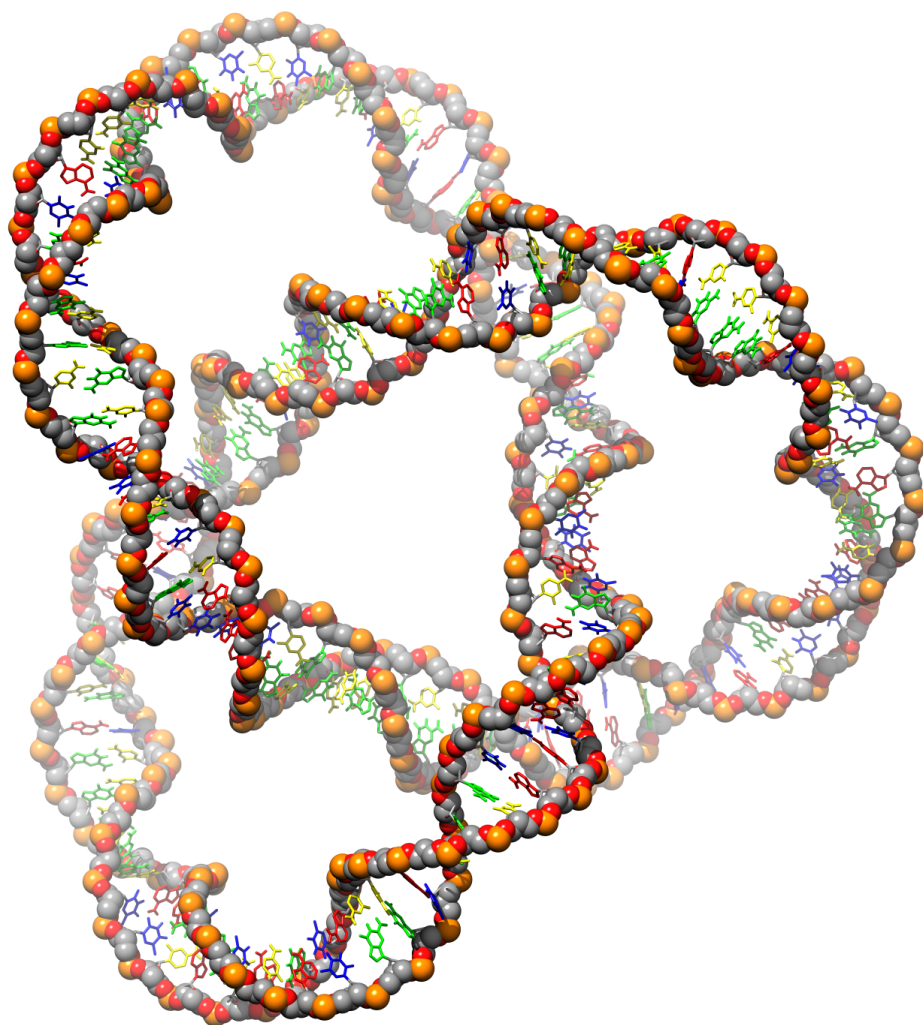


# AmberTools Users' Manual





# AmberTools Users' Manual

Version 1.4, April 25, 2010

*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,<sup>1</sup> W.A. Svrcek-Seiler,<sup>2</sup>  
Russell A. Brown,<sup>3</sup> István Kolossváry,<sup>4</sup>  
Yannick J. Bomble<sup>13</sup> and David A. Case<sup>5</sup>

## **LEaP and gleap**

Wei Zhang,<sup>6</sup> Tingjun Hou,<sup>7</sup> Christian  
Schafmeister,<sup>8</sup> Wilson S. Ross, and David A.  
Case

## **Antechamber**

Junmei Wang<sup>10</sup>

## **Ptraj**

Thomas E. Cheatham, III,<sup>11</sup> *et al.* (see  
<http://ambermd.org/contributors.html>)

## **PBSA**

Jun Wang, Qin Cai, Xiang Ye, Meng-Juei  
Hsieh, Chuck Tan, and Ray Luo<sup>12</sup>

## **Sqm**

Ross C. Walker<sup>15</sup>, Michael F. Crowley<sup>13</sup>,  
Scott Brozell, Tim Giese<sup>17</sup>, Andreas W.  
Goetz<sup>15</sup>, and David A. Case

## **CHAMBER**

Michael F. Crowley, Mark Williamson<sup>15</sup>,  
Ross C. Walker

## **3D-RISM**

Tyler Luchko<sup>5</sup>, David A. Case, Sergey  
Gusarov<sup>16</sup>, Andriy Kovalenko<sup>16</sup>

---

<sup>1</sup>Kosmix Corporation; <sup>2</sup>University of Vienna ; <sup>3</sup>Sun Microsystems, Inc. ; <sup>4</sup>Budapest University of Technology and Economics and D.E. Shaw Research, LLC; <sup>5</sup>Rutgers University; <sup>6</sup> Univ. of Texas, Health Center at Houston; <sup>7</sup> Univ. of California, San Diego; <sup>8</sup>University of Pittsburgh; <sup>10</sup> Univ. of Texas, Southwestern Medical Center; <sup>11</sup> University of Utah; <sup>12</sup>University of California, Irvine, <sup>13</sup>NREL, <sup>14</sup>SUNY Stony Brook, <sup>15</sup>San Diego Supercomputer Center, <sup>16</sup>National Institute for Nanotechnology, University of Alberta, <sup>17</sup>University of Minnesota

## 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](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

<b>Contents</b>	<b>3</b>
<b>1 Getting started</b>	<b>11</b>
1.1 Information flow in Amber	11
1.1.1 Preparatory programs	12
1.1.2 Simulation programs	12
1.1.3 Analysis programs	13
1.2 Installation	13
1.3 Contacting the developers	15
<b>2 Specifying a force field</b>	<b>17</b>
2.1 Specifying which force field you want in LEaP	18
2.2 The AMOEBA potentials	19
2.3 The Duan et al. (2003) force field	19
2.4 The Yang et al. (2003) united-atom force field	20
2.5 1999 force fields and recent updates	20
2.6 The 2002 polarizable force fields	22
2.7 Force related to semiempirical QM	23
2.8 GLYCAM-06 and GLYCAM-04EP force fields for carbohydrates and lipids	23
2.8.1 List of relevant files shipped with AMBER 11.	23
2.8.2 File versioning and obtaining the latest parameters.	24
2.8.3 General information regarding parameter development	24
2.8.4 Scaling of electrostatic and nonbonded interactions	24
2.8.5 Development of partial atomic charges	25
2.8.6 Carbohydrate parameters for use with the TIP5P water model	25
2.8.7 Carbohydrate Naming Convention in GLYCAM	25
2.9 Ions	29
2.10 Solvent models	30
2.11 Obsolete force field files	31
2.11.1 The Cornell et al. (1994) force field	31
2.11.2 The Weiner et al. (1984,1986) force fields	32
2.12 CHAMBER	33
2.12.1 Usage	37
2.12.2 Validation	38
2.12.3 Known limitations / Issues	38
2.12.4 Acknowledgments	39

