amber 9.0 sander module. *J. Computat. Chem.*, **2008**, *29*, 1019–1031.
- [92] Seabra, G.M.; Walker, R.C.; Elstner, M.; Case, D.A.; Roitberg, A.E. Implementation of the SCC-DFTB Method for Hybrid QM/MM Simulations within the Amber Molecular Dynamics Package. *J. Phys. Chem. A*, **2007**, *20*, 5655–5664.

BIBLIOGRAPHY

- [93] Elstner, M.; Porezag, D.; Jungnickel, G.; Elsner, J.; Haugk, M.; Frauenheim, T.; Suhai, S.; Seifert, G. Self-consistent charge density functional tight-binding method for simulation of complex material properties. *Phys. Rev. B*, **1998**, 58, 7260.
- [94] Kruger, T.; Elstner, M.; Schiffels, P.; Frauenheim, T. Validation of the density-functional based tight-binding approximation. *J. Chem. Phys.*, **2005**, 122, 114110.
- [95] Giese, Timothy J.; York, Darrin M. Charge-dependent model for many-body polarization, exchange, and dispersion interactions in hybrid quantum mechanical/molecular mechanical calculations. *J. Chem. Phys.*, **2007**, 127(19), 194101–194111.
- [96] Stewart, J.J.P. Optimization of parameters for semiempirical methods I. Method. *J. Comput. Chem.*, **1989**, 10, 209–220.
- [97] Dewar, M.J.S.; Zoebisch, E.G.; Healy, E.F.; Stewart, J.J.P. AM1: A new general purpose quantum mechanical molecular model. *J. Am. Chem. Soc.*, **1985**, 107, 3902–3909.
- [98] Rocha, G.B.; Freire, R.O.; Simas, A.M.; Stewart, J.J.P. RM1: A Reparameterization of AM1 for H, C, N, O, P, S, F, Cl, Br and I. *J. Comp. Chem.*, **2006**, 27, 1101–1111.
- [99] Dewar, M.J.S.; Thiel, W. Ground states of molecules. 38. The MNDO method, approximations and parameters. *J. Am. Chem. Soc.*, **1977**, 99, 4899–4907.
- [100] Repasky, M.P.; Chandrasekhar, J.; Jorgensen, W.L. PDDG/PM3 and PDDG/MNDO: Improved semiempirical methods. *J. Comput. Chem.*, **2002**, 23, 1601–1622.
- [101] McNamara, J.P.; Muslim, A.M.; Abdel-Aal, H.; Wang, H.; Mohr, M.; Hillier, I.H.; Bryce, R.A. Towards a quantum mechanical force field for carbohydrates: A reparameterized semiempirical MO approach. *Chem. Phys. Lett.*, **2004**, 394, 429–436.
- [102] Bernal-Uruchurtu, M. I.; Ruiz-López, M. F. Basic ideas for the correction of semiempirical methods describing H-bonded systems. *Chem. Phys. Lett.*, **2000**, 330, 118–124.
- [103] Arillo-Flores, O. I.; Ruiz-López, M. F.; Bernal-Uruchurtu, M. I. Can semi-empirical models describe HCl dissociation in water? *Theoret. Chem. Acc.*, **2007**, 118, 425–435.
- [104] Thiel, Walter; Voityuk, Alexander A. Extension of the MNDO formalism to d orbitals: Integral approximations and preliminary numerical results. *Theoret. Chim. Acta*, **1992**, 81, 391–404.
- [105] Thiel, Walter; Voityuk, Alexander A. Extension of the MNDO formalism to d orbitals: Integral approximations and preliminary numerical results. *Theoret. Chim. Acta*, **1996**, 93, 315.
- [106] Thiel, Walter; Voityuk, Alexander A. Erratum: Extension of MNDO to d orbitals: Parameters and results for the second-row elements and for the zinc group. *J. Phys. Chem.*, **1996**, 100, 616–626.
- [107] Imhof, Petra; Noé, Frank; Fischer, Stefan; Smith, Jeremy C. AM1/d Parameters for Magnesium in Metalloenzymes. *J. Chem. Theory Comput.*, **2006**, 2, 1050–1056.

- [108] Nam, Kwangho; Cui, Qiang; Gao, Jiali; York, Darrin M. Specific Reaction Parametrization of the AM1/d Hamiltonian for Phosphoryl Transfer Reactions: H, O, and P Atoms. *J. Chem. Theory Comput.*, **2007**, 3, 486–504.
- [109] Stewart, J.J.P. Optimization of parameters for semiempirical methods V: Modification of NDDO approximations and application to 70 elements. *J. Mol. Mod.*, **2007**, 13, 1173–1213.
- [110] Porezag, D.; Frauenheim, T.; Kohler, T.; Seifert, G.; Kaschner, R. Construction of tight-binding-like potentials on the basis of density-functional-theory: Applications to carbon. *Phys. Rev. B*, **1995**, 51, 12947.
- [111] Seifert, G.; Porezag, D.; Frauenheim, T. Calculations of molecules, clusters and solids with a simplified LCAO-DFT-LDA scheme. *Int. J. Quantum Chem.*, **1996**, 58, 185.
- [112] Elstner, M.; Hobza, P.; Frauenheim, T.; Suhai, S.; Kaxiras, E. Hydrogen bonding and stacking interactions of nucleic acid base pairs: a density-functional-theory based treatment. *J. Chem. Phys.*, **2001**, 114, 5149.
- [113] Kalinowski, J.A.; Lesyng, B.; Thompson, J.D.; Cramer, C.J.; Truhlar, D.G. Class IV charge model for the self-consistent charge density-functional tight-binding method. *J. Phys. Chem. A*, **2004**, 108, 2545–2549.
- [114] Yang, Y.; Yu, H.; York, D.M.; Cui, Q.; Elstner, M. Extension of the self-consistent charge density-functional tight-binding method: Third-order expansion of the density functional theory total energy and introduction of a modified effective Coulomb interaction. *J. Phys. Chem. A*, **2007**, 111, 10861–10873.
- [115] Korth, Martin. Third-generation hydrogen-bonding corrections for semiempirical qm methods and force fields. *J. Chem. Theory Comput.*, **2010**, 6, 3808.
- [116] Jurecka, Petr; Cerný, Jirí; Hobza, Pavel; Salahub, Dennis R. Density functional theory augmented with an empirical dispersion term. Interaction energies and geometries of 80 noncovalent complexes compared with ab initio quantum mechanics calculations. *J. Comp. Chem.*, **2007**, 28, 555–569.
- [117] Korth, Martin; Pitonak, Michal; Rezac, Jan; Hobza, Pavel. A transferable h-bonding correction for semiempirical quantum-chemical methods. *J. Chem. Theory Comput.*, **2010**, 6, 344–352.
- [118] Chou, J.J.; Case, D.A.; Bax, A. Insights into the mobility of methyl-bearing side chains in proteins. *J. Am. Chem. Soc.*, **2003**, 125, 8959–8966.
- [119] Perez, C.; Lohr, F.; Ruterjans, H.; Schmidt, J.M. Self-Consistent Karplus Parameterization of (3)J couplings depending on the polypeptide side-chain torsion $\chi(1)$. *J. Am. Chem. Soc.*, **2001**, 123, 7081–7093.
- [120] Connolly, M.L. Analytical molecular surface calculation. *J. Appl. Cryst.*, **1983**, 16, 548–558.

BIBLIOGRAPHY

- [121] Lu, XJ; Olson, WK. 3dna: a software package for the analysis, rebuilding and visualization of three-dimensional nucleic acid structures. *NUCLEIC ACIDS RESEARCH*, **2003**, *31*(17), 5108–5121.
- [122] Babcock, M.S.; Pednault, E.P.D.; Olson, W.K. Nucleic Acid Structure Analysis. *J. Mol. Biol.*, **1994**, *237*, 125–156.
- [123] Olson, Wilma K.; Bansal, Manju; Burley, Stephen K.; Dickerson, Richard E.; Gerstein, Mark; Harvey, Stephen C.; Heinemann, Udo; Lu, Xiang-Jun; Neidle, Stephen; Shakked, Zippora; Sklenar, Heinz; Suzuki, Masashi; Tung, Chang-Shung; Westhof, Eric; Wolberger, Cynthia; Berman, Helen M. A standard reference frame for the description of nucleic acid base-pair geometry. *J. Mol. Biol.*, **2001**, *313*, 229–237.
- [124] Altona, C; Sundaralingam, M. Conformational analysis of the sugar ring in nucleosides and nucleotides. a new description using the concept of pseudorotation. *J Am Chem Soc*, **1972**, *94*, 8205–8212.
- [125] Harvey, SC; Prabhakaran, M. Ribose puckering - structure, dynamics, energetics, and the pseudorotation cycle. *J Am Chem Soc*, **1986**, *108*, 6128–6136.
- [126] Cremer, D; Pople, JA. A general definition of ring puckering coordinates. *J Am Chem Soc*, **1975**, *97*, 1354–1358.
- [127] Wong, V.; Case, D.A. Evaluating rotational diffusion from protein md simulations. *J. Phys. Chem. B*, **2008**, *112*(19), 6013–6024.
- [128] Kabsch, W.; Sander, C. Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers*, **1983**, *22*, 2577–2637.
- [129] Weiser, J.; Shenkin, P.S.; Still, W.C. Approximate Atomic Surfaces from Linear Combinations of Pairwise Overlaps (LCPO). *J. Computat. Chem.*, **1999**, *20*, 217–230.
- [130] Luo, R.; David, L.; Gilson, M.K. Accelerated Poisson-Boltzmann calculations for static and dynamic systems. *J. Comput. Chem.*, **2002**, *23*, 1244–1253.
- [131] Wang, J.; Luo, R. Assessment of Linear Finite-Difference Poisson-Boltzmann Solvers. *J. Comput. Chem.*, **2010**, *in press*.
- [132] Cai, Q.; Hsieh, M.-J.; Wang, J.; Luo, R. Performance of Nonlinear Finite-Difference Poisson-Boltzmann Solvers. *J. Chem. Theory Comput.*, **2010**, *6*, 203–211.
- [133] Honig, B.; Nicholls, A. Classical electrostatics in biology and chemistry. *Science*, **1995**, *268*, 1144–1149.
- [134] Lu, Q.; Luo, R. A Poisson-Boltzmann dynamics method with nonperiodic boundary condition. *J. Chem. Phys.*, **2003**, *119*, 11035–11047.
- [135] Gilson, M.K.; Sharp, K.A.; Honig, B.H. Calculating the electrostatic potential of molecules in solution: method. *J. Comput. Chem.*, **1988**, *9*, 327–35.

- [136] Warwicker, J.; Watson, H.C. Calculation of the electric potential in the active site cleft due to. *J. Mol. Biol.*, **1982**, *157*, 671–679.
- [137] Klapper, I.; Hagstrom, R.; Fine, R.; Sharp, K.; Honig, B. Focussing of electric fields in the active stie of Cu, Zn superoxide dismutase. *Proteins*, **1986**, *1*, 47–59.
- [138] Sitkoff, D.; Sharp, K.A.; Honig, B. Accurate calculation of hydration free energies using macroscopic solvent models. *J. Phys. Chem.*, **1994**, *98*, 1978–1988.
- [139] Gallicchio, E.; Kubo, M.M.; Levy, R.M. Enthalpy-entropy and cavity decomposition of alkane hydration free energies: Numerical results and implications for theories of hydrophobic solvation. *J. Phys. Chem.*, **2000**, *104*, 6271–6285.
- [140] Floris, F.; Tomasi, J. Evaluation of the dispersion contribution to the solvation energy. A simple computational model in the continuum approximation. *J. Comput. Chem.*, **1989**, *10*, 616–627.
- [141] Davis, M.E.; McCammon, J.A. Solving the finite-difference linearized Poisson-Boltzmann equation – a comparison of relaxation and conjugate gradient methods. *J. Comput. Chem.*, **1989**, *10*, 386–391.
- [142] Nicholls, A.; Honig, B. A rapid finite difference algorithm, utilizing successive over-relaxation to solve the Poisson-Boltzmann equation. *J. Comput. Chem.*, **1991**, *12*, 435–445.
- [143] Bashford, D. An object-oriented programming suite for electrostatic effects in biological molecules. *Lect. Notes Comput. Sci.*, **1997**, *1343*, 233–240.
- [144] Wang, J.; Cai, Q.; Li, Z.; Zhao, H.; Luo, R. Achieving Energy Conservation in Poisson-Boltzmann Molecular Dynamics: Accuracy and Precision with Finite-difference Algorithms. *Chem. Phys. Lett.*, **2009**, *468*, 112.
- [145] Li, Z.; K., Ito. *The Immersed Interface Method: numerical sollutions of PDEs involving interfaces and irregular domains*. SIAM Frontiers in Applied Mathematics, Philadelphia, 2006.
- [146] Luty, B.A.; Davis, M.E.; McCammon, J.A. Electrostatic energy calculations by a finite-difference method: Rapid calculation of charge-solvent interaction energies. *J. Comput. Chem.*, **1992**, *13*, 768–771.
- [147] Cai, Q.; Wang, J.; Zhao, H.; Luo, R. On removal of charge singularity in Poisson-Boltzmann equation. *J. Chem. Phys.*, **2009**, *130*, 145101.
- [148] Cai, Q.; Ye, X.; Wang, J.; Luo, R. Dielectric boundary force in numerical Poisson-Boltzmann methods: Theory and numerical strategies. *Chem. Phys. Lett.*, **2011**, *514*, 368–373.
- [149] Davis, M.E.; McCammon, J.A. Dielectric boundary smoothing in finite difference solutions of the Poisson equation: An approach to improve accuracy and convergence. *J. Comput. Chem.*, **1991**, *12*, 909–912.

BIBLIOGRAPHY

- [150] Davis, M.E.; McCammon, J.A. Electrostatics in biomolecular structure and dynamics. *Chem. Rev.*, **1990**, *90*, 509–521.
- [151] Tan, C. H.; Yang, L. J.; Luo, R. How well does Poisson-Boltzmann implicit solvent agree with explicit solvent? A quantitative analysis. *J. Phys. Chem. B*, **2006**, *110*, 18680–18687.
- [152] Ye, X.; Wang, J.; Luo, R. A Revised Density Function for Molecular Surface Calculation in Continuum Solvent Models. *J. Chem. Theory Comput.*, **2010**, *6*, 1157–1169.
- [153] Cai, Q.; Ye, X.; Wang, J.; Luo, R. On-the-Fly Numerical Surface Integration for Finite-Difference Poisson-Boltzmann Methods. *J. Chem. Theory Comput.*, **2011**, *7*, 3608–3619.
- [154] Hsieh, M.-J.; Luo, R. Exploring a coarse-grained distributive strategy for finite-difference poisson-boltzmann calculations. *Journal of Molecular Modeling*, **2011**.
- [155] Gilson, M.K.; Davis, M.E; Luty, B.A.; McCammon, J.A. Computation of electrostatic forces on solvated molecules using the Poisson-Boltzmann equation. *J Phys Chem*, **1993**, *97*(14), 3591–3600.
- [156] Luchko, T.; Gusarov, S.; Roe, D. R.; Simmerling, C.; Case, D. A.; Tuszynski, J.; Kovalenko, A. Three-dimensional molecular theory of solvation coupled with molecular dynamics in Amber. *J. Chem. Theory Comput.*, **2010**, *6*, 607–624.
- [157] Chandler, D.; Andersen, H. C. Optimized cluster expansions for classical fluids. ii. theory of molecular liquids. *J. Chem. Phys.*, **1972**, *57*(5), 1930–1937.
- [158] Hirata, F.; Rossky, P. J. An extended RISM equation for molecular polar fluids. *Chem. Phys. Lett.*, **1981**, pp 329–334.
- [159] Hirata, F.; Pettitt, B. M.; Rossky, P. J. Application of an extended rism equation to dipolar and quadrupolar fluids. *J. Chem. Phys.*, **1982**, *77*(1), 509–520.
- [160] Hirata, F.; Rossky, P. J.; Pettitt, B. M. The interionic potential of mean force in a molecular polar solvent from an extended rism equation. *J. Chem. Phys.*, **1983**, *78*(6), 4133–4144.
- [161] Chandler, D.; McCoy, J.; Singer, S. Density functional theory of nonuniform polyatomic systems. i. general formulation. *J. Chem. Phys.*, **1986**, *85*(10), 5971–5976.
- [162] Chandler, D.; McCoy, J.; Singer, S. Density functional theory of nonuniform polyatomic systems. ii. rational closures for integral equations. *J. Chem. Phys.*, **1986**, *85*(10), 5977–5982.
- [163] Beglov, D.; Roux, B. Numerical solution of the hypernetted chain equation for a solute of arbitrary geometry in three dimensions. *J. Chem. Phys.*, **1995**, *103*(1), 360–364.
- [164] Beglov, D.; Roux, B. An integral equation to describe the solvation of polar molecules in liquid water. *J. Phys. Chem. B*, **1997**, *101*(39), 7821–7826.

- [165] Kovalenko, A.; Hirata, F. Three-dimensional density profiles of water in contact with a solute of arbitrary shape: a RISM approach. *Chem. Phys. Lett.*, **1998**, 290(1-3), 237–244.
- [166] Kovalenko, A.; Hirata, F. Self-consistent description of a metal–water interface by the Kohn–Sham density functional theory and the three-dimensional reference interaction site model. *J. Chem. Phys.*, **1999**, 110(20), 10095–10112.
- [167] Kovalenko, A. In Hirata [242], chapter 4.
- [168] Kovalenko, A.; Hirata, F. Potentials of mean force of simple ions in ambient aqueous solution. i: Three-dimensional reference interaction site model approach. *J. Chem. Phys.*, **2000**, 112, 10391–10402.
- [169] Kovalenko, A.; Hirata, F. Potentials of mean force of simple ions in ambient aqueous solution. ii: Solvation structure from the three-dimensional reference interaction site model approach, and comparison with simulations. *J. Chem. Phys.*, **2000**, 112, 10403–10417.
- [170] Hansen, J.-P.; McDonald, I. R. *Theory of simple liquids*. Academic Press, London, 1990.
- [171] Hirata, F. In *Molecular Theory of Solvation* [242], chapter 1.
- [172] Perkyns, J. S.; Pettitt, B. M. A site-site theory for finite concentration saline solutions. *J. Chem. Phys.*, **1992**, 97(10), 7656–7666.
- [173] Kovalenko, A.; Ten-No, S.; Hirata, F. Solution of three-dimensional reference interaction site model and hypernetted chain equations for simple point charge water by modified method of direct inversion in iterative subspace. *J. Comput. Chem.*, **1999**, 20(9), 928–936.
- [174] Kaminski, J. W.; Gusarov, S.; Wesolowski, T. A.; Kovalenko, Andiry. Modeling solvatochromic shifts using the orbital-free embedding potential at statistically mechanically averaged solvent density. *J. Phys. Chem. A*, **2010**, in press.
- [175] Frigo, M.; Johnson, S. G. FFTW: An adaptive software architecture for the FFT. in *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pp 1381–1384. IEEE, 1998.
- [176] Frigo, M. A fast Fourier transform compiler. in *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 34, pp 169–180. ACM, May 1999.
- [177] Singer, S. J.; Chandler, D. Free energy functions in the extended RISM approximation. *Mol. Phys.*, **1985**, 55, 621–625.
- [178] Pettitt, B. M.; Rossky, P. J. Alkali halides in water: Ion-solvent correlations and ion-ion potentials of mean force at infinite dilution. *J. Chem. Phys.*, **1986**, 15, 5836–5844.
- [179] Kast, Stefan M. Free energies from integral equation theories: Enforcing path independence. *Phys. Rev. E*, **2003**, 67, 041203.
- [180] Schmeer, G.; Maurer, A.

BIBLIOGRAPHY

- [181] Gusarov, S.; Ziegler, T.; Kovalenko, A. Self-consistent combination of the three-dimensional RISM theory of molecular solvation with analytical gradients and the amsterdam density functional package. *J. Phys. Chem. A*, **2006**, *110*(18), 6083–6090.
- [182] Miyata, T.; Hirata, F. Combination of molecular dynamics method and 3D-RISM theory for conformational sampling of large flexible molecules in solution. *J. Comput. Chem.*, Oct 2007, *29*, 871–882.
- [183] Kast, Stefan M.; Kloss, Thomas. Closed-form expressions of the chemical potential for integral equation closures with certain bridge functions. *J. Chem. Phys.*, **2008**, *129*, 236101.
- [184] Chandler, David; Singh, Yashwant; Richardson, Diane M. Excess electrons in simple fluids. I. General equilibrium theory for classical hard sphere solvents. *J. Chem. Phys.*, **1984**, *81*(4), 1975–1982.
- [185] Ichiye, Toshiko; Chandler, David. Hypernetted chain closure reference interaction site method theory of structure and thermodynamics for alkanes in water. *J. Phys. Chem.*, **1988**, *92*(18), 5257–5261.
- [186] Lee, Pil H.; Maggiora, Gerald M. Solvation thermodynamics of polar molecules in aqueous solution by the XRISM method. *J. Phys. Chem.*, **1993**, *97*(39), 10175–10185.
- [187] Genheden, S.; Luchko, T.; Gusarov, S.; Kovalenko, A.; Ryde, U. An MM/3D-RISM approach for ligand-binding affinities. *J. Phys. Chem.*, **2010**. Accepted.
- [188] Yu, Hsiang-Ai; Roux, Benoit; Karplus, Martin. Solvation thermodynamics: An approach from analytic temperature derivatives. *J. Chem. Phys.*, **1990**, *92*(8), 5020–5033.
- [189] Yamazaki, Takeshi; Blinov, Nikolay; Wishart, David; Kovalenko, Andriy. Hydration effects on the HET-s prion and amyloid- β 2 fibrillous aggregates, studied with three-dimensional molecular theory of solvation. *Biophys. J.*, **2008**, *95*, 4540–4548.
- [190] Yamazaki, T.; Kovalenko, A.; Murashov, V.V.; Patey, G.N. Ion solvation in a water-urea mixture. *J. Phys. Chem. B*, **2010**, *114*(1), 613–619.
- [191] Yu, H.-A.; Karplus, M. A thermodynamic analysis of solvation. *J. Chem. Phys.*, **1988**, *89*, 2366–2379.
- [192] Hawkins, G.D.; Cramer, C.J.; Truhlar, D.G. Parametrized models of aqueous free energies of solvation based on pairwise descreening of solute atomic charges from a dielectric medium. *J. Phys. Chem.*, **1996**, *100*, 19824–19839.
- [193] Hawkins, G.D.; Cramer, C.J.; Truhlar, D.G. Pairwise solute descreening of solute charges from a dielectric medium. *Chem. Phys. Lett.*, **1995**, *246*, 122–129.
- [194] Cerutti, D.S.; Case, D.A. Multi-Level Ewald: A Hybrid Multigrid/Fast Fourier Transform Approach to the Electrostatic Particle-Mesh Problem. *J. Chem. Theory Comput.*, **2010**, *6*, 443–458.

- [195] Essmann, U.; Perera, L.; Berkowitz, M.L.; Darden, T.; Lee, H.; Pedersen, L.G. A smooth particle mesh Ewald method. *J. Chem. Phys.*, **1995**, *103*, 8577–8593.
- [196] Gohlke, H.; Kuhn, L. A.; Case, D. A. Change in protein flexibility upon complex formation: Analysis of Ras-Raf using molecular dynamics and a molecular framework approach. *Proteins*, **2004**, *56*, 322–327.
- [197] Ahmed, A.; Gohlke, H. Multiscale modeling of macromolecular conformational changes combining concepts from rigidity and elastic network theory. *Proteins*, **2006**, *63*, 1038–1051.
- [198] Fulle, S.; Gohlke, H. Analyzing the flexibility of RNA structures by constraint counting. *Biophys. J.*, **2008**, DOI:10.1529/biophysj.107.113415.
- [199] Sigalov, G.; Fenley, A.; Onufriev, A. Analytical electrostatics for biomolecules: Beyond the generalized Born approximation. *J. Chem. Phys.*, **2006**, *124*, 124902.
- [200] Major, F.; Turcotte, M.; Gautheret, D.; Lapalme, G.; Fillon, E.; Cedergren, R. The Combination of Symbolic and Numerical Computation for Three-Dimensional Modeling of RNA. *Science*, **1991**, *253*, 1255–1260.
- [201] Gautheret, D.; Major, F.; Cedergren, R. Modeling the three-dimensional structure of RNA using discrete nucleotide conformational sets. *J. Mol. Biol.*, **1993**, *229*, 1049–1064.
- [202] Turcotte, M.; Lapalme, G.; Major, F. Exploring the conformations of nucleic acids. *J. Funct. Program.*, **1995**, *5*, 443–460.
- [203] Erie, D.A.; Breslauer, K.J.; Olson, W.K. A Monte Carlo Method for Generating Structures of Short Single-Stranded DNA Sequences. *Biopolymers*, **1993**, *33*, 75–105.
- [204] Tung, C.-S.; Carter, E.S. II. Nucleic acid modeling tool (NAMOT): an interactive graphic tool for modeling nucleic acid structures. *CABIOS*, **1994**, *10*, 427–433.
- [205] Carter, E.S. II; Tung, C.-S. NAMOT2—a redesigned nucleic acid modeling tool: construction of non-canonical DNA structures. *CABIOS*, **1996**, *12*, 25–30.
- [206] Zhurkin, V.B.; P. Lysov, Yu.; Ivanov, V.I. Different Families of Double Stranded Conformations of DNA as Revealed by Computer Calculations. *Biopolymers*, **1978**, *17*, 277–312.
- [207] Lavery, R.; Zakrzewska, K.; Skelnar, H. JUMNA (junction minimisation of nucleic acids). *Comp. Phys. Commun.*, **1995**, *91*, 135–158.
- [208] Gabarro-Arpa, J.; Cognet, J.A.H.; Le Bret, M. Object Command Language: a formalism to build molecule models and to analyze structural parameters in macromolecules, with applications to nucleic acids. *J. Mol. Graph.*, **1992**, *10*, 166–173.
- [209] Le Bret, M.; Gabarro-Arpa, J.; Gilbert, J.C.; Lemarechal, C. MORCAD an object-oriented molecular modeling package. *J. Chim. Phys.*, **1991**, *88*, 2489–2496.

BIBLIOGRAPHY

- [210] Crippen, G.M.; Havel, T.F. *Distance Geometry and Molecular Conformation*. Research Studies Press, Taunton, England, 1988.
- [211] Spellmeyer, D.C.; Wong, A.K.; Bower, M.J.; Blaney, J.M. Conformational analysis using distance geometry methods. *J. Mol. Graph. Model.*, **1997**, 15, 18–36.
- [212] Hodsdon, M.E.; Ponder, J.W.; Cistola, D.P. The NMR solution structure of intestinal fatty acid-binding protein complexed with palmitate: Application of a novel distance geometry algorithm. *J. Mol. Biol.*, **1996**, 264, 585–602.
- [213] Macke, T.; Chen, S.-M.; Chazin, W.J. in *Structure and Function, Volume 1: Nucleic Acids*, Sarma, R.H.; Sarma, M.H., Eds., pp 213–227. Adenine Press, Albany, 1992.
- [214] Potts, B.C.M.; Smith, J.; Akke, M.; Macke, T.J.; Okazaki, K.; Hidaka, H.; Case, D.A.; Chazin, W.J. The structure of calcyclin reveals a novel homodimeric fold S100 Ca²⁺-binding proteins. *Nature Struct. Biol.*, **1995**, 2, 790–796.
- [215] Love, J.J.; Li, X.; Case, D.A.; Giese, K.; Grosschedl, R.; Wright, P.E. DNA recognition and bending by the architectural transcription factor LEF-1: NMR structure of the HMG domain complexed with DNA. *Nature*, **1995**, 376, 791–795.
- [216] Gurbiel, R.J.; Doan, P.E.; Gassner, G.T.; Macke, T.J.; Case, D.A.; Ohnishi, T.; Fee, J.A.; Ballou, D.P.; Hoffman, B.M. Active site structure of Rieske-type proteins: Electron nuclear double resonance studies of isotopically labeled phthalate dioxygenase from *Pseudomonas cepacia* and Rieske protein from *Rhodobacter capsulatus* and molecular modeling studies of a Rieske center. *Biochemistry*, **1996**, 35, 7834–7845.
- [217] Macke, T.J. *NAB, a Language for Molecular Manipulation*. Ph.D. thesis, The Scripps Research Institute, 1996.
- [218] Dickerson, R.E. Definitions and Nomenclature of Nucleic Acid Structure Parameters. *J. Biomol. Struct. Dyn.*, **1989**, 6, 627–634.
- [219] Zhurkin, V.B.; Raghunathan, G.; Ulyanov, N.B.; Camerini-Otero, R.D.; Jernigan, R.L. A Parallel DNA Triplex as a Model for the Intermediate in Homologous Recombination. *Journal of Molecular Biology*, **1994**, 239, 181–200.
- [220] Tan, R.; Harvey, S. Molecular Mechanics Model of Supercoiled DNA. *J. Mol. Biol.*, **1989**, 205, 573–591.
- [221] Havel, T.F.; Kuntz, I.D.; Crippen, G.M. The theory and practice of distance geometry. *Bull. Math. Biol.*, **1983**, 45, 665–720.
- [222] Havel, T.F. An evaluation of computational strategies for use in the determination of protein structure from distance constraints obtained by nuclear magnetic resonance. *Prog. Biophys. Mol. Biol.*, **1991**, 56, 43–78.
- [223] Kuszewski, J.; Nilges, M.; Brünger, A.T. Sampling and efficiency of metric matrix distance geometry: A novel partial metrization algorithm. *J. Biomolec. NMR*, **1992**, 2, 33–56.

- [224] deGroot, B.L.; van Aalten, D.M.F.; Scheek, R.M.; Amadei, A.; Vriend, G.; Berendsen, H.J.C. Prediction of protein conformational freedom from distance constraints. *Proteins*, **1997**, *29*, 240–251.
- [225] Agrafiotis, D.K. Stochastic Proximity Embedding. *J. Computat. Chem.*, **2003**, *24*, 1215–1221.
- [226] Saenger, W. in *Principles of Nucleic Acid Structure*, p 120. Springer-Verlag, New York, 1984.
- [227] Berendsen, H.J.C.; Postma, J.P.M.; van Gunsteren, W.F.; DiNola, A.; Haak, J.R. Molecular dynamics with coupling to an external bath. *J. Chem. Phys.*, **1984**, *81*, 3684–3690.
- [228] Loncharich, R.J.; Brooks, B.R.; Pastor, R.W. Langevin dynamics of peptides: The frictional dependence of isomerization rates of N-acetylananyl-N'-methylamide. *Biopolymers*, **1992**, *32*, 523–535.
- [229] Brooks, C.; Brünger, A.; Karplus, M. Active site dynamics in protein molecules: A stochastic boundary molecular-dynamics approach. *Biopolymers*, **1985**, *24*, 843–865.
- [230] Onufriev, A.; Bashford, D.; Case, D.A. Modification of the generalized Born model suitable for macromolecules. *J. Phys. Chem. B*, **2000**, *104*, 3712–3720.
- [231] Nguyen, D.T.; Case, D.A. On finding stationary states on large-molecule potential energy surfaces. *J. Phys. Chem.*, **1985**, *89*, 4020–4026.
- [232] Kolossváry, I.; Guida, W.C. Low mode search. An efficient, automated computational method for conformational analysis: Application to cyclic and acyclic alkanes and cyclic peptides. *J. Am. Chem. Soc.*, **1996**, *118*, 5011–5019.
- [233] Kolossváry, I.; Guida, W.C. Low-mode conformatinoal search elucidated: Application to C₃₉H₈₀ and flexible docking of 9-deazaguanine inhibitors into PNP. *J. Comput. Chem.*, **1999**, *20*, 1671–1684.
- [234] Kolossváry, I.; Keserü, G.M. Hessian-free low-mode conformational search for large-scale protein loop optimization: Application to c-jun N-terminal kinase JNK3. *J. Comput. Chem.*, **2001**, *22*, 21–30.
- [235] Keserü, G.M.; Kolossváry, I. Fully flexible low-mode docking: Application to induced fit in HIV integrase. *J. Am. Chem. Soc.*, **2001**, *123*, 12708–12709.
- [236] Shi, Z.J.; Shen, J. New inexact line search method for unconstrained optimization. *J. Optim. Theory Appl.*, **2005**, *127*, 425–446.
- [237] Press, W.H.; Flannery, B.P.; Teukolsky, S.A.; Vetterling, W.T. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, New York, 1989.
- [238] Liu, D.C.; Nocedal, J. On the limited memory method for large scale optimization. *Math. Programming B*, **1989**, *45*, 503–528.