## CONTENTS

<b>3</b>	<b>LEaP</b>	<b>41</b>
3.1	Introduction	41
3.2	Concepts	41
3.2.1	Commands	41
3.2.2	Variables	42
3.2.3	Objects	42
3.3	Basic instructions for using LEaP	46
3.3.1	Building a Molecule For Molecular Mechanics	47
3.3.2	Amino Acid Residues	47
3.3.3	Nucleic Acid Residues	48
3.4	Commands	48
3.4.1	add	48
3.4.2	addAtomTypes	49
3.4.3	addIons	50
3.4.4	addIons2	50
3.4.5	addPath	50
3.4.6	addPdbAtomMap	50
3.4.7	addPdbResMap	51
3.4.8	alias	52
3.4.9	bond	52
3.4.10	bondByDistance	52
3.4.11	check	52
3.4.12	combine	53
3.4.13	copy	53
3.4.14	createAtom	54
3.4.15	createResidue	54
3.4.16	createUnit	54
3.4.17	deleteBond	54
3.4.18	desc	54
3.4.19	groupSelectedAtoms	55
3.4.20	help	56
3.4.21	impose	56
3.4.22	list	57
3.4.23	loadAmberParams	57
3.4.24	loadAmberPrep	57
3.4.25	loadOff	57
3.4.26	loadMol2	58
3.4.27	loadPdb	58
3.4.28	loadPdbUsingSeq	58
3.4.29	logFile	58
3.4.30	measureGeom	59
3.4.31	quit	59
3.4.32	remove	59
3.4.33	saveAmberParm	60
3.4.34	saveMol2	60

3.4.35	saveOff	60
3.4.36	savePdb	60
3.4.37	sequence	61
3.4.38	set	61
3.4.39	solvateBox and solvateOct	63
3.4.40	solvateCap	63
3.4.41	solvateShell	64
3.4.42	source	64
3.4.43	transform	64
3.4.44	translate	65
3.4.45	verbosity	65
3.4.46	zMatrix	65
3.5	Building oligosaccharides and lipids	66
3.5.1	Procedures for building oligosaccharides using the GLYCAM 06 parameters	67
3.5.2	Procedures for building a lipid using GLYCAM 06 parameters	70
3.5.3	Procedures for building a glycoprotein in LEaP	71
3.6	Differences between <i>tleap</i> and <i>sleap</i>	73
3.6.1	Limitations	73
3.6.2	Unsupported Commands	73
3.6.3	New Commands or New Features of old Commands	73
3.6.4	New keywords	74
3.6.5	The basic idea behind the new commands	75
<b>4</b>	<b>Antechamber</b>	<b>77</b>
4.1	Principal programs	78
4.1.1	antechamber	78
4.1.2	parmchk	80
4.2	A simple example for antechamber	81
4.3	Programs called by antechamber	84
4.3.1	atomtype	84
4.3.2	am1bcc	85
4.3.3	bondtype	86
4.3.4	prepgen	87
4.3.5	espgen	87
4.3.6	respgen	88
4.4	Miscellaneous programs	89
4.4.1	acdofactor	89
4.4.2	database	90
4.4.3	parmcal	90
4.4.4	residuegen	90
4.4.5	top2ff	91
4.4.6	top2mol2	92
4.4.7	translate	92

## CONTENTS

4.5	New Development of Antechamber And GAFF	93
4.5.1	Extensive Test of AM1-BCC Charges	94
4.5.2	New Development of GAFF	94
<b>5</b>	<b>Semiempirical quantum chemistry</b>	<b>97</b>
5.1	Introduction	97
5.2	General &qmmm Namelist Variables	98
<b>6</b>	<b>ptraj</b>	<b>103</b>
6.1	Understanding ptraj commands and actions	104
6.2	ptraj coordinate input/output commands	105
6.3	ptraj commands that override the molecular information specified	107
6.4	ptraj <i>action</i> commands - short list	108
6.5	ptraj <i>action</i> commands - detailed discussion	108
6.6	Correlation and fluctuation facility	122
6.7	Parallel ptraj	126
6.8	Examples	127
6.8.1	Calculating and analyzing matrices and modes	128
6.8.2	Projecting snapshots onto modes	128
6.8.3	Calculating time correlation functions	128
6.9	Hydrogen bonding facility	129
6.10	rdparm	131
6.11	AMBER Trajectory NetCDF Format	133
6.11.1	Program behavior	133
6.11.2	NetCDF file encoding	133
6.11.3	Global attributes	134
6.11.4	Dimensions	135
6.11.5	Label variables	135
6.11.6	Data variables	136
6.11.7	Example	137
<b>7</b>	<b>PBSA</b>	<b>139</b>
7.1	Introduction	139
7.1.1	Numerical solutions of the PB equation	140
7.1.2	Numerical interpretation of energy and forces	141
7.1.3	Numerical accuracy and related issues	142
7.2	Usage and keywords	143
7.2.1	File usage	143
7.2.2	Basic input options	143
7.2.3	Options to define the physical constants	145
7.2.4	Options to select numerical procedures	146
7.2.5	Options to compute energy and forces	147
7.2.6	Options for visualization and output	149
7.2.7	Options to select a non-polar solvation treatment	149

7.3	Example inputs	151
7.3.1	Single-point calculation of solvation free energies	151
7.3.2	Visualization of electrostatic potentials	152
7.3.3	Single point calculation of forces	153
7.3.4	Comparing with Delphi results	153
7.4	PBSA in SANDER	154
7.5	PBSA in NAB	154
7.5.1	Electrostatic Forces/Gradients in PBSA	155
7.5.2	Example	155
<b>8</b>	<b>Reference Interaction Site Model of Molecular Solvation</b>	<b>157</b>
8.1	Introduction	157
8.1.1	1D-RISM	159
8.1.2	3D-RISM	160
8.2	Practical Considerations	161
8.2.1	Computational Requirements and Parallel Scaling	161
8.2.2	Output	162
8.2.3	Numerical Accuracy	162
8.3	Work Flow	162
8.4	rism1d	163
8.4.1	Parameters keywords	163
8.4.2	Example	165
8.5	3D-RISM in NAB	166
8.5.1	Solvation Box Size	166
8.5.2	I/O	166
8.5.3	Examples	166
8.5.4	Compiling MPI 3D-RISM	167
8.6	File Formats	167
8.6.1	MDL	167
8.6.2	XVV	169
8.6.3	OpenDX	170
<b>9</b>	<b>Miscellaneous utilities</b>	<b>173</b>
9.1	ambpdb	173
9.2	reduce	175
9.2.1	Running reduce	175
9.2.2	General input flags	176
9.2.3	Fixing an orientation	177
9.2.4	Cliques	178
9.2.5	Contact	178
9.3	elsize	178

## CONTENTS

<b>10 NAB: Introduction</b>	<b>181</b>
10.1 Background	182
10.1.1 Conformation build-up procedures	183
10.1.2 Base-first strategies	183
10.2 Methods for structure creation	184
10.2.1 Rigid-body transformations	184
10.2.2 Distance geometry	185
10.2.3 Molecular mechanics	186
10.3 Compiling nab Programs	187
10.4 Parallel Execution	187
10.5 First Examples	188
10.5.1 B-form DNA duplex	188
10.5.2 Superimpose two molecules	189
10.5.3 Place residues in a standard orientation	190
10.6 Molecules, Residues and Atoms	191
10.7 Creating Molecules	192
10.8 Residues and Residue Libraries	193
10.9 Atom Names and Atom Expressions	195
10.10 Looping over atoms in molecules	197
10.11 Points, Transformations and Frames	198
10.11.1 Points and Vectors	198
10.11.2 Matrices and Transformations	198
10.11.3 Frames	199
10.12 Creating Watson Crick duplexes	200
10.12.1 bdna() and fd_helix()	201
10.12.2 wc_complement()	201
10.12.3 wc_helix() Overview	202
10.12.4 wc_basepair()	203
10.12.5 wc_helix() Implementation	206
10.13 Structure Quality and Energetics	210
10.13.1 Creating a Parallel DNA Triplex	210
10.13.2 Creating Base Triads	211
10.13.3 Finding the lowest energy triad	213
10.13.4 Assembling the Triads into Dimers	215
<b>11 NAB: Language Reference</b>	<b>221</b>
11.1 Introduction	221
11.2 Language Elements	221
11.2.1 Identifiers	221
11.2.2 Reserved Words	221
11.2.3 Literals	222
11.2.4 Operators	222
11.2.5 Special Characters	223
11.3 Higher-level constructs	223
11.3.1 Variables	223

11.3.2	Attributes	224
11.3.3	Arrays	226
11.3.4	Expressions	227
11.3.5	Regular expressions	228
11.3.6	Atom Expressions	228
11.3.7	Format Expressions	229
11.4	Statements	231
11.4.1	Expression Statement	231
11.4.2	Delete Statement	232
11.4.3	If Statement	232
11.4.4	While Statement	232
11.4.5	For Statement	233
11.4.6	Break Statement	234
11.4.7	Continue Statement	234
11.4.8	Return Statement	234
11.4.9	Compound Statement	234
11.5	Structures	234
11.6	Functions	236
11.6.1	Function Definitions	236
11.6.2	Function Declarations	237
11.7	Points and Vectors	237
11.8	String Functions	238
11.9	Math Functions	238
11.10	System Functions	240
11.11	I/O Functions	240
11.11.1	Ordinary I/O Functions	240
11.11.2	matrix I/O	242
11.12	Molecule Creation Functions	242
11.13	Creating Biopolymers	243
11.14	Fiber Diffraction Duplexes in NAB	244
11.15	Reduced Representation DNA Modeling Functions	245
11.16	Molecule I/O Functions	246
11.17	Other Molecular Functions	247
11.18	Debugging Functions	249
11.19	Time and date routines	250
<b>12</b>	<b>NAB: Rigid-Body Transformations</b>	<b>251</b>
12.1	Transformation Matrix Functions	251
12.2	Frame Functions	251
12.3	Functions for working with Atomic Coordinates	252
12.4	Symmetry Functions	252
12.4.1	Matrix Creation Functions	253
12.4.2	Matrix I/O Functions	254
12.5	Symmetry server programs	255
12.5.1	matgen	255

## CONTENTS

12.5.2	Symmetry Definition Files	255
12.5.3	matmerge	257
12.5.4	matmul	258
12.5.5	matextract	258
12.5.6	transform	258
<b>13</b>	<b>NAB: Distance Geometry</b>	<b>259</b>
13.1	Metric Matrix Distance Geometry	259
13.2	Creating and manipulating bounds, embedding structures	260
13.3	Distance geometry templates	265
13.4	Bounds databases	268
<b>14</b>	<b>NAB: Molecular mechanics and dynamics</b>	<b>271</b>
14.1	Basic molecular mechanics routines	271
14.2	Typical calling sequences	280
14.3	Second derivatives and normal modes	281
14.4	Low-MODe (LMOD) optimization methods	284
14.4.1	LMOD conformational searching	284
14.4.2	LMOD Procedure	285
14.4.3	XMIN	286
14.4.4	Sample XMIN program	288
14.4.5	LMOD	291
14.4.6	Sample LMOD program	294
14.4.7	Tricks of the trade of running LMOD searches	298
<b>15</b>	<b>NAB: Sample programs</b>	<b>301</b>
15.1	Duplex Creation Functions	301
15.2	nab and Distance Geometry	302
15.2.1	Refine DNA Backbone Geometry	303
15.2.2	RNA Pseudoknots	306
15.2.3	NMR refinement for a protein	309
15.3	Building Larger Structures	313
15.3.1	Closed Circular DNA	313
15.3.2	Nucleosome Model	317
15.4	Wrapping DNA Around a Path	320
15.4.1	Interpolating the Curve	320
15.4.2	Driver Code	324
15.4.3	Wrap DNA	325
15.5	Other examples	328
	<b>Bibliography</b>	<b>329</b>
	<b>Bibliography</b>	<b>329</b>
	<b>Index</b>	<b>343</b>

# 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.

The programs here are mostly released under the GNU General Public License (GPL). A few components are included that are in the public domain or which 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.

## 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* 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 Xray crystallography, NMR spectroscopy, or model-building. They should be in Protein Databank (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.
2. "Topology": connectivity, atom names, atom types, residue names, and charges. This information comes from the database, which is found in the *amber11/dat/leap/parm* 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 *amber11/dat/leap/parm* directory; consult 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.

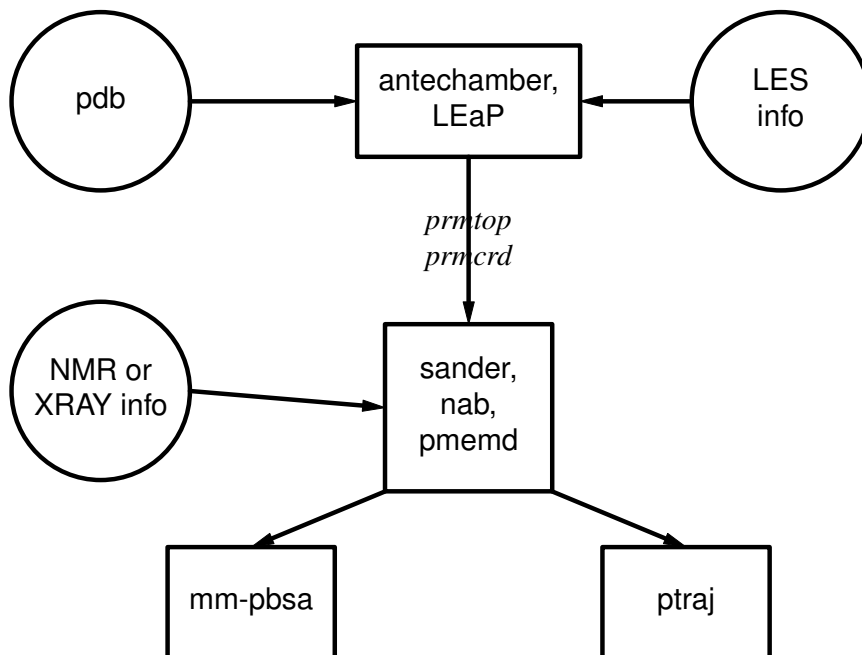


Figure 1.1: Basic information flow in Amber

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 old systems. It combines the functionality of prep, link, edit, and parm from earlier versions. The program **sleap** is an updated version of this, with some additional functionality.