BIBLIOGRAPHY

- [239] Nocedal, J.; L. Morales, J. Automatic preconditioning by limited memory quasi-Newton updating. *SIAM J. Opt.*, **2000**, *10*, 1079–1096.
- [240] Anandakrishnan, Ramu; Onufriev, Alexey V. An N log N approximation based on the natural organization of biomolecules for speeding up the computation of long range interactions. *J. Comput. Chem.*, **2010**, *31*(4), 691–706.
- [241] Anandakrishnan, R.; Daga, M.; Onufriev, A.V. An n log n Generalized Born Approximation. *J. Chem. Theory Comput.*, **2011**, *7*, 544–559.
- [242] Hirata, F., Ed. *Molecular Theory of Solvation*. Kluwer Academic Publishers, 2003.
- [243] Shao, J.; Tanner, S.W.; Thompson, N.; Cheatham, T.E. III. Clustering molecular dynamics trajectories: 1. Characterizing the performance of different clustering algorithms. *J. Chem. Theory Comput.*, **2007**, *3*, 2312–2334.
- [244] Prompers, J.J.; Brüschweiler, R. General framework for studying the dynamics of folded and unfolded proteins by NMR relaxation spectroscopy and MD simulation. *J. Am. Chem. Soc.*, **2002**, *124*, 4522–4534.
- [245] Prompers, J.J.; Brüschweiler, R. Dynamic and structural analysis of isotropically distributed molecular ensembles. *Proteins*, **2002**, *46*, 177–189.
- [246] Kollman, P.A.; Dixon, R.; Cornell, W.; Fox, T.; Chipot, C.; Pohorille, A. in *Computer Simulation of Biomolecular Systems, Vol. 3*, Wilkinson, A.; Weiner, P.; van Gunsteren, W.F., Eds., pp 83–96. Elsevier, 1997.
- [247] Beachy, M.D.; Friesner, R.A. Accurate ab initio quantum chemical determination of the relative energies of peptide conformations and assessment of empirical force fields. *J. Am. Chem. Soc.*, **1997**, *119*, 5908–5920.
- [248] Wang, L.; Duan, Y.; Shortle, R.; Imperiali, B.; Kollman, P.A. Study of the stability and unfolding mechanism of BBA1 by molecular dynamics simulations at different temperatures. *Prot. Sci.*, **1999**, *8*, 1292–1304.
- [249] Higo, J.; Ito, N.; Kuroda, M.; Ono, S.; Nakajima, N.; Nakamura, H. Energy landscape of a peptide consisting of α -helix, 3_{10} helix, β -turn, β -hairpin and other disordered conformations. *Prot. Sci.*, **2001**, *10*, 1160–1171.
- [250] Cheatham, T.E. III; Cieplak, P.; Kollman, P.A. A modified version of the Cornell et al. force field with improved sugar pucker phases and helical repeat. *J. Biomol. Struct. Dyn.*, **1999**, *16*, 845–862.
- [251] Weiner, S.J.; Kollman, P.A.; Case, D.A.; Singh, U.C.; Ghio, C.; Alagona, G.; Profeta, S. Jr.; Weiner, P. A new force field for molecular mechanical simulation of nucleic acids and proteins. *J. Am. Chem. Soc.*, **1984**, *106*, 765–784.
- [252] Weiner, S.J.; Kollman, P.A.; Nguyen, D.T.; Case, D.A. An all-atom force field for simulations of proteins and nucleic acids. *J. Comput. Chem.*, **1986**, *7*, 230–252.

- [253] Singh, U.C.; Weiner, S.J.; Kollman, P.A. Molecular dynamics simulations of d(C-G-C-G-A).d(T-C-G-C-G) with and without "hydrated" counterions. *Proc. Nat. Acad. Sci.*, **1985**, *82*, 755–759.
- [254] Wang, J.; Cieplak, P.; Kollman, P.A. How well does a restrained electrostatic potential (RESP) model perform in calculating conformational energies of organic and biological molecules? *J. Comput. Chem.*, **2000**, *21*, 1049–1074.
- [255] Aduri, R.; Psciuk, B.T.; Saro, P.; Taniga, H.; Schlegel, H.B.; SantaLucia, J. Jr. AMBER force field parameters for the naturally occurring modified nucleosides in RNA. *J. Chem. Theory Comput.*, **2007**, *3*, 1465–1475.
- [256] Cieplak, P.; Cornell, W.D.; Bayly, C.; Kollman, P.A. Application of the multimolecule and multiconformational RESP methodology to biopolymers: Charge derivation for DNA, RNA and proteins. *J. Comput. Chem.*, **1995**, *16*, 1357–1377.

Index

1D-RISM, [231](#), [233](#), [239](#)

3D-RISM, [231](#), [234](#), [243](#)

A

accept, [210](#), [422](#)

acdoctor, [107](#)

acos, [386](#)

activeref, [167](#)

adbcor, [242](#)

add, [65](#)

addAtomTypes, [66](#)

addIons, [66](#)

addIons2, [66](#)

addPath, [66](#)

addPdbAtomMap, [67](#)

addPdbResMap, [67](#)

addresidue, [338](#), [389](#)

addstrand, [338](#), [389](#)

alias, [68](#)

alignframe, [346](#), [397](#)

allatom_to_dna3, [392](#)

allocate, [373](#)

am1bcc, [102](#)

andbounds, [406](#)

angle, [181](#), [394](#)

anglep, [394](#)

antechamber, [94](#)

apply_rism_force, [425](#)

arcres, [209](#), [422](#)

asin, [386](#)

assert, [395](#)

asymptfile, [423](#)

atan, [386](#)

atan2, [386](#)

atof, [386](#)

atoi, [386](#)

atomicfluct, [181](#)

atommap, [177](#)

atomtype, [102](#)

average, [182](#)

avgcoord, [182](#)

B

basepair, [346](#)

bcopt, [211](#), [422](#)

bdna, [346](#), [449](#)

bdna(), [347](#)

blocksize, [426](#)

bond, [68](#)

bondByDistance, [68](#)

bondsearch, [171](#)

bondtype, [103](#)

break, [380](#)

bridge function, [232](#)

buffer, [424](#)

C

ceil, [386](#)

center, [178](#)

centering, [425](#)

check, [69](#), [183](#)

chgdist, [423](#)

closest, [178](#)

closure, [240](#), [423](#)

closureOrder, [242](#)

closureorder, [424](#)

cluster, [184](#)

clusterdihedral, [183](#)

combine, [69](#)

complement, [346](#)

conjgrad, [417](#)

connectres, [338](#), [389](#)

continue, [380](#)

copy, [70](#)

copymolecule, 389
 corr, 199
 cos, 386
 cosh, 386
 countmolatoms, 394
 createAtom, 70
 createResidue, 70
 createUnit, 70
 cut, 419
 cutfd, 212
 cutnb, 212, 422
 cuvfile, 423

D

datafile_create, 175
 datafile_invert, 176
 datafile_noheader, 176
 datafile_noxcol, 176
 datafile_precision, 177
 datafile_xlabel, 176
 date, 396
 dbfopt, 422
 deallocate, 373
 debug, 167, 395
 decompopt, 213
 del, 424
 delete, 378
 deleteBond, 70
 density, 241
 desc, 71
 dftb_3rd_order, 154
 dftb_chg, 154
 dftb_disper, 154
 dftb_maxiter, 154
 dftb_telec, 154
 dg_helix, 449
 dg_options, 407
 diag_routine, 156
 diel, 420
 dielc, 420
 dieps, 242
 dihedral, 186
 dim, 419
 direct correlation function, 231
 dist, 394

distance, 186
 distp, 394
 dna3, 392
 dna3_to_allatom, 392
 dprob, 208, 421
 dr, 240
 drmsd, 186
 dumpatom, 395
 dumpbounds, 395
 dumpboundsviolations, 395
 dumpmatrix, 395
 dumpmolecule, 395
 dumpresidue, 395
 DX, 254

E

e_debug, 419
 embed, 407
 eneopt, 211, 422
 epsext, 421
 epsin, 207, 421
 epsout, 207, 421
 errconv, 158
 espgen, 105
 excess chemical potential, 235
 exit, 385
 exp, 386
 extra_precision, 242

F

fabs, 386
 fclose, 387
 fd_helix, 391
 fillratio, 210, 422
 floor, 386
 fmod, 386
 fopen, 387
 fprintf, 387
 frcopt, 211, 422
 freemolecule, 389
 freeresidue, 389
 fscale, 210, 422
 fscanf, 387
 ftime, 396

INDEX

G

gamma_ln, 420
gauss, 386
Gaussian fluctuation, 235
gb, 420
gb2_debug, 419
gb_debug, 419
gbsa, 421
genmass, 420
geodesics, 407
getchivol, 407
getchivolp, 407
getcif, 392
getline, 387
getmatrix, 388
getpdb, 392
getpdb_prm, 417
getres, 339, 346
getresidue, 338, 339, 392
gettriad(), 363
getxv, 417
grdspc, 424
grms_tol, 157
groupSelectedAtoms, 72
gsub, 384
guvfile, 423

H

h1, 427
hbond, 187
hcp, 426
helix, 346
helixanal, 394
hist, 199
HNC, 232, 235
huvfile, 423
hypernetted-chain approximation, 232

I

image, 179
imin, 206
impose, 72
index, 384
inp, 206, 421
intramolecular pair correlation matrix, 233

ipb, 206, 421
iprob, 208, 421
irism, 423
istrng, 207, 421
itrmax, 157

J

jcoupling, 188

K

k4d, 419
kappa, 421
KH, 232, 235
Kovalenko-Hirata, 232
ksave, 241

L

length, 384
link_na, 390
linkprot, 390
list, 74
lmod, 438
loadAmberParams, 75
loadAmberPrep, 75
loadMol2, 75
loadOff, 75
loadPdb, 76
loadPdbUsingSeq, 76
log, 386
log10, 386
logFile, 76

M

mask, 189
match, 384
MAT_cube, etc, 399
matextract, 404
MAT_fprint, etc, 400
matgen, 401
matmerge, 403
maxarcdot, 422
maxcyc, 157
maxitn, 210, 422
maxsph, 214
maxstep, 241, 424
md, 417

- MDIIS, 234
- mdiis_del, 241
- mdiis_method, 424
- mdiis_nvec, 241, 424
- MDL, 249
- mean solvation force, 235
- measureGeom, 77
- mergestr, 338, 389
- mk_dimer(), 363
- mme, 417
- mme_rism_max_memory, 396
- mme_timer, 396
- mme_init, 417
- mme_rattle, 417
- mm_options, 417
- mm_set_checkpoint, 417
- model, 241
- modified direct inversion of the iterative sub-space, 234
- molsearch, 171
- molsurf, 190, 394
- MPI, 333
- N**
- NAB, 243, 246
- nastruct, 190
- nbuffer, 210
- nchk, 419
- nchk2, 419
- ndiis_attempts, 157
- ndiis_matrices, 157
- newbounds, 406
- newmolecule, 338, 389
- newton, 429
- newtransform, 344, 397
- nfocus, 210, 422
- ng, 424
- ngrdblxx, 215
- ngrdblky, 215
- ngrdblz, 215
- nkout, 241
- nmode, 429
- noexitonerror, 168
- noprogress, 168
- npbgrid, 210, 423
- npbopt, 209, 422
- npbverb, 213, 422
- npropagate, 237, 425
- nr, 240
- nrout, 241
- nscm, 419
- nsnb, 419
- nsnba, 212
- nsp, 242
- ntpr, 157, 419
- ntprism, 426
- ntpr_md, 420
- ntwrism, 425
- ntwx, 420
- ntx, 206
- nvec, 237
- O**
- offset, 214
- OMP_NUM_THREADS, 291, 333
- orbounds, 406
- Ornstein-Zernike, 231
- outlist, 240
- outtraj, 191
- P**
- pair-distribution function, 231
- parameterfle, 156
- parm, 169
- parmbondinfo, 170
- parmbbox, 170
- parmcals, 107
- parmchk, 97
- ParmEd, 310
- parminfo, 169
- parmmolinfo, 171
- parmresinfo, 171
- parmstrip, 170
- parmwrite, 170
- pbtemp, 208
- peptide_corr, 157
- phiform, 212
- plane, 394
- point, 383
- polardecomp, 237, 426

INDEX

pow, 386
prepgen, 104
printcharges, 156
printf, 387
progress, 241, 426
PSE-n, 232, 235
pseudo_diag, 155
pseudo_diag_criteria, 156
pucker, 191
putbnd, 392
putcif, 392
putdist, 392
putmatrix, 389
putpdb, 392
putxv, 417

Q

qmcharge, 154
qmmm_int, 158
qmqmdx, 154
qm_theory, 153
quvfile, 423
qxd, 156

R

radgyr, 192
radial, 192
radiopt, 208, 421
rand2, 386
rattle, 419
readparm, 417
reference, 175
remove, 77
residuegen, 107
respgen, 105
rgbmax, 421
rhow_effect, 214
RISM, 231
risml, 231, 239
rms2d, 195
rmsavgcorr, 193
rmsd, 193, 394
rot4, 345, 397
rot4p, 345, 397
rseed, 386

runavg, 180

S

saopt, 208
sasopt, 208
saveAmberParm, 77
saveMol2, 78
saveOff, 78
savePdb, 78
ScaLAPACK, 333
scalec, 211
scanf, 387
scfconv, 155
secstruct, 197
select, 168
selftest, 241
sequence, 78
set, 79
setbounds, 406
setboundsfromdb, 407
setchiplane, 407
setchivol, 407
setframe, 345, 397
setframep, 346, 397
setmol_from_xyz, 398
setmol_from_xyzw, 398
setpoint, 398
setseed, 386
setxyz_from_mol, 398
setxyzw_from_mol, 398
showbounds, 406
sin, 386
sinh, 386
smear, 242
smoothopt, 207, 421
solvateBox, 81
solvateCap, 81
solvateOct, 81
solvateShell, 82
solvation, 231, 235
solvation free energy, 235
solvbox, 424
solvcut, 424
solvopt, 209, 422
space, 210, 422

spin, 154
 split, 384
 sprintf, 387
 sprob, 213, 422
 sqrt, 386
 sscanf, 387
 static_arrays, 426
 strip, 180
 sub, 384
 substr, 384
 sugarpuckeranal, 394
 superimpose, 394
 surf, 198
 surfen, 214, 421
 system, 385
T
 t, 419
 tan, 386
 tanh, 386
 tautp, 419
 temp0, 420
 temperature, 242
 temperature_deriv, 240
 tempi, 420
 theory, 239
 thermodynamics, 235
 tight_p_conv, 155
 timeofday, 396
 tolerance, 241, 424
 torsion, 394
 torsionp, 394
 total correlation function, 231
 trajin, 171
 trajout, 173
 trans4, 345
 trans4p, 345
 transform, 72, 82, 404
 transformmol, 345, 398
 transformres, 338, 339, 345, 398
 translate, 83
 triopt, 208
 tsmooth, 407
U
 unlink, 387

unstrip, 180
 useboundsfrom, 406
 use_rmin, 213
 use_sav, 214
 uuvfile, 423

V

verbose, 426
 verbosity, 83, 155
 vlimit, 420
 volfmt, 423
 vprob, 213

W

wc_basepair, 449
 wc_basepair(), 349
 wc_complement, 449
 wc_complement(), 347
 wc_helix, 449
 wc_helix(), 352
 wcons, 419