**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.

### 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*.

### 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 the program evolved), can examine and modify *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.

**mm-pbsa** (part of Amber) is a script that automates energy analysis of snapshots from a molecular dynamics simulation using ideas generated from continuum solvent models. There are two versions of this, an older perl script (called *mm\_pbsa*) and a new python script (called *mmpbsa\_py*). See the Amber11 Users' Manual for more information.

## 1.2 Installation

1. First, extract the files in some location (we use `/usr/local` as an example here):

```
cd /usr/local
tar xvfj AmberTools-1.4.tar.bz2
```

After this, check for bugfixes at <http://ambermd.org/bugfixesat.html>; follow the instructions there to install any needed updates. **Note:** if you have an *amber11* directory that contains a copy of AmberTools version 1.3, do **not** unpack AmberTools 1.4 in the same place!

2. Next, set your AMBERHOME environment variable:

## 1 Getting started

```
export AMBERHOME=/usr/local/amber11      # (for bash, zsh, ksh...)
setenv AMBERHOME /usr/local/amber11     # (for csh, tcsh)
```

Be sure to change the “/usr/local” above to whatever directory is appropriate for your machine. You should also add \$AMBERHOME/bin to your PATH.

3. Now, in the *src* directory, you should run the configure script:

```
cd amber11/AmberTools/src
./configure --help
```

will show you the options. Choose a 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.)

4. Then,

```
make
```

will compile the codes. If the make fails, it is possible that some of the entries in "config.h" are not correct.

5. This can be followed by

```
cd ../test
make test
```

which will run tests and will report successes or failures.

6. If you wish to prepare parallel (MPI) versions of NAB and ptraj, do this:

```
cd $AMBERHOME/AmberTools/src
make clean
./configure -mpi <...other options...> <compiler-choice>
make parallel
cd ../test
make test.parallel
```

This assumes that you have installed MPI, and that *mpicc* is in your PATH.

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

- a) The parallel version of *ptraj* can speed up some analysis, but requires special hardware and software to support parallel i/o; see Section 6.7 for more information. If you don't have a high performance file environment, you may get better performance from the regular version. If you want parallel *ptraj* only, type `cd ptraj; make parallel`.

- b) Currently, the MPI version of *nab* will over-write the serial version; that is: only the most recently-compiled version of *nab* will be used. Be sure you are familiar with the serial version, and that you really need the parallel version before compiling it. If you have shared-memory nodes, the openmp version might be a better alternative. See Section 10.4 for more information.

7. NAB can also be compiled using openmp:

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

See Section 10.4 for information on running the openmp version of NAB. There is no current provision for more than one version of NAB; the last one compiled will be the one that is used.

## 1.3 Contacting the developers

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



## 2 Specifying a force field

Amber is designed to work with several simple types of force fields, although it is most commonly used with parameterizations developed by Peter Kollman and his co-workers. There are now a variety of such parameterizations, with no obvious "default" value. The "traditional" parameterization uses fixed partial charges, centered on atoms. Examples of this are *ff94*, *ff99* and *ff03* (described below). The default in versions 5 and 6 of Amber was *ff94*; a comparable default now would probably be *ff03* or *ff99SB*, but users should consult the papers listed below to see a detailed discussion of the changes made.

Less extensively used, but very promising, recent 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 parameterization), and the latter adds in off-center charges (or "extra points"), primarily to help describe better the angular dependence of hydrogen bonds. Again, users should consult the papers cited below to see details of how these new force fields have been developed.

(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 2.12, 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 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 "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 have a ".in" extension. Standard libraries of residue descriptions are in the *amber11/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 *amber11/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 needed) a smaller *frmod* file that keeps track of any changes to the default parameters that are needed. The *frmod* files for changing the default water model (which is TIP3P) into other water

## 2 Specifying a force field

models are in files like *amber11/dat/leap/parm/frcmod.tip4p*. The *parmchk* program (part of *antechamber*) can also generate *frcmod* files.

### 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:

<i>filename</i>	<i>topology</i>	<i>parameters</i>
leaprc.ff99SB	"	parm99.dat+frcmod.ff99SB
leaprc.ff99bsc0	"	parm99.dat+frcmod.ff99SB+frcmod.parmbsc0
leaprc.ff03.r1	Duan et al. 2003	parm99.dat+frcmod.ff03
leaprc.ff03ua	Yang et al. 2003	parm99.dat+frcmod.ff03+frcmod.ff03ua
leaprc.ff02	reduced charges	parm99.dat+frcmod.ff02pol.r1
leaprc.gaff	none	gaff.dat
leaprc.GLYCAM_06	Woods et al.	GLYCAM_06g.dat
leaprc.GLYCAM_04EP	"	GLYCAM_04EP.dat
leaprc.amoeba	Ren & Ponder	Ren & Ponder

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.ff03.r1 leaprc
```

or

```
cd $AMBERHOME/dat/leap/cmd
ln -s leaprc.ff99SB 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 *ff03* or *ff99SB* force field. A *leaprc* file in the current 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* (or *saveAmberParmPert*) to save the *prmtop* file, and keep the default *ipol=0* in *sander* or *gibbs*.

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. The files above assume that nucleic acids are DNA, if not explicitly specified. Use the files *leaprc.rna.ff98*, *leaprc.rna.ff99*, *leaprc.rna.ff02* or *leaprc.rna.ff02EP* to make the default RNA. If you have a mixture of DNA and RNA, you will need to edit your PDB file, or use the `loadPdbUsingSequence` command in LEaP (see that chapter) in order to specify which nucleotide is which.
5. There is also a *leaprc.gaff* file, which sets you up for the "general" Amber force field. This is primarily for use with Antechamber (see that chapter), and does not load any topology files.
6. 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.
7. Our experience with generalized Born simulations is mainly with *ff99* or *ff03*; the current GB models are not compatible with polarizable force fields. Replacing explicit water with a GB model is equivalent to specifying a different force field, and users should be aware that none of the GB options (in Amber or elsewhere) is as "mature" as simulations with explicit solvent; user discretion is advised! For example, it was shown that salt bridges are too strong in some of these models [2, 3] and some of them provide secondary structure distributions that differ significantly from those obtained using the same protein parameters in explicit solvent, with GB having too much  $\alpha$ -helix present.[4, 5]

## 2.2 The AMOEBA potentials

The **amoeba** force field for proteins, ions, organic solvents and water, developed by Ponder and Ren [6–10], is available in *sander*. This force field is specified by setting *iamoeba* to 1 in the sander input file. Setting up the system is described in Section 3.6. Basically, you follow the usual procedure, loading *leaprc.amoeba* at the beginning, and using `saveAmoebaParm` (rather than the usual `saveAmberParm`) at the end.

## 2.3 The Duan et al. (2003) force field

<code>frcmod.ff03</code>	For proteins: changes to <code>parm99.dat</code> , primarily in the <code>phi</code> and <code>psi</code> 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

## 2 Specifying a force field

The **ff03** force field [11, 12] 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  $\phi$  and  $\psi$  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*.

### 2.4 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_aminooct03.in</code>	COO- amino acid input for building database.

The **ff03ua** force field [13] 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 sidechains 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 all-atom protein backbone, the  $\phi$  and  $\psi$  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*.