X

xmax, 215
 xmin, 215, 433
 XVV, 250
 xvfile, 423

Y

ymax, 215
 ymin, 215

Z

zerofrc, 425
 zeroV, 420
 zMatrix, 83
 zmax, 215
 zmin, 215

INDEX

A. ptraj

`ptraj` is really two interfaces contained within the same C source code:

1. `rdparm`: a program to read and help interpret Amber `prmtop` files

usage: `rdparm prmtop`

2. `ptraj` (and the MPI parallelized compilation of the code, `ptraj.MPI`): programs to process and analyze a series of 3-D atomic coordinates (one molecular configuration or *frame* at a time). Molecular information, such as atom and residue names, is loaded from the file `prmtop` and this file can be an Amber `prmtop`, CHARMM PSF or PDB file. Note that the input atomic coordinates must be in the same order as the atoms stored in the molecular information file (i.e., `prmtop`).

usage: `ptraj prmtop script` or `ptraj prmtop < script >& ptraj.out`

The interface used at runtime by default is `ptraj`, unless the executable is named “`rdparm`”. `rdparm` is interactive – type “?” or “help” for a list of commands – and only supports the reading of Amber `prmtop` files. If the executable name does not contain the string “`rdparm`”, `ptraj` is run instead. Commands to `ptraj` can either be piped in through standard input or from a file (*script*). To save runtime information from `ptraj` it is often convenient to pipe the standard output and error to a file (`>& ptraj.out`). The code is documented and can be extended by users. Information absent from this manual can often be found by consulting the code directly. An example input script is shown below:

```
trajin traj1.gz 1 20 1
trajin traj2.Z 2 100 2
trajin restrt.bz2
trajout fixed.traj nobox
center :1-20 mass origin
image origin center
rms first out rms @CA,C,N
strip :WAT
average avg.pdb pdb
go
```

The above `ptraj` script reads in three sets of coordinates (specified with the `trajin` keyword), i.e., `traj1.gz` frames 1 to 20, `traj2.Z` frames 2, 4, 6, ..., 100, and the single frame of `restrt.bz2`. A variety of file types are supported including Amber ascii and NetCDF trajectories and restart files, CHARMM DCD files of either endian order, PDB files, and binpos

A. *ptraj*

binary files. For each frame of coordinates, a series of *actions* are performed, in the order specified, and a new trajectory file (`fixed.traj`) of the processed coordinates is output without any periodic box information. The actions performed include centering the coordinates, periodically imaging the coordinates to the primary unit cell, rms fitting of the coordinates to the first frame, deleting the coordinates for residues named WAT, and accumulation of an average structure over all the frames. Atom selections, for example specifying the list of atoms to be used in the rms fitting (i.e., @CA, C, N), are chosen using the Amber mask syntax which is similar to that used in Midas/Chimera. Note that some commands alter the coordinates, and therefore the ordering of action commands is important. For example, since rotation information is not stored at present when rms fitting, imaging should be performed prior to rms fitting otherwise the periodic box could rotate out of its reference frame.

A.1. Understanding *ptraj* commands and actions

In the discussion that follows commands are listed in **bold** type. Words in *italics* are values that need to be specified by the user, and words in *typewriter* text are keywords to specify an option (which may or may not be followed by a value). In the specification of the commands, arguments in square brackets ([]) are optional and the "|" character represents "or". Arguments that are not in square brackets are required. In general, if there is an error in processing a particular action, that action will be ignored and the user warned (rather than terminating the program), so check the printed warning's carefully. A few of the commonly used argument types are described below:

mask: this is used to specify a list of atoms. The syntax for specifying atoms is equivalent to the Amber mask syntax (see `ambmask` as described in the miscellaneous section of the manual) and also supports most of the MidasPlus/Chimera syntax. Note that spaces are only interpreted if the mask is surrounded by double quotes (" "). The "@" character selects atom names or numbers, the ":" character selects residue names or numbers, the "-" character represents a continuation, the ";" character allows specifying multiple names or numbers, and the "*" and "?" commands are wildcard matches for multiple or single characters, respectively. To understand or check what atoms are selected, the `checkmask` command to `rdparm` can be applied.

filename: this refers to the name of the file, relative to the current directory (unless the full path is provided). No checking is done for existing files, so be careful when specifying output files as existing files of the same name will be overwritten.

commands: each command effectively needs to be on a single line and values separated by whitespace are considered as different tokens. If you have a *mask* with spaces, be sure to wrap the specification in quotes. If you want to break up a command over multiple lines, use a continuation character "\".

The following sections discuss input and output commands to *ptraj*, commands that modify the state (such as the box size or definition of solvent atoms), and each of the defined action commands.

A.2. ptraj coordinate input/output commands

trajin *filename* [*start stop offset*] [remdtraj remdtrajtemp reptemp]

Load the trajectory file specified by *filename* using only the frames starting with *start* (default 1) and ending with (and including) *stop* (default, the final configuration) using an offset of *offset* (default 1) if specified. Amber trajectory, Amber restart, PDB, binpos, CHARMM DCD, and Amber REMD trajectory files are all currently supported. To read an Amber REMD trajectory at a particular temperature from an ensemble of replica trajectories, specify the `remdtraj remdtrajtemp` keywords followed immediately by the replica temperature (*reptemp*, default 0.0), and then files will be searched and the final trajectory found at that particular temperature will be used. Note that in this case, **trajin** assumes the Amber REMD files are named according to the convention that the end of the filename includes “.N”, “.N.gz” or “.N.bz2” where N is the number of replicas ending specification of the replica number (i.e., NNN = 000, 001, ...). An example call is “`trajin remd.000.gz remdtraj remdtrajtemp 300.0`” which will search all existing files at that path location named `remd.N.gz` for the final replica at 300.0 degrees. Filenames with an appended .Z or .gz or .bz2 are also recognized and treated appropriately, with the exception of compressed Amber NetCDF files (and .Z compressed Amber REMD files). Finally, please note that the coordinates *must* match the names/ordering of the molecular information file (`prmtop`) previously read in.

reference *filename*

Load up the first coordinate set from the trajectory specified by the file named *filename* and save this for use as a reference structure. Currently the only action command to use the reference structure is the **rms** command. Note that as the state is modified (for example by **strip** or **closestwaters**), the reference coordinates are also modified internally. If multiple **reference** commands are specified, the structure referenced is the one loaded previous to the current **rms** command. For example, if you wanted to calculate the rms deviation to two different pdb structures (`one.pdb` and `two.pdb`) you could use the following script:

```
trajin mdcrd
reference one.pdb
rms reference out rms-to-one.dat
reference two.pdb
rms reference out rms-to-two.dat
```

Note that it is possible for the reference coordinate set to be incomplete (for example an unsolvated protein). Although a warning is printed, as long as the RMS command does not refer to the missing coordinates and there is still a 1-to-1 mapping between the reference and actual coordinates loaded and the atoms to fit are within this set, the RMS fit is valid.

trajout *filename* [*format*] [nobox] [nowrap] [append] [remdtraj] [les
split|average] [little | big] \
[dumpq| parse] [title *title*] [application *application*] [program *program*]

A. ptraj

filename [*format*]: Specify the name of a file for output coordinates (*filename*) written in a specific format (*format*). Currently supported formats are:

- `trajectory` – Amber ascii trajectory, *the default*
- `restart` – Amber restart
- `binpos` – Scripps binary format
- `pdb` – PDB, the traditional format (not the newer CIF files); if molecule information is present, TER cards will be written between molecules.
- `cdf | netcdf` – Amber NetCDF binary trajectory
- `charmm` – CHARMM DCD binary trajectory

Note that the allowable formats include both trajectory files (i.e., a series of frames) and files that traditionally include only a single coordinate set. In this latter case, the *filename* will be appended with *.N* where *N* is the frame number (unless the optional keyword `append` is specified).

- `nobox`: This keyword suppresses the output of box information in AMBER trajectory and restart file formats. If you read in a trajectory with periodic box information, the box information will automatically be printed in an output trajectory. This is not always desirable. For example, if you are stripping solvent from the trajectory (`strip :WAT`), you may want to specify the “`nobox`” option so that the trajectory can be later processed with a corresponding non-periodic prmtop (or PDB) file. *For non-periodic trajectories, always specify `nobox`.*
- `nowrap`: This option is only of relevance to PDB files and prevents automatically wrapping the fourth character of the atom name to the first column (as is normally done automatically per PDB guidelines).
- `append`: For single set coordinate file outputs (for example PDB), specification of “`append`” will suppress creation of separate files and put each frame into the same file (*filename*) with TER and ENDMDL cards between coordinate sets. Append is not applicable to the restart format.
- `remdtraj`: This keyword, only applicable to Amber trajectory formats (trajectory and netcdf), adds Amber specific temperature replica exchange MD temperature information consistent with the current replica temperature. If the input trajectory has no temperature information, a temperature of 0.0 will be written in the output trajectory. A common use of this functionality is to convert Amber ascii trajectories to NetCDF format. Note that the replica number, mdstep and exchange# fields of the REMD section of the Amber ascii REMD trajectory are not preserved and will be converted to 0, the frame number, and the frame number, respectively.
- `les split | append`: The `les` keyword is used for the merging or splitting of Amber formatted locally enhanced sampling (LES) trajectories. The “`split`” keyword will output *N* separate trajectories, one for each LES group (where *N* is the copy number) whereas “`average`” will output a single averaged trajectory. Note that to subsequently process the split or merged LES trajectories, the corresponding non-LES prmtop (without extra copies) is required.

A.3. ptraj commands that override the molecular information specified

- `little` | `big`: These keywords can only be used with the CHARMM DCD output to specify the byte ordering as “little” or “big” endian; by default the endian order of the first CHARMM DCD trajectory read in is preserved.
- `dumpq` | `parse`: This option is only of relevance to PDB files and allows dumping of charges and radii into the temperature and occupancy columns of the PDB file (similar to a PQR file). With “dumpq” Amber charges and van der Waals radii (*not* the generalized Born radii!) are output. With “parse” Amber charges and parse radii are output. A current limitation of this implementation is that the implicit solvent radii specified in the Amber prmtop file (%FLAG RADII) cannot be dumped automatically at present.
- `title`, `application`, `program`: When comments are allowed in the output trajectory, optional title, application and program names can be specified as text strings (without any spaces).

Extra note on the CHARMM DCD trajectory output: If periodic box information is present in the CHARMM trajectory file, the symmetric box information output in a new CHARMM trajectory file (in versions > 22) will be *very* slightly different due to numerical issues in the diagonalization procedure; this will not effect analysis but shows up when diffing the binary files.

Limitations: Note that only a single `trajout` command is currently supported and it outputs the coordinates after all action commands have been processed. In the future look for the command `trajout-action` which will provide the same support but allow multiple context dependent specifications.

A.3. ptraj commands that override the molecular information specified

These commands change the state of the system, such as to define the solvent or alter the box information.

box [`x value`] [`y value`] [`z value`] [`alpha value`] [`beta value`] [`gamma value`] \
[`fixx`] [`fixy`] [`fixz`] [`fixalpha`] [`fixbeta`] [`fixgamma`]

This command allows specification and optionally fixing of the periodic box (unit cell) dimensions. This can be useful when reading PDB files that do not contain box information, trajectory formats that do not support non-standard triclinic cells, or to override the box information in the trajectory file. For example, if you wanted to process the coordinates with average values rather than the instantaneous box coordinates for each frame. The `x`, `y`, and `z` keywords change the box size (in Å) and the `alpha`, `beta` and `gamma` values change the angles of the triclinic unit cell. In the standard usage, without the “fix” keywords, if the box information is not already present in the input trajectory (such as the case with restart files or trajectory files) this command can be used to set the default values that will be applied. If you want to force a particular box size or shape, the `fixx`, `fixy`, etc commands can be used to override any box information already present in the input coordinate files. For example, the following

A. ptraj

command will set the x-component of the box size to be 100.0 Å and fix its value throughout the trajectory:

```
box x 100.0 fixx
```

solvent [byres | byname] *mask1* [*mask2*] [*mask3*] ...

This command can be used to override the solvent information specified in the Amber prmtop file or that which is set by default (based on residue name) upon reading a CHARMM PSF file. Applying this command overwrites any previously set solvent definitions. The solvent can be selected by residue with the “byres” modifier using all the residues specified in the one or more atom masks listed. The byname option searches for solvent by residue name (where the mask contains the name of the residue), searching over all residues.

As an example, say you want to select the solvent to be all residues from 20-100, then you would do

```
solvent byres :20-100
```

Note that if you don’t know the final residue number of your system offhand, yet you do know that the solvent spans all residues starting at residue 20 until the end of the system, just chose an upper bound and the program will reset accordingly, i.e.,

```
solvent byres :20-999999
```

To select all residues named "WAT" and "TIP3" and "ST2":

```
solvent byname WAT TIP3 ST2
```

Note that if you just want to peruse the current solvent information (or, more generally, to obtain information about the current state), specify **solvent** with no arguments and a summary of the current state will be printed.

Other commands which also modify the molecular information are **strip** and **closest**. These commands are described in the next section since they also modify the coordinates.

A.4. ptraj *action* commands - short list

A.5. ptraj *action* commands - detailed discussion

The following are commands that involve an *action* performed on each coordinate set as it is read in. The commands are listed in alphabetical order, and are summarized in Table A.1. Note that when ptraj processes the list of commands, they are applied in the order specified. Some of these may change the overall state or molecular information (i.e., the list of active atoms; more on this later). All of the actions can be applied repeatedly. Note that in general (except where otherwise mentioned) implicit imaging in non-orthorhombic systems (for example of distances) is supported.

A.5. ptraj action commands - detailed discussion

<i>action</i>	<i>description of the action</i>
analyze	calculate summary properties for values accumulated during analysis, such as distances, etc.
angle	compute the angle in degrees between the center of mass of the three atom specifications listed
atomicfluct	compute the atomic positional fluctuations (RMSF) or B-factors: Note this does not implicitly perform a RMS fit prior
average	compute the straight coordinate average of the coordinate sets read in
center	move all the coordinates to the origin or box center
checkoverlap	look for atoms that are closer (and optionally further separated than) a specified distance
closest	strip the trajectories such that only the closest N (chosen by the user) solvent molecules are retained
cluster	group configurations from the MD trajectory into distinct sets
clusterdihedral	group configurations from the MD trajectory by binning dihedral angles; output can be used with RREMD in SANDER
contacts	find the number of atom interactions (and native contacts) within a given distance
dihedral	calculate the dihedral values from the center of mass of the four specified atom masks
diffusion	calculate the mean-squared displacements vs. frame number
distance	calculated the distance in Å between the center of mass of the two atom specifications listed
grid	create a grid in X-Plor density format that has a count of the number of selected atoms in each grid cell
hbond	keeps track of distances and angles between triplets of atoms
image	with periodic boundary conditions, bring molecules outside the periodic box into the central unit cell
matrix	calculate 2D correlations and fluctuations
principal	align to the principal coordinate axis
pucker	compute the pucker for 5-membered rings
radial	compute a radial distribution function
radgyr	determine the radius of gyration for each frame
randomizeions	move single atom ion residues to new positions by swapping with random water molecules
rms	compute the RMSd to the first frame or a reference frame
secstruct	determine the secondary structure of proteins
strip	remove atoms from the trajectory
translate	move the entire system in the x, y and/or z directions
trunctoct	create an old style AMBER truncated octahedron; this is obsolete
unwrap	with periodic boundary, unwrap molecules to generate continuous trajectories
watershell	determine how many solvent models are within a given radius of the solute
vector	calculate and analyze various different vectors

Table A.1.: Summary of ptraj action commands

A. ptraj

A number of the commands store derived data internally, such as per frame RMSd values or angles. This data can be analyzed or processed later (see the information on the **analyze** command). To provide specification of the data for each data generating command, a unique *name* is associated with the data. See the examples for the **angle** command below for clarification.

angle *name mask1 mask2 mask3* [*out filename*] [*time interval*]

Calculate the angle between the three atoms listed, each specified in a separate mask, *mask1* through *mask3*. If more than one atom is listed in each mask, then the center of mass of the atoms in that mask is used. Note that masks that include spaces must be enclosed in quotation marks. The results are saved internally with the name *name* (which must be unique) on the scalarStack for later processing (with the **analyze** command). Data will be dumped to a file named *filename* if “out” is specified in two columns (time and angle) with a time interval between configurations of *interval* if “time” is listed (in picoseconds, default is 1.0); for example if the time between frames is 5 ps, specify *time 5.0*). All the angles are stored in degrees. For example, two **angle** commands are shown below. The first simply calculates the angles between three atoms in residue #1, namely CA, C, and N. The second calculates the angle between the center of mass of residues 1, 2 and 3. Each are referenced into an internal name space (*a1* and *angle-name2*). The second command also outputs the data to a file named *angle2.dat* with the x-axis (time) scaled by 10, essentially meaning a spacing of 10 ps per frame of data. See the command **analyze statistics all** to print averages and standard deviations...

```
angle a1 :1@CA :1@C :1@N
angle angle-name2 :1 :2 :3 out angle2.dat time 10.0
```

atomicfluct [*out filename*] [*mask*] [*start start*] [*stop stop*] [*offset offset*] [*byres*
| *byatom* | *bymask*] [*bfactor*]

Compute the atomic positional fluctuations for all the atoms; output is performed only for the atoms specified in the *mask*. If “byatom” is specified, dump the calculated fluctuations by atom (default). If “byres” is specified, dump the average (mass-weighted) for each residue. If “bymask” is specified, dump the average (mass-weighted) over all the atoms in the original *mask*. If “out” is specified, the data will be dumped to *filename* (otherwise the values will be dumped to the standard output). The optional “start”, “stop”, and “offset” keywords are specified, they should follow with specification of a value that respectively represents the first, last and offset for coordinate processing of the coordinates read in. This paring down of snapshots acts on top of, or in addition to, any offsets specified with “*trajin*”. If the keyword “bfactor” is specified, the data is output as B-factors rather than atomic positional fluctuations (which simply means multiplying the *squared* fluctuations by $\frac{8}{3}\pi^2$). The data is dumped into two columns (n and value) where n goes from 1 to the number of atoms or groups specified and the value is the appropriate RMSF (Å) or B-factor ($\text{\AA}^2 \times \frac{8}{3}\pi$).

So, to dump the mass-weighted B-factors for the protein backbone atoms C, CA, and N, by residue use the command:

```
atomicfluct out back.apf @C,CA,N byres bfactor
```


To dump the RMSF or atomic positional fluctuations of the same atoms, use the command:

```
atomicfluct out backbone-atoms.apf @C,CA,N
```

Note that RMS fitting is not done implicitly. If you want fluctuations without rotations or translations (for example to the average structure), perform an RMS fit to the average structure (best) or the first structure (see **rms**) prior to this calculation.

```
average filename [ mask ] [ start start ] [ stop stop ] [ offset offset ] [ nobox ] \  
[ pdb [ parse | dumpq ] [ nowrap ] | binpos | rest ] [ stddev ]
```

Compute the average structure over all the configurations read in (subject to values specified for *start*, *stop* and *offset*, if set, noting that the number is relative to the order of frames read in and acts on top of, or in addition to, any such values specified with *trajin*). The resulting structure is written (or appended if the optional keyword “append” is provided) to a file named *filename*. The optional *mask* trims the output coordinates (but does not change the state or alter the coordinates in the action stream as they are processed). If you want to alter the coordinates on the action stream via averaging, use the **runningaverage** command.

nobox: If specified, the box information is not written to file formats that accept box information (i.e., AMBER trajectory, *binpos* or *rest*). This is very important if you are processing non-periodic systems—or are generating solvent stripped trajectories—since otherwise after each coordinate set the box coordinates are written. If you subsequently try to process the newly written trajectory with a non-periodic *prmtop* file, only the first frame will be read in correctly (due to the offsets).

File formats: At present only four output file types are supported, specifically the AMBER trajectory (the default), PDB (if *pdb* is specified), AMBER *binpos* (if *binpos* is specified) or AMBER restart (if *rest* is specified). Additional options are supported for the PDB format: *nowrap* prevents movement of the fourth character of the atom name to the first column (as is normally done in the PDB standard format), *parse* will write parse radii and AMBER charges to the temperature and occupancy columns, and *dumpq* will write AMBER van der Waals radii (*r**) and charges to the temperature and occupancy columns. A current limitation in this command is that there is not an option to dump the actual GB radii stored in the AMBER *prmtop* files.

stddev: Write out the standard deviations (fluctuations) instead of the average coordinates.

An example below creates the file “average_1-2ns.pdb” in PDB format, average over all atoms named CA, and starts and ends with the 1000th and 2000th frames read in, respectively:

```
average average_1-2ns.pdb @CA start 1000 stop 2000 pdb
```

```
center [ mask ] [ origin ] [ mass ]
```

This action moves all of the atoms in the system to the origin or box center. With the *mass* keyword, the center of mass is moved to the periodic box center or origin, otherwise the center of the coordinates is moved. If the optional *mask* is specified, all of the coordinates

A. ptraj

are translated to place the center of the selected coordinates at the origin or box center. If `origin` is specified, periodic coordinates are moved to the zero of coordinates rather than the box center. With non-periodic trajectories, the center is always at the origin or zero of the coordinates. By default, for periodic trajectories the center is the center of the periodic unit cell.

Note: Moving all of the coordinates to the center can hide systematic motion in a particular direction; in the mid-1990's, we automatically centered and RMS fit all of our trajectories after a long set of runs and looked at movies of these trajectories stripped of solvent rather than the raw trajectories. This hid a problem resulting from a potential energy drain and velocity scaling for temperature control that led to build-up of translational motion. The so-called “flying box of ice” is described in *J. Comp. Chem.* 19, 726-740 (1998) and later by another group in *J. Comp. Chem.* 21, 121-131 (2000). Many MD trajectories were subsequently thrown away... This problem was fixed and AMBER now conserves energy in routine use. *Do remember that processed trajectories may not be fully representative of, or may hide motions in, the raw data!*

The following command will place the center of mass of all coordinates at the origin.

```
center mass origin
```

If you want the protein atoms at the origin (so you can subsequently **image** the trajectory to put solvent around the protein using `image origin center`), use:

```
center @CA mass origin
```

```
checkoverlap [ mask ] [ min value ] [ max value ] [ noimage ] [ around mask ]
```

Look for pair distances in the *mask* selected atoms (all by default) that are less than the specified minimum value (in Å, min 0.95 Å by default) apart or greater than the maximum value (if specified with *max*). The “around” keyword can be used to limit search of pair distances only around a selected set of atoms. This command is extremely computationally demanding, particularly if imaging is turned on (by default; imaging can be turned off with `noimage`), so expect to wait a while.

This command is extremely useful for diagnosing problems in input coordinates related to poor model building, such as to find overlapping atoms that can lead to infinite van der Waals or electrostatic energies. An example below looks for overlap of atoms in residues named Na+ and K+:

```
checkoverlap @Na+,K+
```

To look over atoms with a distance less than 1.2 Å between any atom in residues 1 – 20 with any other atom:

```
checkoverlap min 1.2 around :1-20
```

```
closest total mask [ oxygen | first ] [ noimage ]
```

This command can be used to retain only the closest solvent molecules (based on the number specified by the integer *total*) to the atoms specified in the *mask*. By default, this means residues named “WAT”. The definition of solvent can be changed using the **solvent** command (defined previously in section 5.3), for example to save only the closest

set of *total* ions. If “oxygen” or “first” are specified, only the distance to the first atom in the solvent molecule (to each atom in the mask) is measured. This command is rather time consuming since many distances need to be measured. Note that imaging is implicitly performed on the distances and this gets extremely expensive in non-orthorhombic systems due to the need to possibly check all the distances of the nearest images (up to 26!). Imaging can be disabled by specifying the “noimage” keyword.

Note that the solvent residue numbers are not preserved and are ordered at output such that the closest solvent is first. This means that water 1000 is not always the same water molecule and therefore you cannot use commands like **diffusion** that depend on unchanging residue numbers for each solvent. Like the **strip** command, this modifies the current state and changes the coordinates for subsequent actions. A restriction of this command is that each of the solvent molecules must have the same number of atoms; this leads to a fixed size “configuration” in each coordinate set output which is necessary for most of the file formats and avoids really complicating the code.

Of course, say you have two solvents of differing sizes and you want to perform **closest** for each of these, this can be done sequentially. For example, if we have both ethanol “:ETH” and water “:WAT” solvent residues present in the system and you want to save the closest 50 solvent molecules of each to residues :1-20, then the following commands could be applied (noting that both **closestwater** and **closest** are recognized as the same command):

```
solvent byres :WAT
closestwater 50 :1-20 first
solvent byres :ETH
closestwater 50 :1-20 first
```

Note that to further process the output coordinates later with ptraj or other programs, you will need to generate a corresponding prmtop, PDB or PSF file. We have used this command in practice to save only a small number of close waters or ions to nucleic acids for explicit representation of these waters or ions in MM-PBSA calculations; see for example *J. Amer. Chem. Soc.* 125, 1759-1769 (2003) for use with water around drugs in the minor groove of DNA and *Biophys. J.* 85, 1787-1804 (2003) for use with ions around DNA quadruplexes.

```
cluster out filename [ representative format ] [ average format ] [ all format ] algorithm
[ clusters n | epsilon critical_distance ] \
[ rms | dme ] [ sieve s [ start start_frame | random ] ] [ verbose verb ] [ mass ] mask
```

Clustering refers to grouping together similar objects. In the context of ptraj, this means grouping together coordinate frames from the trajectory into distinct sets. Several different algorithms for clustering have been implemented. The most common similarity metric is RMSd (specified by the `rms` keyword). Distance matrix error is also a potential similarity metric (with keyword `dme`), however this is considerably more computationally demanding. It is now also possible to cluster by attribute, i.e., the values of time series measured, and will be discussed later. The ideas used here are discussed in considerable detail in Ref. [243], and users should consult that paper for background and details. A simple example is as follows:

A. ptraj

```
trajin traj.1.gz
trajin traj.2.gz
cluster out testcluster representative pdb \
        average pdb averagelinkage clusters 5 rms @CA
```

The above reads in two trajectory files and then clusters using the average-linkage algorithm to produce 5 clusters using the pairwise RMSd between frames as a metric comparing the atoms named CA. PDB files are dumped for the average and representative structures from the clusters and full trajectories (over ALL atoms) are dumped in AMBER format. If you only want to output only the CA atoms, the strip command could be applied prior. The files output will be prefixed with “testcluster”.

Output information will be dumped to a series of files prefixed with *filename*. *filename.txt* contains the clustering results and statistics. “*filename.rep.ci*” contains the representative structure of cluster *i* with its specified format ($i = 0$ to $n - 1$). “*filename.avg.ci*” contains the average structure of cluster *i* with its specified format. “*filename.ci*” contains all the frames in the cluster *i*-1 with specified format. Available *formats* include “none”, “pdb”, “rest”, “binpos”, or “amber”. The default format is the “amber” trajectory.

Algorithms implemented in the ptraj include *averagelinkage*, *linkage*, *complete*, *edge*, *centripetal*, *centripetalcomplete*, *hierarchical*, *means*, *SOM*, *COBWEB*, and *Bayesian*. Please see Ref. [243] for more details on the advantages and disadvantages of each algorithm. For *averagelinkage*, *linkage*, *complete*, *edge*, *centripetal*, *centripetalcomplete*, and *hierarchical*, the user can specify a critical distance so that the clustering will stop when this distance is met. All algorithms will try to generate n clusters. However, sometimes SOM and Bayesian algorithms will generate less than n clusters and this may indicate a more reasonable number of clusters of the trajectory.

The distance metric can be *rms* or *dme* (distance matrix error). Users are encouraged to use *rms* since *dme* is significantly more computationally demanding yet returns similar results. *rms* is the default value. The keyword *mass* indicates the *rms* or *dme* matrix will be mass-weighted. The users are advised to always turn this “*mass*” option on. *Mask* is the atom selection where the clustering method is focused.

The sieve keyword is useful when dealing with large trajectories. The command “*sieve s*” tells ptraj to cluster every *sth* frame in the first pass. The default sieve size is 0 (equivalent to sieve 1). The user can state where the first frame will be picked for the first pass by specifying the parameter *start_frame*. The default value of *start_frame* is 1. To avoid the potential problem of periodicity, frames can be picked randomly if the keyword “*random*” is specified. Since the coordinates of unsampled frames are not saved during the process, the DBI and pSF values can not be calculated for the whole trajectory, although those values for the first pass will be saved in a file called “EndFirstPass.txt”. The DBI and pSF values for a sieving algorithm can be calculated later by running the ptraj clustering again, using “DBI” as the algorithm. This will read the clustering result from the “*filename.txt*” and appended the DBI and pSF values to the file “*filename.txt*”.

The cluster facility will calculate a pairwise distance matrix between each pair of frames and save the matrix in a file called “PairwiseDistances”. This file will be read in (and checked) for clustering if it is found in the current directory. Although not all algorithms

require this distance matrix, this matrix will be helpful for the calculation of DBI and pSF in the post-clustering process. In the case of sieving, the file “PairwiseDistance” will be generated for just those sampled frames in the first pass. A user provided “FullPairwise-Matrix” containing a full pairwise matrix would further expedite the calculation of DBI and pSF.

For the COBWEB algorithm, a special file “CobwebPreCoalesce.txt” will be saved for the COBWEB tree structures. The first level of branches usually indicates the natural clustering. Use “clusters -1” (minus one) will achieve this natural clustering. If the specified number of clusters, n , is not equal to its natural number of clusters, branches will be merged or split. COBWEB will read a pre-written CobwebPreCoalesce.txt if it found in the current directory. Another special parameter for COBWEB is [acuity *acu*]. Acuity is set to be the minimal standard deviation of a cluster attribute. The default value of acuity is 0.1.

For the agglomerative algorithms, specifically averagelinkage, linkage, complete, edge, centripetal, and centripetalcomplete, every merging step will be saved in the file “ClusterMerging.txt”. This file can be read in to generate other number of clusters by using “ReadMerge” as the cluster algorithm in the ptraj command. For each line, the first field is the newly formed cluster, which is followed by the two fields representing the sub-clusters. The fourth field is the current critical distance, which is followed by (the DBI and) pSF values. The DBI values are omitted if the number of clusters is greater than 50 because the time to calculate DBI is intractable as cluster number increases. Obviously, this will not yield less clusters (i.e., more merging steps) than the clustering when the ClusterMerging.txt file is generated. Therefore, the users can use “clusters 1” at first for these algorithms, and then generate other number of clusters by ReadMerge.

Some parameters are designed for specific algorithms. The [iteration *iter*] parameter is used in the means algorithm which specifies the maximum iteration for the refinement steps. The default value of iteration is 100. There is a variation of means algorithm, decoy. The “decoy” method allows the users to provide seed structures for the means algorithm. Use “decoy *decoy_structure*” as the algorithm to provide the initial structures in a trajectory file “decoy_structure”. If the users want the real decoy by providing the well-defined structures, “iteration 1” can be used to prevent subsequent refinement.

clusterdihedral *out filename cut clustersize_cutoff framefile filename clusterinfo filename*
name
 [dihedralfile *filename*] | [phibins *bins* psibins *bins* mask]

Cluster frames in a trajectory using dihedral angles. To define which dihedral angles will be used for clustering either an atom mask or an input file specified by the dihedralfile keyword should be used. If dihedralfile is used, each line in the file should contain a dihedral to be binned with format:

ATOM#1 ATOM#2 ATOM#3 ATOM#4 #BINS

where the ATOM arguments are the atom numbers defining the dihedral and #BINS is the number of bins to be used (so if #BINS=10 the width of each bin will be 36°. If an atom mask is specified, only protein backbone dihedrals (Phi and Psi defined using atom names

A. ptraj

C-N-CA-C and N-CA-C-N) within the mask will be used, with the bin sizes specified by the phibins and psibins keywords (default for each is 10 bins).