### 2.5 1999 force fields and recent updates

<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>frcmod.ff99SB</code>	"Stony Brook" modification to ff99 backbone torsions
<code>frcmod.ff99SP</code>	"Sorin/Pande" modification to ff99 backbone torsions
<code>frcmod.parmbsc0</code>	"Barcelona" changes to ff99 for nucleic acids
<code>all_modrna08.lib</code>	topologies and charges for modified nucleotides

**all\_modrna08.frcmod parameters for modified nucleotides**

The **ff99** force field [14] points toward a common force field for proteins for "general" organic and bioorganic 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.

**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.[15] 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,  $\beta$ ,  $\alpha_L$ , and  $\alpha_R$ ). A detailed explanation of the parametrization as well as an extensive comparison with many other variants of fixed charge Amber forcefields 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  $\beta$ -turn structures, two things which were especially problematic with most previous Amber force field variants. In order to use *ff99SB*, issue "source leaprc.ff99SB" at the start of your LEaP session.

An alternative is to simply zero out the torsional terms for the  $\phi$  and  $\psi$  backbone angles.[16] Another alteration along the same lines has been developed by Sorin and Pande,[17] and is implemented in the *frcmod.ff99SP* file. Research in this area is ongoing, and users interested in peptide and protein folding are urged to keep abreast of the current literature.

**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  $\alpha$  and  $\gamma$  backbone torsion angles.[18] These updated parameters are in the *frcmod.parmbsc0* file, and are the ones we now recommend. The *leaprc.ff99bsc0* file loads these, along with the *ff99SB* protein parameters.

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*. [19] The modular nature of RNA is taken into consideration in computing the atom-centered partial charges for these modified nucleosides, based on the charging model for the "normal" nucleotides.[20] All the ab initio calculations are done at the Hartree-Fock level of theory with 6-31G(d) basis sets, using GAUSSIAN suite of programs. The computed electrostatic potential (ESP) is fit using RESP charge fitting with the Antechamber module of AMBER. 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. [19].

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.

## 2 Specifying a force field

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>.

**General organic molecules.** The General Amber Force Field (**gaff**) is discussed in Chap. 4.

### 2.6 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 <sup>-</sup> amino acid input ...
<code>all_aminonc02.in</code>	NH3 <sup>+</sup> 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 <sup>-</sup> amino acid input ...
<code>all_aminonc02EP.in</code>	NH3 <sup>+</sup> amino acid input ...

The **ff02** force field is a polarizable variant of *ff99*. (See Ref. [21] 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 intramolecular self polarization has been included.[22] 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.[23] These are available in the *frcmod.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.[24] 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.[22, 25] 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.

## 2.7 Force related to semiempirical QM

**ParmAM1** and **parmPM3** are classical force field parameter sets that reproduce the geometry of proteins minimized at the semiempirical AM1 or PM3 level, respectively.[26] These new force fields provide an inexpensive, yet reliable, method to arrive at geometries that are more consistent with a semiempirical 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 semiempirical level can become cost-prohibitive, a "preminimization" with an appropriately parameterized classical treatment will facilitate future analysis using AM1 or PM3 Hamiltonians.

## 2.8 GLYCAM-06 and GLYCAM-04EP force fields for carbohydrates and lipids

### 2.8.1 List of relevant files shipped with AMBER 11.

#### GLYCAM 2006 force field

<code>GLYCAM_06g.dat</code>	Parameters for oligosaccharides (Check <a href="http://www.glycam.org">www.glycam.org</a> for more recent versions)
<code>GLYCAM_06.prep</code>	Structures and charges for glycosyl residues
<code>GLYCAM_06_lipids.prep</code>	Structures and charges for sample lipid residues (Check <a href="http://www.glycam.org">www.glycam.org</a> for additional residues)
<code>leaprc.GLYCAM_06</code>	LEaP configuration file for GLYCAM 06
<code>GLYCAM_amino_06.lib</code>	Glycoprotein library for centrally-positioned residues
<code>GLYCAM_aminoc_06.lib</code>	Glycoprotein library for C-terminal residues
<code>GLYCAM_aminont_06.lib</code>	Glycoprotein library for N-terminal residues

#### GLYCAM 2004EP force field using lone pairs (extra points)

<code>GLYCAM_04EP.dat</code>	Parameters for oligosaccharides
<code>GLYCAM_04EP.prep</code>	Structures and charges for glycosyl residues
<code>leaprc.GLYCAM_04EP</code>	LEaP configuration file for GLYCAM 04EP

#### GLYCAM Force Field Parameters Download Page

## 2 Specifying a force field

[http://www.glycam.org/documents/gl\\_params.jsp](http://www.glycam.org/documents/gl_params.jsp)

Unlike in previous releases of the GLYCAM force field, individual prep files will no longer be released with Amber. Instead, one file contains prep entries for all carbohydrate residues. Another file contains prep entries for lipid residues. For linking glycans to proteins, the libraries containing amino acid residues that have been modified for the purpose must be loaded. At present, it is possible to link to serine, threonine, hydroxyproline and asparagine.

### 2.8.2 File versioning and obtaining the latest parameters.

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 Download Page. Researchers are strongly encouraged to obtain the most recent files for their projects. Online file names might vary slightly from those listed above. In general, prep files, library files and leaprc files are not versioned, though they do change slightly from time to time, primarily to correct typographical errors. The atomic charges, found in the prep and library files, rarely change. Main parameter files are versioned by appending a letter to the year designation. The first version is "a," and the current version is "g". 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.

### 2.8.3 General information regarding parameter development

In GLYCAM06,[27] 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 GLCYAM06 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). Thus, GLYCAM and AMBER may be combined for modeling carbohydrate-protein complexes and glycoproteins. Because the GLYCAM06 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  $\alpha$ - and  $\beta$ -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 GLYCAM06 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.

### 2.8.4 Scaling of electrostatic and nonbonded interactions

As in previous versions of GLYCAM,[28] the parameters were derived for use without scaling 1-4 non-bonded and electrostatic interactions, e.g., SCNB and SCEE should typically be

## 2.8 GLYCAM-06 and GLYCAM-04EP force fields for carbohydrates and lipids

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.[29] 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. For further information on the format of the parameter files and the required LEAP commands, please see "write14scale" in Section 3.6.4

### 2.8.5 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,[27, 30] 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.[31] 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 conformational ensembles.[27, 32] 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.[27, 32]

### 2.8.6 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, GLYCAM04EP, has been derived in which lone pairs (or extra points, EPs) have been incorporated on the oxygen atoms.[33] 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 sp<sup>3</sup> oxygen atom was found to be 0.70 Å; for an sp<sup>2</sup> 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.[33]

### 2.8.7 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 database (pdb) files by the RCSB PDB Advisory Committee ([www.rcsb.org/pdb/pdbac.html](http://www.rcsb.org/pdb/pdbac.html)),

## 2 Specifying a force field

Carbohydrate	Pyranose	Furanose
	$\alpha/\beta$ , D/L	$\alpha/\beta$ , 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 2.1: *Current Status of Monosaccharide Availability in GLYCAM.* (a) *Currently under development.* (b) *Only one enantiomer and ring form known.*