Output will either be written to the terminal or the file specified by the out keyword.

First information about which dihedrals were clustered will be printed. Then the number of clusters will be printed, followed by detailed information of each cluster. The clusters are sorted from most populated to least populated. Each cluster line has format

```
Cluster CLUSTERNUM CLUSTERPOP [ dihedral1bin, dihedral2bin ... dihedralNbin ]
```

followed by a list of frame numbers that belong to that cluster.

If specified by the framefile keyword a file containing cluster information for each frame will be written with format

```
Frame CLUSTERNUM CLUSTERSIZE DIHEDRALBINID
```

where DIHEDRALBINID is a number that identifies the unique combination of dihedral bins this cluster belongs to (specifically it is a 3*number-of-dihedral-characters long number composed of the individual dihedral bins).

If specified by the clusterinfo keyword a file containing information on each dihedral and each cluster will be printed. This file can be read by SANDER for use with REMD with a structure reservoir (-rremd=3). The file, which is essentially a simplified version of the main output file, has the following format:

```
#DIHEDRALS
dihedral1_atom1 dihedral1_atom2 dihedral1_atom3 dihedral1_atom4
...
#CLUSTERS
CLUSTERNUM1 CLUSTERSIZE1 DIHEDRALBINID1
...
```

If a cutoff is specified by CUT only clusters with population greater than CUT will be printed.

contacts [first|reference] [byresidue] [out *filename*] [time *interval*] [distance *cutoff*] [*mask*]

For each atom given in *mask*, calculate the number of other atoms (contacts) within the distance *cutoff*. The default cutoff is 7.0 Å. Only atoms in *mask* are potential interaction partners (e.g., a mask @CA will evaluate only contacts between CA atoms). The results are dumped to *filename* if the keyword “out” is specified. Thereby, the time between snapshots is taken to be *interval*. In addition to the number of overall contacts, the number of native contacts is also determined. Native contacts are those that have been found either in the first snapshot of the trajectory (if the keyword “first” is specified) or in a reference structure (if the keyword “reference” is specified). Finally, if the keyword “byresidue” is provided, results are output on a per-residue basis for each snapshot, whereby the number of native contacts is written to *filename.native*.

dihedral *name mask1 mask2 mask3 mask4* [out *filename*] [time *value*] [type *tag-name*]

Calculate the dihedral angle for the four atoms listed in *mask1* through *mask4* (representing rotation about the bond from *mask2* to *mask3*). If more than one atom is listed in each mask, treat the position of that atom as the center of mass of the atoms in the mask. The results are saved internally with the name *name* (which must be unique) and the data is stored on the scalarStack for later processing with the **analyze** command. Data will be dumped to a file named *filename* if "out" is specified. All the angles are specified in degrees, and the file will contain two columns: specifically, the frame number (multiplied by the optional *value* specified with the *time* keyword) in the first column and the angle in the second.

See the command `analyze statistics all` to print averages and standard deviations. Note that there are some special types that can be specified to output summary information for particular angles; at present this is limited to nucleic acid angles (*alpha*, *beta*, *gamma*, *delta*, *epsilon*, *zeta*). Two examples are provided below. This first calculates the dihedral values for residue 1 @CA,C,N and residue 2 @CA with data stored internally referenced by the name "d1". The second calculates the dihedral between the center of mass of residues 1, 2, 3, and 4 respectively, stores the data internally with reference "d2", output to a file named "d2.dat" with the frame number column multiplied by 10.

```
dihedral d1 :1@CA :1@C :1@N :2@CA
dihedral d2 :1 :2 :3 :4 out d2.dat time 10.0
```

diffusion *mask time_per_frame* [*average*] [*filenameroot*]

Compute a mean square displacement plot for the atoms in the *mask*. The time between frames in picoseconds is specified by *time_per_frame*. If "average" is specified, then the average mean square displacement is calculated and dumped (only). If "average" is not specified, then the average and individual mean squared displacements are dumped. They are all dumped to a file in the format appropriate for xmgr (dumped in multicolumn format if necessary, i.e., use `xmgr -nxy`). The units are displacements (in Å²) vs time (in ps). The *filenameroot* is used as the root of the filename to be dumped. The average mean square displacements are dumped to "*filenameroot_r.xmgr*", the x, y and z mean square displacements to "*filenameroot_x.xmgr*", etc and the total distance traveled to "*filenameroot_a.xmgr*".

This will fail if a coordinate moves more than 1/2 the box in a single step. Also, this command implicitly unfolds the trajectory (in periodic boundary simulations) hence will currently only work with orthorhombic unit cells.

Note: this documentation is out of date.

dipole *filename nx x_spacing ny y_spacing nz z_spacing mask1 origin* | *box* [*max_max_percent*]

Same as **grid** (see below) except that dipoles of the solvent molecules are binned. Dumping is to a grid in a format for Chris Bayly's discern delegate program that comes with Midas/Plus. Consult the code in `actions.c`, `transformDipole()` for more information and note that this command is potentially obsolete.

distance *name mask1 mask2* [*out filename*] [*noimage*] [*time interval*]

This command will calculate a distance between the center of mass of the atoms in *mask1* to the center of mass of the atoms in *mask2* and store this information into an array with *name* as the identifier (a name which must be unique and which is placed on the scalarStack for later processing) for each frame in the trajectory. If the optional keyword “out” is specified, then the data is dumped to a file named *filename*. The distance is implicitly imaged (for both orthorhombic and non-orthorhombic unit cells) and the shortest imaged distance will be saved (unless the “noimage” keyword is specified; this is considerably faster, however imaging of the distance will not be performed).

grid *filename nx x_spacing ny y_spacing nz z_spacing mask1* [*origin* | *box*] [*negative*] [*max fraction*]

Create a grid representing the histogram of atoms in *mask1* on the 3D grid that is “*nx* * *x_spacing* by *ny* * *y_spacing* by *nz* * *z_spacing* angstroms (cubed). Either “origin” or “box” can be specified and this states whether the grid is centered on the origin or half box. Note that to provide any meaningful representation of the density, the solute of interest (about which the atomic densities are binned) should be rms fit, centered and imaged prior to the **grid** call. If the optional keyword “negative” is also specified, then these density will be stored as negative numbers. Output is in the format of a XPLOR formatted contour file (which can be visualized by the density delegate to Midas/Plus or Chimera or VMD or other programs). Upon dumping the file, pseudo-pdb HETATM records are also dumped to standard out which have the most probable grid entries (those that are 80% of the maximum by default which can be changed with the max keyword, i.e., max .5 makes the dumping at 50% of the maximum).

Note that as currently implemented, since the XPLOR grids are integer based, the grid is offset from the origin (towards the negative size) by half the grid spacing.

image [*origin*] [*center*] *mask* [*bymol* | *byres* | *byatom* | *bymask*] *mask* [*triclinic* | *familiar* [*com mask*]]

Under periodic boundary conditions, which particular unit cell a given molecule is in does not matter as long as, as a whole, all the molecules “image” into a single unit cell. In an MD simulation, molecules drift over time and may span multiple periodic cells unless “imaging” is enabled to shift molecules that leave back into the primary unit cell. In sander, the IWRAP variable controls this, with IWRAP=1 implying turning on imaging. This command, **image** allows post-processing of the imaging to force all the molecules into the primary unit cell.

If the optional argument “origin” is specified, then imaging will occur to place the box center at the coordinate origin (0.0, 0.0, 0.0) rather than the center of the box (as is the Amber standard). By default all atoms are imaged *by molecule* based on the position of the first atom (or the center of mass of the molecule if “center” is specified; the latter is recommended). If the *mask* is specified, only the atoms in the *mask* will be imaged. It is now possible to image by atom (*byatom*), by residue (*byres*), by molecule (*bymol*, default) or by atom mask (where all the atoms in the *mask* are treated as belonging to a single molecule). The behavior of the “by molecule” imaging is different in CHARMM

and Amber; with Amber the molecules are specified directly by the periodic box information whereas with the CHARMM parameter/topology, each segid is treated as a different molecule. With this newer implementation of the imaging code, it is possible to avoid breaking up double stranded DNA during imaging, i.e.:

```
image :1-20 bymask :1-20
image byres :WAT
```

[Of course this assumes that the coordinates of the two strands were not displaced during the dynamics as well!] Imaging only makes sense if there is periodic box information present.

Non-orthorhombic unit cells are supported. Use of the triclinic imaging can be forced with the “*triclinic*” keyword. Note that this puts the box into the triclinic shape, not the more familiar, more spherical shapes one might expect for some of the unit cells (e.g., the truncated octahedron). To get into the more familiar shape, specify the “*familiar*” keyword. In this case, to specify atoms that imaged molecules should be closest to, specify a center of the atoms in the mask specified with the “*com*” keyword. Note that imaging “*familiar*” is more time consuming (but recommended) since each of the possible imaged distances (27) are checked to see which is closest to the center.

The recommended usage is “*image origin center familiar*”.

principal *mask* [*dorotation*] [*mass*]

Principal axis transformation to align the atoms specified in *mask*. This is reasonably functional as there are still issues with degenerate eigenvalues and unwanted coordinate swapping. To align whole system along the principal axes specify “*dorotation*”.

pucker *name mask1 mask2 mask3 mask4 mask5* [*out filename*] [*amplitude*] [*altona* | *cremer*] [*offset offset*] [*time interval*]

Calculate the pucker for the five atoms specified in each of the mask’s, *mask1* through *mask5*, associating *name* (which must be unique) with the calculated values. If more than one atom is specified in a given mask, the center of mass of all the atoms in that mask is used to define the position. If the “*out*” keyword is specified the data is dumped to *filename*. If the keyword “*amplitude*” is present, the amplitudes are saved rather than the pseudorotation values. If the keyword “*altona*” is listed, use the Altona & Sundarlingam conventions/algorithm (for nucleic acids) (the default) [see Altona & Sundaralingam, *JACS* 94, 8205-8212 (1972) or Harvey & Prabhakaran, *JACS* 108, 6128-6136 (1986).] In this convention, both the pseudorotation phase and amplitude are in degrees.

If “*cremer*” is specified, use the Cremer & Pople conventions/algorithm [see Cremer & Pople, *JACS* 97:6, 1354-1358 (1975).]

Note that to calculate nucleic acid puckers, specify C1’ first, followed by C2’, C3’, C4’ and finally O4’. Also note that the Cremer & Pople convention is offset from the Altona & Sundarlingam convention (with nucleic acids) by 90.0; to add in an extra 90.0 to “*cremer*” (offset -90.0) or subtract 90.0 from the “*altona*” (offset 90.0) specify an *offset* with the *offset* keyword; this value is subtracted from the calculated pseudorotation value (or amplitude).

radial *root-filename spacing maximum solvent-mask [solute-mask] [closest] [density value] [noimage]*

Compute radial distribution functions and store the results into files with *root-filename* as the root of the filename. Three files are currently produced, "*root-filename_carnal.xmgr*" (which corresponds to a carnal style RDF), "*root-filename_standard.xmgr*" (which uses the more traditional RDF with a density input by the user) and "*root-filename_volume.xmgr*" (which uses the more traditional RDF and the average volume of the system). The total number of bins for the histogram is determined by the spacing between bins (*spacing*) and the range which runs from zero to *maximum*. If only a *solvent-mask* is listed (i.e., a list of atoms) then the RDF will be calculated for the interaction of every *solute-mask* atom with ALL the other *solute-mask* atoms.

If the optional *solute-mask* is specified, then the RDF will represent the interaction of each *solute-mask* atom with ALL of the *solvent-mask* atoms. If the optional keyword "closest" is specified, then the histogram will bin, over all the *solvent-mask* atoms, the distance of the closest atom in the *solute-mask*. If the *solute-mask* and *solvent-mask* atoms are not mutually exclusive, zero distances will be binned (although this should not break the code). If the optional keyword "density", followed by the *density value* is specified, this will be used in the calculations. The default value is 0.033456 molecules Å⁻³, which corresponds to a density of water approximately equal to 1.0 g mL⁻¹. To convert a standard density in g mL⁻¹, multiply the density by $\frac{0.6022}{M_r}$, where M_r is the mass of the molecule in atomic mass units. This will only effect the "*root-filename_standard.xmgr*" file.

Note that although imaging of distances is performed (to find the shortest imaged distance unless the "noimage" keyword is specified), minimum image conventions are applied.

Also note that when LES prmtop and trajectories is processed, the interaction between atoms from different copy is ignored, which allows users to get the right RDF, but users may still need to adjust the density to get the right answer.

radgyr [*out filename*] [*time interval*] [*mask*]

Calculate the radius of gyration and the maximal distance of an atom from the center of geometry considering atoms in *mask*. The results are dumped to *filename* if the keyword "out" is specified. Thereby, the time between snapshots is taken to be *interval*.

randomizeions *mask* [*around mask by distance*] [*overlap value*] [*noimage*] [*seed value*]

This can be used to randomly swap the positions of solvent and single atom ions. The "overlap" specifies the minimum distance between ions, and the "around" keyword can be used to specify a solute (or set of atoms) around which the ions can get no closer than the distance specified. The optional keywords "noimage" disable imaging and "seed" update the random number seed. An example usage is

```
randomizeions @Na+ around :1-20 by 5.0 overlap 3.0
```

The above will swap Na⁺ ions with water getting no closer than 5.0 Å from residues 1 – 20 and no closer than 3.0 Å from any other Na⁺ ion.

rms *mode* [*mass*] [*out filename*] [*time interval*] *mask* [*name name*] [*nofit*]

This will RMS fit all the atoms in the *mask* based on the current *mode* which is

previous: fit to previous frame

first: fit to the "start" frame of the first trajectory specified.

reference: fit to a reference structure (which must have been previously read in)

If the keyword "mass" is specified, then a mass-weighted RMSd will be performed. If the keyword "out" is specified (followed immediately by a *filename*), the RMSd values will be dumped to a file. If you want to specify an time interval between frames (used only when dumping the RMSd vs time), this can be done with the "time" keyword. To save the calculated values for later processing, associate a name with the "name" keyword (where the chosen *name must be unique and the data will be stored on the scalarStack* for later processing with the **analyze** command). If the keyword "nofit" is specified, then the coordinates are not modified, just the RMSd values are calculated (and stored or output if the name or out keywords were specified).

secstruct [*out filename*] [*time interval*] [*mask*]

Calculate the secondary structure information for residues of atoms contained in *mask*, following the DSSP method by Kabsch & Sander.[128] The *mask* is primarily intended to strip water molecules etc. Not providing contiguous protein sequences may result in erroneous secondary structure assignments (even at residues that are included in the mask!). The results are dumped to *filename* if the keyword "out" is specified. Thereby, the time between snapshots is taken to be *interval*. For every snapshot and every residue, an alpha-helix is indicated by "H", a 3-10-helix by "G", a pi-helix by "I", a parallel beta-sheet by "b", and an antiparallel beta-sheet by "B". A summary providing the percentage for each residue to adopt one of the above secondary structure types over the course of the analyzed snapshots is given in *filename.sum*.

strip *mask*

Strip all atoms matching the atoms in *mask*. This changes the state of the system such that all commands (actions) following the strip (including output of the coordinates which is done last) are performed on the stripped coordinates (i.e., if you strip all the waters and then on a later action try to do something with the waters, you will have trouble since the waters are gone). Stripping is beneficial, beyond simply paring down a trajectory, for data intensive commands that read entire sections of a trajectory into memory; with stripping to retain only selected atoms, it is much less likely that the available memory will be exceeded.

translate *mask* [*x x-value*] [*y y-value*] [*z z-value*]

Move the coordinates for the atoms in the *mask* in each direction by the offset(s) in Angstroms specified.

truncoc *mask distance* [*prmtop filename*]

Note: This command is largely obsolete.

A. ptraj

Create a truncated octahedron box with solvent stripped to a distance *distance* away from the atoms in the *mask*. Coordinates are output to the trajectory specified with the **trajout** command. *Note that this is a special command* and will only really make sense if a single coordinate set is processed (i.e., any prmtop written out will only correspond to the first configuration!) and commands after the **truncoc** will have undefined behavior since the state will not be consistent with the modified coordinates. It is intended only as an aid for creating truncated octahedron restrt files for running in Amber.

The “prmtop” keyword can be used to specify the writing of a new prmtop (to a file named *filename*; this prmtop is only consistent with the first set of coordinates written. Moreover, this command will only work with Amber prmtop files and assumes an Amber prmtop file has previously been read in (rather than a CHARMM PSF). This command also assumes that all the solvent is located contiguously at the end of the file and that the solvent information has previously been set (see the **solvent** command).

unwrap [*reference*] *mask*

Under periodic boundary conditions, MD trajectories are not continuous if molecules are wrapped(imaged) into the central unit cell. Especially, in sander, with *iwrap*=1, molecular trajectories become discontinuous when a molecule crosses the boundary of the unit cell. This command, **unwrap** processes the trajectories to force the *masked* molecules continuous by translating the molecules into the neighboring unit cells. It is the opposite function of **image**, but this command can also be used to place molecules side by side, for example, two strands of a DNA duplex. However, this command fails when the *masked* molecules travel more than half of the box size within a single frame.

If the optional argument “*reference*” is specified, then the first frame is unwrapped according to the reference structure. Otherwise, the first frame is not modified.

As an example, assume that :1-10 is the first strand of a DNA duplex and :11-20 is the other strand of the duplex. Then the following commands could be used to create system where the two strands are not separated artificially:

```
unwrap :1-20
center :1-20 mass origin
image origin center familiar
```

watershell *mask filename* [*lower lower upper upper*] [*solvent-mask*] [*noimage*]

This option will count the number of waters within a certain distance of the atoms in the *mask* in order to represent the first and second solvation shells. The output is a file into *filename* which has, on each line, the frame number, number of waters in the first shell and number of waters in the second shell. If *lower* is specified, this represents the distance from the *mask* which represents the first solvation shell; if this is absent 3.4 angstroms is assumed. Likewise, *upper* represents the range of the second solvation shell and if absent is assumed to be 5.0 angstroms. The optional *solvent-mask* can be used to consider other atoms as the solvent; the default is “:WAT”. Imaging on the distances is done implicitly unless the “noimage” keyword has been specified.

A.6. Correlation and fluctuation facility

The *ptraj* program now contains several related sets of commands to analyze correlations and fluctuations, both from trajectories and from normal modes. These items replace the *correlation* command in previous versions of *ptraj*, and also replace what used to be done in the *nmanal* program. Some examples of command sequences are given at the end of this section.

vector *name mask* [*principal* [*x|y|z*] | *dipole* | *box* | *corrplane* | *ired mask2* | *corr mask2* | *corried mask2*] \

[*out filename*] [*order order*] [*modes modesfile*] [*beg beg*] [*end end*] [*npair npair*]

This command will keep track of a vector value (and its origin) over the trajectory; the data can be referenced for later use based on the *name* (which must be unique among the vector specifications). "Ired" vectors, however, may only be used in connection with the command "matrix ired". If the optional keyword "out" is specified (not valid for "ired" vectors), the data will be dumped to the file named *filename*. The format is frame number, followed by the value of the vector, the reference point, and the reference point plus the vector. What kind of vector is stored depends on the keyword chosen.

principal: [*x* | *y* | *z*]: store one of the principal axis vectors determined by diagonalization of the inertial matrix from the coordinates of the atoms specified by the *mask*. If none of *x* or *y* or *z* are specified, then the principal axis (i.e., the eigenvector associated with the largest eigenvalue) is stored. The eigenvector with the largest eigenvalue is "x" (i.e., the hardest axis to rotate around) and the eigenvector with the smallest eigenvalue is "z" and if one of *x* or *y* or *z* are specified, that eigenvector will be dumped. The reference point for the vector is the center of mass of the *mask* atoms.

dipole: store the dipole and center of mass of the atoms specified in the *mask*. The vector is not converted to appropriate units, nor is the value well-defined if the atoms in the mask are not overall charge neutral.

box: store the box coordinates of the trajectory. The reference point is the origin or (0.0, 0.0, 0.0).

ired mask2: This defines ired vectors necessary to compute an ired matrix (see matrix command). The vectors must be defined *prior* to the matrix command.

corrplane: This defines a vector perpendicular to the (least-squares best) plane through a series of atoms given in *mask*, for which a time correlation function can be calculated subsequently with the command "**analyze** *timecorr* ...". *order* specifies the order of the Legendre polynomial used (0 ≤ *order* ≤ 2). It defaults to 2.

corr mask2: This defines a vector between the center of mass of *mask* and the one of *mask2*, for which a time correlation function can be calculated subsequently with the command "**analyze** *timecorr* ...". *order* specifies the order of the Legendre polynomial used (0 ≤ *order* ≤ 2). It defaults to 2.

corried mask2: This defines a vector between the center of mass of *mask* and the one of *mask2*, for which a time correlation function according to the Isotropic Reorientational Eigenmode Dynamics (*ired*) approach [244] can be calculated. *order*

A. ptraj

specifies the order of the Legendre polynomial used ($0 \leq \text{order} \leq 2$). It defaults to 2. To calculate this vector, ired modes need to be provided by *modesfile*. They can be calculated by the commands “**matrix ired ...**”, followed by “**analyze matrix ...**”. Only modes <beg> to <end> are considered. Default is beg = 1, end = 50. To obtain meaningful results, it is important that the vector definition agrees with the one used for calculation of the ired matrix (there is no internal check for this). Along these lines, *npair* needs to be specified, which relates to the position of this definition in the sequence of ired (not corried!) vectors used to obtain the ired matrix.

```
matrix dist | covar | mwcovar | distcovar | correl | idea | ired [ name name ] [ order  
order ] [ mask1 ] [ mask2 ]  
[ out filename ] [ start start ] [ stop stop ] [ offset offset ] [ byatom | byres |  
bymask ]
```

Compute distance (*distance*), covariance (*covar*), mass-weighted covariance (*mwcovar*), correlation (*correl*), distance-covariance (*distcovar*), Isotropically Distributed Ensemble Analysis (*idea*),[\[245\]](#) or Isotropic Reorientational Eigenmode Dynamics (*ired*) [\[244\]](#) matrices. Results are output to *filename* if given. Be aware, matrix dimension will be of the order of $N \times M$ for *dist*, *correl*, *idea*, and *ired*, $3N \times 3M$ for *covar* and *mwcovar*, and $N(N-1) \times N(N-1) / 4$ for *distcovar* (with N being the number of atoms in *mask1* and M being the number of atoms either in *mask1* or *mask2*).

“byatom” dumps the results by atom (default). This is the sole option for *covar*, *mwcovar*, *distcovar*, *idea*, and *ired*. In the case of *dist* or *correl*, “byres” calculates an average for each residue and “bymask” dumps the average over all atoms in the mask(s). With “mass”, mass-weighted averages will be computed.

In the case of *ired*, mask information must not be given. Instead, “ired vectors” need to be defined *prior* to the matrix command by using the **vector** command. Otherwise, if no mask is given, all atoms against all are used. If only *mask1* is given, a symmetric matrix is computed. In the case of *distcovar* and *idea*, only *mask1* (or none) may be given. In the case of *distcovar*, *mwcovar*, and *correl*, if *mask1* and *mask2* is given, on output *mask1* atoms are listed column-wise while *mask2* atoms are listed row-wise. The number of atoms covered by *mask1* must be \geq the number of atoms covered by *mask2* (this is also checked in the function).

The matrix may be stored internally on the *matrixStack* with the name *name* for latter processing (with the “**analyze matrix**” command). Since at the moment this only involves diagonalization, storing is only available for (symmetric) matrices generated with *mask1* (or no mask or ired matrices).

The *start*, *stop*, and *offset* parameters can be used to specify the range of coordinates processed (as a subset of all of those read in across all input files).

The *order* parameter chooses the order of the Legendre polynomial used to calculate the ired matrix.

```
analyze matrix matrixname [ out filename ] [ thermo | order ] [ vecs vecs ] [ reduce ] [  
orderparamfile orderparamfilename ]
```

Diagonalizes the matrix *matrixname*, which has been generated and stored before by the **matrix** command. This is followed by Principal Component Analysis (in cartesian coordinate space in the case of a covariance matrix or in distance space in the case of a distance-covariance matrix), or Quasiharmonic Analysis (in the case of a mass-weighted covariance matrix). Diagonalization of distance, correlation, idea, and ired matrices are also possible. Eigenvalues are given in cm^{-1} in the case of a mass-weighted covariance matrix and in the units of the matrix elements in all other cases. In the case of a mass-weighted covariance matrix, the eigenvectors are mass-weighted.

Results [average coordinates (in the case of covar, mwcovar, correl), average distances (in the case of distcovar), main diagonal elements (in the case of idea and ired), eigenvalues, eigenvectors] are output to *filename*. *vecs* determines, how many eigenvectors and eigenvalues are calculated. The value must be ≥ 1 , except if the “thermo” flag is given (see below). In that case, setting *vecs* = 0 results in calculating all eigenvalues, but no eigenvectors. This option is mainly intended for saving memory in the case of thermodynamic calculations. “reduce” (only possible for covar, mwcovar, and distcovar) results in reduced eigenvectors [Abseher & Nilges, *J. Mol. Biol.* **279**, 911, (1998)]. They may be used to compare results from PCA in distance space with those from PCA in cartesian-coordinate space.

“thermo” calculates entropy, heat capacity, and internal energy from the structure of a molecule (average coordinates, see above) and its vibrational frequencies using standard statistical mechanical formulas for an ideal gas. This option is only available for mwcovar matrices.

“order” calculates order parameters based on eigenvalues and eigenvectors with the isotropic reorientational eigenmode dynamics (iRED) approach [Prompers & Brüschweiler, *J. Am. Chem. Soc.* **124**, 4522, (2002)] and outputs them to standard output. This option is only available for ired matrices.

If *orderparamfile* is specified, the ired order parameters will be written to *orderparamfilename* instead of standard output.

```
analyze modes fluct | displ | corr stack stackname | file filename [ beg beg ] [ end end ] \
[ bose ] [ factor factor ] [ out outfile ] [ maskp mask1 mask2 [...] ]
```

Calculates rms fluctuations (“fluct”), displacements of cartesian coordinates along mode directions (“displ”), or dipole-dipole correlation functions (“corr”) for modes obtained from principal component analyses (of covariance matrices) or quasiharmonic analyses (of mass-weighted covariance matrices). Thus, a possible series of commands would be “**matrix** covar | mwcovar ...” to generate the matrix, “**analyze** matrix ...” to calculate the modes, and, finally, “**analyze** modes ...”.

Modes can be taken either from an internal stack, identified by their name on the stack, *stackname*, or can be read from a file *filename*. Only modes *beg* to *end* are considered. Default for *beg* is 7 (which skips the first 6 zero-frequency modes in the case of a normal mode analysis); for *end* it is 50. If “bose” is given, quantum (Bose) statistics is used in populating the modes. By default, classical (Boltzmann) statistics is used. *factor* is

A. ptraj

used as multiplicative constant on the amplitude of displacement. Default is factor = 1. Results are written to *outfile*, if specified, otherwise to stdout. In the case of “corr”, pairs of atom masks (*mask1*, *mask2*; each pair preceded by “maskp” and each mask defining only a single atom) have to be given that specify the atoms for which the correlation functions are desired.

analyze timecorr *vec1 vecname1 vec2 vecname2* [*relax*] [*NHdist nhdistance*] [*freq MHz*] [*tstep tstep*] [*tcorr tcorr*] \
[*drct*] [*dplr*] [*norm*] *out filename* [*noefilename*]

Calculates time correlation functions for vectors *vecname1* (*vecname2*) of type “corr” or “corrred”, using a fast Fourier method. If two different vectors are specified for “*vec1*” and “*vec2*”, a cross-correlation function is calculated; if the two vectors are the same, the result is an autocorrelation function. If the *drct* keyword is given, a direct approach is used instead of the FFT approach. Note that this is less efficient than the FFT route. If *dplr* is given, in addition to the P_l correlation function, also correlation functions $C_l \equiv \langle P_l / (r(0)^3 r(\tau)^3) \rangle$ and $\langle 1 / (r(0)^3 r(\tau)^3) \rangle$ are output. If *norm* is given, all correlation functions are normalized, i.e., $C_l(t=0) = P_l(t=0) = 1.0$. Results are written to *filename*. *tstep* specifies the time between snapshots (default: 1.0), and *tcorr* denotes the maximum time for which the correlations functions are to be computed (default: 10000.0).

relax can only be used when the vectors are of type “corrred” and when they represent an N-H bond. If *relax* is given, correlation times τ_m for each eigenmode and relaxation rates and NOEs for each N-H vector will be calculated following the iRED approach [Prompers & Brüschweiler, *J. Am. Chem. Soc.* **124**, 4522, (2002)]. *NHdist* and *freq* are only considered if *relax* is given. *NHdist* specifies the length of the NH bond in Angstroms (default is 1.02). It is mandatory for the user to set *freq*, which is the Larmor frequency of the measurement. Autocorrelation functions for each mode and the corresponding correlation time τ_m will be written to *filename.cmt*. Autocorrelation functions for each vector will be written to *filename.cjt*. Relaxation rates and NOEs for each N-H vector will be added to the the end of the standard output. For the calculation of τ_m the normalized correlation functions and only the first third of the analyzed time steps will be used. For further information on the convergence of correlation functions see [Schneider, Brünger, Nilges, *J. Mol. Biol.* **285**, 727 (1999)].

If *noefile* is specified, the NOEs and relaxation rates will be written to *noefilename* instead of standard output.

projection *modes modesfile* *out outfile* [*beg beg*] [*end end*] [*mask*] [*start start*] [*stop stop*] [*offset offset*]

Projects snapshots onto modes obtained by diagonalizing covariance or mass-weighted covariance matrices. The modes are read from *modesfile*. The results are written to *outfile*. Only modes *beg* to *end* are considered. Default values are *beg* = 1, *end* = 2. *mask* specifies the atoms that will be projected. The user has to make sure that these atoms agree with the ones used to calculate the modes (i.e., if *mask1* = @CA was used in the “matrix” command, *mask* = @CA needs to be set here as well). The *start*, *stop*, and *offset*

parameters can be used to specify the range of coordinates processed (as a subset of all of those read in across all input files).

A.7. Examples

Please note that in most cases the trajectory needs to be aligned against a reference structure to obtain meaningful results. Use the “**rms**” command for this.