## 2.8 GLYCAM-06 and GLYCAM-04EP force fields for carbohydrates and lipids

	Carbohydrate <sup>a</sup>	One letter code <sup>b</sup>	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 <sup>d</sup>	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 <sup>d</sup>	GalA
21	D-Glucuronic Acid	Z <sup>d</sup>	GlcA
22	D-Iduronic Acid	U <sup>d</sup>	IdoA
23	D-N-Acetylgalactosamine	V <sup>d</sup>	GalNAc
24	D-N-Acetylglucosamine	Y <sup>d</sup>	GlcNAc
25	D-N-Acetylmannosamine	W <sup>d</sup>	ManNAc
26	N-Acetyl-neuraminic Acid	S <sup>d</sup>	NeuNAc, Neu5Ac
	KDN	KN <sup>c,d</sup>	KDN
	KDO	KO <sup>c,d</sup>	KDO
	N-Glycolyl-neuraminic Acid	SG <sup>c,d</sup>	NeuNGc, Neu5Gc

Table 2.2: *The one-letter codes that form the core of the GLYCAM residue names for monosaccharides* <sup>a</sup>*Users requiring prep files for residues not currently available may contact the Woods group ([www.glycam.org](http://www.glycam.org)) to request generation of structures and ensemble averaged charges.* <sup>b</sup>*Lowercase letters indicate L-sugars, thus L-Fucose would be “f”, see Table 2.5 .* <sup>c</sup>*Less common residues that cannot be assigned a single letter code are accommodated at the expense of some information content.* <sup>d</sup>*Nomenclature involving these residues will likely change in future releases.[34] Please visit [www.glycam.org](http://www.glycam.org) for the most updated information.*

## 2 Specifying a force field

	$\alpha$ -D-Glcp	$\beta$ -D-Galp	$\alpha$ -D-Arap	$\beta$ -D-Xylp
Linkage Position	Residue Name	Residue Name	Residue Name	Residue Name
Terminal <sup>b</sup>	0GA <sup>b</sup>	0LB	0AA	0XB
1- <sup>c</sup>	1GA <sup>c</sup>	1LB	1AA	1XB
2-	2GA	2LB	2AA	2XB
3-	3GA	3LB	3AA	3XB
4-	4GA	4LB	4AA	4XB
6-	6GA	6LB		
2,3-	ZGA <sup>d</sup>	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 2.3: Specification of linkage position and anomeric configuration in D-hexo- and D-pentopyranoses in three-letter codes based on the GLYCAM one-letter code <sup>a</sup>In pyranoses A signifies  $\alpha$ -configuration; B =  $\beta$ . <sup>b</sup>Previously 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. <sup>c</sup>Introduced to facilitate the formation of a 1-1' linkage as in  $\alpha$ -D-Glc-1-1'- $\alpha$ -D-Glc {1GA 0GA}. <sup>d</sup>For 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.

	$\alpha$ -D-Glcf	$\beta$ -D-Manf	$\alpha$ -D-Araf	$\beta$ -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 2.4: 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  $\alpha$ ; U (up) =  $\beta$ .

	$\alpha$ -L-Glcp	$\beta$ -L-Manp	$\alpha$ -L-Arap	$\beta$ -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 2.5: Specification of linkage position and anomeric configuration in L-hexo- and L-pentofuranoses in three-letter codes.

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 2.2).[34] 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 2.3 to 2.5), the present GLYCAM residue names encode the following content: carbohydrate residue name (Glc, Gal, etc.), ring form (pyranosyl or furanosyl), anomeric configuration ( $\alpha$  or  $\beta$ ), 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 Ions

```

frcmod.ionsjc_tip3p      Jung/Cheatham ion parameters for TIP3P water
frcmod.ionsjc_spce      same, but for SPC/E water
frcmod.ionsjc_tip4pew   same, but for TIP4P/EW water
ions08.lib              topologies for ions with the new naming scheme
ions94.lib              topologies for ions with the old naming scheme

```

In the past, for alkali ions with TIP3P waters, Amber has provided the values of Aqvist,[35] 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

## 2 Specifying a force field

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.[36] 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.[37] The naming scheme for ions in the older Amber force fields is also not very straightforward.

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.[38] These have been separately parameterized 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 *frmod* file.

### 2.10 Solvent models

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

Amber now provides direct support for several water models. The default water model is TIP3P.[39] 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 frmod.pol3 (sets the HW,OW parameters to POL3)
```

(The above is obviously for the POL3 model.) The *solvents.lib* file contains TIP3P,[39] TIP3P/F,[40] TIP4P,[39, 41] TIP4P/Ew,[42, 43] TIP5P,[44] POL3[45] and SPC/E[46] 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[47] (called “SPF”) and one for path-integral MD, qSPC/Fw[48] (called “SPG”). You would use these in the following manner:

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

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, SPCFBOX, TIP3PBOX, TIP3PFBOX, TIP4PBOX, and TIP4PEWBOX. 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, along with a box for an 8M urea-water mixture. The input files for a single molecule are in *amber11/dat/leap/prep*, and the corresponding frcmod files are in *amber11/dat/leap/parm*. Pre-equilibrated boxes are in *amber11/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.

## 2.11 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*.

### 2.11.1 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.

## 2 Specifying a force field

<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 1990's.[49] 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 [49].

The **ff96** force field [50] differs from *parm94.dat* in that the torsions for  $\phi$  and  $\psi$  have been modified in response to *ab initio* calculations [51] 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  $\phi$  and  $\psi$ , 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 not been as extensively used as *parm94.dat*, and that it almost certainly has its own biases and idiosyncrasies, including strong bias favoring extended  $\beta$  conformations.[15, 52, 53]

The **ff98** force field [54] 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.

### 2.11.2 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.[55, 56] [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.*[57] 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.

## 2.12 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 [58]. There are two components to CHAMBER:

1. The tool (*\$AMBERHOME/exe/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[49] and CHARMM[59, 60] are two approaches to the parameterization 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.

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

## 2 Specifying a force field

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

In the case of the CHARMM force field, its MDE is also called CHARMM[61, 62]. 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 within CHARMM, but 1.2 in Amber and the van der Waal's 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 Waal's 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 and the final additional term is a cross term, named CMAP, [63, 64], which is a function of two sequential protein back bone dihedrals. This term originates from differences observed between classically calculated two dimensional  $\phi/\psi$  peptide free energy surfaces using the CHARMM22 force field and that of experiment. CMAP is a numerical energy correction which essentially transforms the 2D  $\phi/\psi$  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 ( $\theta_0$ , Eq.2.2) parameter in degrees in its parameter file and AMBER stores this 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 CHARGE and VDW sections for the same reasons. Specifically the modified sections of the prmtop format. and new additions 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.*

```
%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/rad**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)
```

## 2 Specifying a force field

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

```
%FLAG CHARMM_IMPROPER
%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.

*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.*

## 2.12.1 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)
-radius_set (GB radius set) options are: <default>
    0 bondi radii (bondi)
    1 amber6 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 Specifying a force field

### 2.12.2 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, 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 (**frcdump**) has been implemented which provides a validation route for alternate implementations of the CHARMM force fields. The command, **frcdump**, for a given system, 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 validation 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:

```
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 it is recommended that the user initially carries out such a comparison prior to running a long production run.

### 2.12.3 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 in tree tests. 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.

### 2.12.4 Acknowledgments

The authors acknowledge financial support for this work under DoE SciDAC grant DE-AC36-99G0-10337. RCW acknowledges additional financial support under UC Lab award 09-LR-06-117792 and the San Diego Supercomputer Center Advanced User Support program and NSF grant TG-MCB090110 for providing supercomputer resources in support of this work.



## 3 LEaP

### 3.1 Introduction

LEaP is a module from the AMBER suite of programs, which can be used to generate force field files compatible with NAB. 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.

### 3.2 Concepts

In order to effectively use LEaP it is necessary to understand the philosophy behind the program, especially of 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.

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:

### 3 LEaP

```
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, STRINGS, or LISTS 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, pound sign, equal sign, space, semicolon, double quote, or list open or close characters { and }. LEaP commands should not be used as variable names. Variable names are case-sensitive: "ARG" and "arg" are different variables. Variables are associated with objects using an assignment statement not unlike regular computer 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 (6.02E23). 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 and 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 objects NUMBERS and STRINGS to the complex objects UNITS, RESIDUES, 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 and they are identical to double precision variables in Fortran and double in C.

### STRINGs

STRINGs are simple objects that are identical to character arrays in C and similar to character strings in Fortran. STRINGs are represented by 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 nonbond parameters for AMBER force field calculations. They are normally loaded from e.g. parm94.dat and frcmod files.

### ATOMs

ATOMs are complex objects that do not contain any other objects. The ATOM object is similar to the chemical concept of atoms. Thus, it is a single entity that may be bonded to other ATOMs and it may be 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.

### 3 LEaP

**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 "parm.dat" or "frcmod" file. For smooth operation, all atom types need to have element and hybridization defined by the addAtomTypes command. The standard AMBER force field atom types are added by the default "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 an ATOM that is used in making links to other RESIDUES. In UNITS containing single RESIDUES, the RESIDUES' connect0 ATOM is usually defined as the UNITS' 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 an ATOM that is used in making links to other RESIDUES. In UNITS containing single RESIDUES, the RESIDUES' connect1 ATOM is usually defined as the UNITS' 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 is an ATOM property which defines an ATOM that can be used in making links to other RESIDUES. In amino acids, the convention is that this is the ATOM to which disulphide 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. UNITs, when paired with one or more PARMSETs, contain all of the information required to perform a calculation using AMBER. 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 then the bounding box will be defined to be a cube with each side being NUMBER of angstroms across. If the value is a LIST then it must be a LIST containing three numbers, the lengths of the three sides of the bounding box.

**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 then no solvent cap is defined. If the value is a LIST then it must contain four numbers, the first three define the Cartesian coordinates (X, Y, Z) of the origin of the solvent cap in angstroms, the fourth NUMBER defines the radius of the solvent cap in angstroms.

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 angstroms. The third example defines a solvent cap centered at ( 15.0, 10.0, 5.0 ) angstroms with a radius of 8.0 . Note: the "set cap" command does

### 3 LEaP

not actually solvate, it just sets an attribute. See the `solvateCap` command for a more practical case.

UNITS are complex objects that can contain RESIDUES and ATOMS. UNITS can be created using the `createUnit` command and modified using the set commands. The contents of a UNIT can be modified using the add and remove commands.

#### 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 translates to "some RESIDUE named ALA within the UNIT named dipeptide". The second form translates as "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, the notation to use is as follows:

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

Assuming that the ATOM with the name CA has a sequence number 3, then both of the above commands will print a description of the  $\alpha$ -carbon of RESIDUE `dipeptide.ALA` 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 zmatrix. 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 as easy as it might be. 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 mis-read 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 for examples of this.
3. The saveAmberParm command cited above is appropriate for calculations that do not compute free energies; for the latter you will need to use saveAmberParmPert. For polarizable force fields, you will need to add Pol to the above commands (see the Commands section, below.)

### 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:

### 3 LEaP

```
{ 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 delta or epsilon 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 HID. Thus, if a PDB file is read that contains the residue HIS, the residue will be assigned to the HID 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" or "R" prefix can be used to distinguish between deoxyribose and ribose units; with the default `leaprc` file, ambiguous residues are assumed to be deoxy. Residue names like "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 capped by hydrogens are end in "N," such as "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 brackets ([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 tape.

```
> 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 {
> { 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

### 3 LEaP

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 cannot 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 said molecule, and the latter is deleted. (To avoid this behavior, either solvate `_after_addions`, or use `addIons2`.) Ions must be monoatomic. 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 angstrom, extending from an inner radius of ( `maxIonVdwRadius + maxSoluteAtomVdwRadius` ) to an outer radius 4 angstroms 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 start-up file, "leaprc", so that assignments are made at the

beginning of the session. The LIST is a LIST of LISTS. Each sublist contains 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 start-up 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 string string }
```

where double can be 0 or 1, the first string is the name within the PDB file, and the second string is the variable name to which the first string 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 tape:

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

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 this string is removed from the Alias Table. If no arguments are given with the command, the current aliases stored in the Alias Table will be listed.

The proposed alias is first checked for conflict with the LEP commands and it is 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 container that are within maxBond angstroms of each other. If maxBond is not specified then 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 the 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:

- o long bonds
- o short bonds
- o non-integral total charge of the UNIT.
- o missing force field atom types
- o close contacts (< 1.5 ) 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>

```

### 3.4.13 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

```

### 3 LEaP

then both tripeptide and tripeptideSol would be solvated since they would both point 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. 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-
             poskwn+ 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.

```

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 the 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

```

### 3 LEaP

An expression like "TRP@sideChain" returns a LIST, so any commands that require LIST 's 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 the STRING is not given then 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 the UNIT. The list of RESIDUES to impose the internal coordinates upon is in seqlist. The internal coordinates to impose are in the LIST internals.

The command works by looking into each RESIDUE within the UNIT that is listed in the seqlist argument 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. Ranges of sequence numbers are 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 the UNIT. For example, the range { 1 999 } represents 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 alpha 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 angstroms between the alpha and beta carbons. RESIDUES without an ATOM named CB (like glycine) will be unaffected.

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 are applied to each RESIDUE.

### 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 tape:

```
> 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 is provided it 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 LEaP