A.7.1. Calculating and analyzing matrices and modes

As a simple example, a distance matrix of all CA atoms is generated and output to `distmat.dat`.

```
matrix dist @CA out distmat.dat
```

In the following, a mass-weighted covariance matrix of all atoms is generated and stored internally with the name `mwcvmat` (as well as output). Subsequently, the matrix is analyzed by performing a quasi-harmonic analysis, whereby 5 eigenvectors and eigenvalues are calculated and output to `evcs.dat`.

```
matrix mwcovar name mwcvmat out mwcvmat.dat
analyze matrix mwcvmat out evcs.dat vecs 5
```

Alternatively, the eigenvectors can be stored internally and used for calculating rms fluctuations or displacements of cartesian coordinates.

```
analyze matrix mwcvmat name evcs vecs 5
analyze modes fluct out rmsfluct.dat stack evcs beg 1 end 3
analyze modes displ out resdispl.dat stack evcs beg 1 end 3
```

Finally, dipole-dipole correlation functions for modes obtained from principle component analysis or quasi-harmonic analysis can be computed.

```
analyze modes corr out cffromvec.dat stack evcs beg 1 end 3 ...
... maskp @1 @2 maskp @3 @4 maskp @5 @6
```

A.7.2. Projecting snapshots onto modes

After calculating modes, snapshots can be projected onto these in an additional “sweep” through the trajectory. Here, snapshots are projected onto modes 1 and 2 read from `evcs.dat` (which have been obtained by the “`matrix mwcovar`”, “`analyze matrix`” commands from above).

```
projection modes evcs.dat out project.dat beg 1 end 2
```

A.7.3. Calculating time correlation functions

Vectors between atoms 5 and 6 as well as 7 and 8 are calculated below, for which auto and cross time correlation functions are obtained.

```
vector v0 @5 corr @6 order 2
vector v1 @7 corr @8 order 2
analyze timecorr vec1 v0 tstep 1.0 tcorr 100.0 out v0.out
analyze timecorr vec1 v1 tstep 1.0 tcorr 100.0 out v1.out
analyze timecorr vec1 v0 vec2 v1 tstep 1.0 tcorr 100.0 out v0_v1.out
```

Similarly, a vector perpendicular to the plane through atoms 18, 19, and 20 is obtained and further analyzed.

```
vector v2 @18,@19,@20 corrplane order 2
analyze timecorr vec1 v3 tstep 1.0 tcorr 100.0 out v2.out
```

For obtaining time correlation functions according to the *ired* approach, two "sweeps" through the trajectory are necessary. First, *ired* vectors are defined and an *ired* matrix is calculated and analyzed. *Ired* eigenvectors are output to *ired.vec*. If the parameter *order* is specified, order parameters based on *ired* are calculated.

```
vector v0 @5 ired @6
vector v1 @7 ired @8
...
vector v5 @15 ired @16
vector v6 @17 ired @18
matrix ired name matired order 2
analyze matrix matired vecs 6 out ired.vec order
```

In a subsequent *ptraj* run, *ired* time correlation functions are calculated by projecting the snapshots onto the *ired* eigenvectors (read from *ired.vec*), which results in *corried* vectors. Then, time correlation functions are computed. By specifying the *relax* parameter, relaxation rates and NOEs can be obtained for each N-H vector. Please note that it is important that the *corried* vector definition agrees with the one used for calculation of the *ired* matrix.

```
vector v0 @5 corried @6 order 2 modes ired.vec beg 1 end 6 npair 1
vector v1 @7 corried @8 order 2 modes ired.vec beg 1 end 6 npair 2
...
vector v5 @15 corried @16 order 2 modes ired.vec beg 1 end 6 npair 6
vector v6 @17 corried @18 order 2 modes ired.vec beg 1 end 6 npair 7
analyze timecorr vec1 v0 relax NHdist 1.02 freq 500.0 tstep 1.0 tcorr 100.0 out v0.out
```

A.8. Hydrogen bonding facility

The *ptraj* program now contains a generic facility for keeping track of lists of pair interactions (subject to a distance and angle cutoff) useful for calculation hydrogen bonding or other

interactions. It is designed to be able to track the interactions between a list of hydrogen bond "donors" and hydrogen bond "acceptors" that the user specifies.

Important Note: In ptraj the definition of donors and acceptors is reversed with respect to standard conventions; the “acceptor” is bonded to the hydrogen and the “donor” is the atom the hydrogen bond is formed to (i.e., in ptraj a “donor” can be thought of as “donating” electrons to the hydrogen atom). This has not been changed in order to preserve backwards-compatibility.

donor *resname atomname* | *mask mask* | *clear* | *print*

This command sets the list of hydrogen bond donors. It can be specified repeatedly to add to the list of potential donors. The usage is either as a pair of residue and atom names or as a specified atom mask. The former usage,

```
donor ADE N7
```

would set all atoms named N7 in residues named ADE to be potential donors.

```
donor mask :10@N7
```

would set the atom named N7 in residue 10 to be a potential donor.

The keyword “clear” will clear the list of donors specified so far and the keyword “print” will print the list of donors set so far.

acceptor *resname atomname atomnameH* | *mask mask maskH* | *clear* | *print*

Similar to the **donor** command, this command sets the list of hydrogen bond acceptors. It can be specified repeatedly to add to the list of potential acceptors. The usage is either as a residue name followed by two atom names (the heavy atom and the hydrogen bonded to the heavy atom respectively), or as two masks, one specifying heavy atoms and one specifying corresponding hydrogen atoms. In either case, the number of heavy atoms must match the number of hydrogen atoms. If the same atom is specified twice (as might be the case to probe ion interactions) then no angle is calculated between the donor and acceptor.

For example:

```
acceptor THY N3 H3
```

would set all atoms named N3 and H3 in residues named THY to be potential acceptors.

```
acceptor mask :11@N3 :11@H3
```

would set the atoms named N3 and H3 in residue 11 to be a potential acceptor.

The keyword “clear” will clear the list of acceptors specified so far and the keyword “print” will print the list of acceptors set so far.

The **donor** and **acceptor** commands do not actually keep track of distances but instead simply set of the list of potential interactions. To actually keep track of the distances, the **hbond** command needs to be specified:

```
hbond [ distance value ] [ angle value ] [ solventneighbor value ] \
      [ solventdonor donor-spec ] [ solventacceptor acceptor-spec ] \
      [ nosort ] [ time value ] [ print value ] [ series name ]
```

A. ptraj

The optional “distance” keyword specifies the cutoff distance for the pair interactions and the optional “angle” keyword specifies the angle cutoff for the hydrogen bond. The default is no angle cutoff and a distance of 3.5 angstroms. To keep track of potential hydrogen bond interactions where we don’t care *which* molecule of a given type is interaction as long as one is (such as with water), the solvent keywords can be specified. An example would be keeping track of water or ions interacting with a particular donor or acceptor. The maximum number of possible interactions per a given donor or acceptor is specified with the “solventneighbor” keyword. The list of potential solvent donors/acceptors is specified with the `solventdonor` and `solventacceptor` keywords (with a format the same as the **donor/acceptor** keywords above).

As an example, if we want to keep track of water interactions with our list of donors/acceptors:

```
hbond distance 3.5 angle 120.0 solventneighbor 6 solventdonor WAT O \
solventacceptor WAT O H1 solventacceptor WAT O H2
```

If you wanted to keep track of interactions with Na⁺ ions (assuming the atom name was Na⁺ and residue name was also Na⁺):

```
hbond distance 3.5 angle 0.0 solventneighbor 6 solventdonor Na+ Na+ \
solventacceptor Na+ Na+ Na+
```

To print out information related to the time series, such as maximum occupancy and lifetimes, specify the “series” keyword.

A.9. rdparm

rdparm requires an Amber *prmtop* file for operation and is menu driven. Rudimentary online help is available with the “?” command. The basic commands are summarized here.

angles *mask*

Print all the angles in the file. If the *mask* is present, only print angles involving these atoms. For example, `angles :CYT@C?` will print all angles involving atoms which have 2-letter names beginning with “C” from “CYT” residues.

atoms *mask*

Print all the atoms in the file. If the *mask* is present, only print these atoms.

bonds *mask*

Print all the bonds in the file. If the *mask* is present, only print bonds involving these atoms.

checkcoords *Amber-trajectory*

Perform a rudimentary check of the coordinates from the filename specified. This is to look for obvious problems (such as overflow) and to count the number of frames.

dihedrals *mask*

Print all the dihedrals in the file. If the *mask* is present, only print dihedrals involving one of these atoms.

delete [*bond* | *angle* | *dihedral*] *number*

This command will delete a given bond, angle or dihedral angle based on the number specified from the current prmtop. The number specified should match that shown by the corresponding **print** command. Note that a new prmtop file is not actually saved. To do this, use the **writeparm** command. For example, “delete bond 5” will delete with 5th bond from the parameter/topology file.

openparm *filename*

Open up the prmtop file specified.

writeparm *filename*

Write a new prmtop file to *filename* based on the current (and perhaps modified) parameter/topology file. Note that this command is obsolete and writes old style prmtop files.

system *string*

Execute the command *string* on the system.

mardi2sander *constraint-file*

A rudimentary conversion of Mardigras style restraints to sander NMR restraint format.

rms *Amber-trajectory*

Create a 2D RMSd plot in postscript or PlotMTV format using the trajectory specified. The user will be prompted for information. This command is rather slow... Use **2Drms** in *ptraj* instead.

stripwater

This command will remove or add three point waters to a prmtop file that already has water. The user will be prompted for information. This is useful to take an existing prmtop and create another with a different amount of water. Of course, corresponding coordinates will also have to be built and this is not done by *rdparm*. To do this, ideally construct a PDB file and convert to Amber coordinate format using *ptraj*. Note that this command is obsolete and writes old style prmtop files.

ptraj *script-file*

This command reads a file or from standard input a series of commands to perform processing of trajectory files. See the *ptraj* documentation.

A. ptraj

translateBox *Amber-coords*

Translate the coordinates (only if they contain periodic box information) specified to place the center either at the origin or at half the box (Amber format). This is obsolete and the user is encouraged to use the **center** command of ptraj instead.

modifyBoxInfo

This is a command to modify the box information, such as to change the box size. The changes are not saved until a **writeparm** command is issued.

modifyMolInfo

This command checks the molecule info (present with periodic box coordinates are specified) and points out problems if they exist. In particular, this is useful to overcome the deficiency in edit which places all the “add” waters into a single molecule.

parmInfo Print out information about the current prmtop file.

printAngles Same as **angles**.

printAtoms Same as **atoms**.

printBonds Same as **bonds**.

printDihedrals Same as **dihedrals**.

printExcluded Print the excluded atom list.

printLennardJones Print out the Lennard-Jones parameters.

printTypes Print out the atom types.

quit Quit the program.

B. Obsolete force field files

The following files are included for historical interest. We do *not* recommend that these be used any more for molecular simulations. The leaprc files that load these files have been moved to `$AMBERHOME/dat/leap/parm/oldff`.

B.1. The Cornell et al. (1994) force field

<code>all_nuc94.in</code>	Nucleic acid input for building database.
<code>all_amino94.in</code>	Amino acid input for building database.
<code>all_aminoc94.in</code>	COO- amino acid input for database.
<code>all_aminont94.in</code>	NH3+ amino acid input for database.
<code>nacl.in</code>	Ion file.
<code>parm94.dat</code>	1994 force field file.
<code>parm96.dat</code>	Modified version of 1994 force field, for proteins.
<code>parm98.dat</code>	Modified version of 1994 force field, for nucleic acids.

Contained in **ff94** are parameters from the so-called “second generation” force field developed in the Kollman group in the early 1990s.[57] These parameters are especially derived for solvated systems, and when used with an appropriate 1-4 electrostatic scale factor, have been shown to perform well at modeling many organic molecules. The parameters in *parm94.dat* omit the hydrogen bonding terms of earlier force fields. This is an all-atom force field; no united-atom counterpart is provided. 1-4 electrostatic interactions are scaled by 1.2 instead of the value of 2.0 that had been used in earlier force fields.

Charges were derived using Hartree-Fock theory with the 6-31G* basis set, because this exaggerates the dipole moment of most residues by 10-20%. It thus “builds in” the amount of polarization which would be expected in aqueous solution. This is necessary for carrying out condensed phase simulations with an effective two-body force field which does not include explicit polarization. The charge-fitting procedure is described in Ref [57].

The **ff96** force field [246] differs from *parm94.dat* in that the torsions for ϕ and ψ have been modified in response to *ab initio* calculations [247] which showed that the energy difference between conformations were quite different than calculated by Cornell *et al.* (using *parm94.dat*). To create *parm96.dat*, common V1 and V2 parameters were used for ϕ and ψ , which were empirically adjusted to reproduce the energy difference between extended and constrained alpha helical energies for the alanine tetrapeptide. This led to a significant improvement between molecular mechanical and quantum mechanical relative energies for the remaining members of the set of tetrapeptides studied by Beachy *et al.* Users should be aware that *parm96.dat* has

B. Obsolete force field files

not been as extensively used as *parm94.dat*, and that it almost certainly has its own biases and idiosyncrasies, including strong bias favoring extended β conformations.[10, 248, 249]

The **ff98** force field [250] differs from *parm94.dat* in torsion angle parameters involving the glycosidic torsion in nucleic acids. These serve to improve the predicted helical repeat and sugar pucker profiles.

B.2. The Weiner et al. (1984,1986) force fields

<code>all.in</code>	All atom database input.
<code>allct.in</code>	All atom database input, COO- Amino acids.
<code>allnt.in</code>	All atom database input, NH3+ Amino acids.
<code>uni.in</code>	United atom database input.
<code>unict.in</code>	United atom database input, COO- Amino acids.
<code>unint.in</code>	United atom database input, NH3+ Amino acids.
<code>parm91X.dat</code>	Parameters for 1984, 1986 force fields.

The **ff86** parameters are described in early papers from the Kollman and Case groups.[251, 252] [The “parm91” designation is somewhat unfortunate: this file is really only a corrected version of the parameters described in the 1984 and 1986 papers listed above.] These parameters are not generally recommended any more, but may still be useful for vacuum simulations of nucleic acids and proteins using a distance-dependent dielectric, or for comparisons to earlier work. The material in *parm91X.dat* is the parameter set distributed with Amber 4.0. The *STUB* nonbonded set has been copied from *parmuni.dat*; these sets of parameters are appropriate for united atom calculations using the “larger” carbon radii referred to in the “note added in proof” of the 1984 JACS paper. If these values are used for a united atom calculation, the parameter *scnb* must be defined in the *prmtop* file and should be set to 8.0; for all-atom calculations it should be 2.0. The *scee* parameter should be defined in the *prmtop* file and set to 2.0 for both united atom and all-atom variants. *Note that the default value for scee is now 1.2 (the value for 1994 and later force fields); this must be explicitly defined in the prmtop file when using the earlier force fields.*

parm91X.dat is not recommended. However, for historical completeness a number of terms in the non-bonded list of *parm91X.dat* should be noted. The non-bonded terms for I (iodine), CU (copper) and MG (magnesium) have not been carefully calibrated, but are given as approximate values. In the *STUB* set of non-bonded parameters, we have included parameters for a large hydrated monovalent cation (IP) that represent work by Singh *et al.*[253] on large hydrated counterions for DNA. Similar values are included for a hydrated anion (IM).

The non-bonded potentials for hydrogen-bond pairs in *ff86* use a Lennard-Jones 10-12 potential. If you want to run *sander* with *ff86* then you will need to recompile, adding -DHAS_10_12 to the Fortran preprocessor flags.

B.3. The Wang et al. (1999) force field

<code>parm99.dat</code>	Basic force field parameters
<code>all_amino94.in</code>	topologies and charges for amino acids
<code>all_amino94nt.in</code>	same, for N-terminal amino acids

<code>all_amino94ct.in</code>	same, for C-terminal amino acids
<code>all_nuc94.in</code>	topologies and charges for nucleic acids
<code>gaff.dat</code>	Force field for general organic molecules
<code>all_modrna08.lib</code>	topologies for modified nucleosides
<code>all_modrna08.frcmod</code>	parameters for modified nucleosides

The **ff99** force field [254] points toward a common force field for proteins for “general” organic and bio-organic systems. The atom types are mostly those of Cornell *et al.* (see below), but changes have been made in many torsional parameters. The topology and coordinate files for the small molecule test cases used in the development of this force field are in the *parm99_lib* subdirectory. The *ff99* force field uses these parameters, along with the topologies and charges from the Cornell *et al.* force field, to create an all-atom nonpolarizable force field for proteins and nucleic acids.

There are more than 99 naturally occurring modifications in RNA. Amber force field parameters for all these modifications have been developed to be consistent with *ff94* and *ff99*. [255] The modular nature of RNA was taken into consideration in computing the atom-centered partial charges for these modified nucleosides, based on the charging model for the “normal” nucleotides. [256] All the *ab initio* calculations were done at the Hartree-Fock level of theory with 6-31G(d) basis sets, using the GAUSSIAN suite of programs. The computed electrostatic potential (ESP) was fit using RESP charge fitting in *antechamber*. Three-letter codes for all of the fitted nucleosides were developed to standardize the naming of the modified nucleosides in PDB files. For a detailed description of charge fitting for these nucleosides and an outline for the three letter codes, please refer to Ref. [255].

The AMBER force field parameters for 99 modified nucleosides are distributed in the form of library files. The *all_modrna08.lib* file contains coordinates, connectivity, and charges, and *all_modrna08.frcmod* contains information about bond lengths, angles, dihedrals and others. The AMBER force field parameters for the 99 modified nucleosides in RNA are also maintained at the modified RNA database at <http://ozone3.chem.wayne.edu>.