#### 3.4.26 loadMol2

```
variable = loadMol2 filename
```

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

#### 3.4.27 loadPdb

```
variable = loadPdb filename
```

Load a Protein Databank format file with the file name filename. 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 Protein Data Bank format file from the 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 b is not contained by a then 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 some variable name then 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
```

Save the AMBER/NAB topology and coordinate files for the 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 the UNIT. This command 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 that are in the 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 doesn't have any 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 in the argument of this command. If the file filename already exists then the new objects will be added to the file. 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 create a new UNIT by combining the contents of a LIST of UNITS. The first argument is a LIST of UNITS. A new UNIT is constructed by taking each UNIT in the sequence 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 Amber6 and before; 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", and 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. [65], which may be used with *igb*=2, 5 or 7; *mbondi*, which is the default, and the recommended parameter set for *igb*=1 [66]; *mbondi2*, which is a second modification of the Bondi radii set [67], 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 [68].

### 3 LEaP

For ATOMs:

**name** A unique STRING descriptor used to identify ATOMs.

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

**pertType** The 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 defines an ATOM that is used in making links to other RESIDUEs. In UNITs containing single RESIDUEs, the RESIDUEsS connect0 ATOM is usually defined as the UNIT's head ATOM.

**connect1** This is an ATOM property which defines an ATOM that is used in making links to other RESIDUEs. In UNITs containing single RESIDUEs, the RESIDUEsS connect1 ATOM is usually defined as the UNIT's tail ATOM.

**connect2** This is an ATOM property which defines an ATOM that can be used in making links to other RESIDUEs. In amino acids, the convention is that this is the ATOM to which disulphide 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** 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 then the bounding box will be defined to be a cube with each side being NUMBER of angstroms across. If the value is a LIST then it must be a LIST containing three numbers, the lengths of the three sides of the bounding box.
- cap** The property defines the solvent cap of the UNIT. If it is defined as null then no solvent cap is defined. If the value is a LIST then it must contain four numbers, the first three define the Cartesian coordinates (X, Y, Z) of the origin of the solvent cap in angstroms, the fourth NUMBER defines the radius of the solvent cap in angstroms.

### 3.4.39 solvateBox and solvateOct

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

The *solvateBox* command creates a periodic solvent rectangular box around the solute UNIT. The shape for *solvateOct* is a truncated octahedron. The solute UNIT is modified by the addition of solvent RESIDUES, such that the closest distance between any atom of the solute and the edge of the periodic box is given by the *distance* parameter. The solvent box will be repeated in all three spatial directions.

The optional closeness parameter can be used to control how close, in angstroms, solvent ATOMS can come to solute ATOMS. The default value of the closeness argument 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 to the closest solute ATOM is less than the sum of the ATOMS VANDERWAAL's distances multiplied by the closeness argument.

```
> 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 the solute UNIT. The solute UNIT is modified by the addition of solvent RESIDUES. The solvent box will be repeated in all three spatial directions to create a large solvent sphere with a radius of radius angstroms.

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

The optional closeness parameter can be used to control how close, in angstroms, solvent ATOMS can come to solute ATOMS. The default value of the closeness argument is 1.0. Smaller values allow solvent ATOMS to come closer to solute ATOMS. The criterion for rejection of

### 3 LEaP

overlapping solvent RESIDUES is if the distance between any solvent ATOM to the closest solute ATOM is less than the sum of the ATOMs VANDERWAAL's distances multiplied by the closeness argument.

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

```
> 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 the solute UNIT. The resulting solute/solvent UNIT will be irregular in shape since it will reflect the contours of the solute. The solute UNIT is modified by the addition of solvent RESIDUES. 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 angstroms thick. The solvent RESIDUES are then added to the solute UNIT if they lie within the shell defined by thickness and do not overlap with the solute ATOMS. 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 commands within a text file. 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 the ( 3 x 3 ) or ( 4 x 4 ) matrix represented by the nine or sixteen NUMBERS in the LIST of LISTs matrix. The general matrix looks like:

```
r11 r12 r13 -tx r21 r22 r23 -ty r31 r32 r33 -tz 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 six angstrom 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 defined by the three NUMBERS in the LIST direction.

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 by placing it bond12 angstroms along the x-axis from ATOM a2. If ATOM a2 does not have coordinates defined then ATOM a2 is placed at the origin.

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

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

### 3 LEaP

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

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

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

This entry defines the coordinate of a1 by placing it bond12 angstroms away from ATOM a2, making angles angle123 between ATOMs a1, a2, and a3, and angle124 between ATOMs a1, a2, and a4. The argument orientation defines whether the ATOM a1 is above or below a plane defined by the ATOMs a2, a3, and a4. If orientation is positive then a1 will be placed in such a way so that the inner product of (a3-a2) cross (a4-a2) with (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, a4 ) are either ATOMs or STRINGS containing names of ATOMs within object. The subsequent arguments are all NUMBERS. Any ATOM can be placed at the a1 position, even those that have 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

Before continuing in this section, you should review the GLYCAM naming conventions covered in Section 2.8. 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 they are “lacking” either a hydroxyl group or a hydroxyl proton, and may be lacking more than one proton depending on the number of branching locations. The result of this is that none of the residues is a complete molecule unto itself. For example, if you wish to build  $\alpha$ -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

### 3.5 Building oligosaccharides and lipids

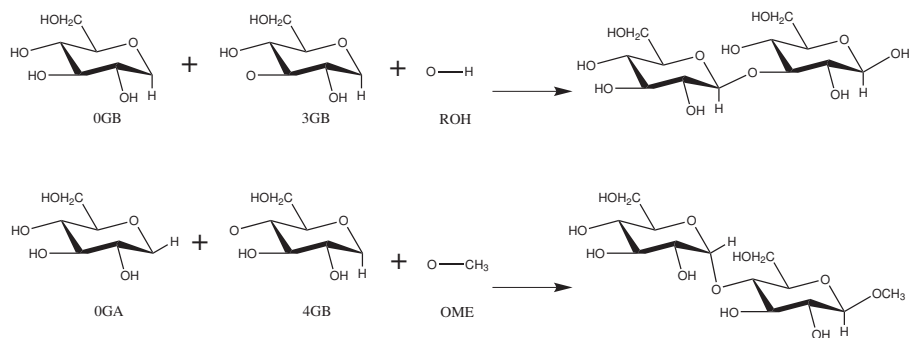


Figure 3.1: Schematic representation of disaccharide formation, indicating the need for open valences on carbon and oxygen atoms at linkage positions.

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.ccr.c.uga.edu> or <http://www.glycam.org>).

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.:

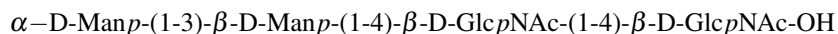
```
sleap -f leap_input_file
```

**Note that it is, as of the time of this writing, necessary to build your topology files using sleap because of the required electrostatic and nonbonded interaction scaling. See Section 2.8.4 for further discussion on the requirement. See "write14scale" in Section 3.6.4 for discussion of its implementation in sleap.** Also, note that any GLYCAM parameter set shipped with AMBER is likely to be updated in the future. The current version is GLYCAM\_06g.dat. This file and GLYCAM\_06.prep are automatically loaded with the default leaprc.GLYCAM\_06. The user is encouraged to check [www.glycam.org](http://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:





Note that the  $\beta$ -D-mannopyranose is now branched at the number three and six positions. Consulting Tables 2.2 to 2.5 informs us that the first character assigned to a carbohydrate linked at the three and six positions is V. So, 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 three and six 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_06 # 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 number six 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 6} { {H1 C1 O6 C6 -60.0} } # set phi torsion and
impose branch {4 6} { {C1 O6 C6 H6 0.0} } # set psi OMA(6) & VMB
impose branch {4 3} { {H1 C1 O4 C4 60.0} } # set phi torsion and
impose branch {4 3} { {C1 O4 C4 H4 0.0} } # set psi 3MB & 4YB
impose branch {3 2} { {H1 C1 O4 C4 60.0} } # set phi torsion and
impose branch {3 2} { {C1 O4 C4 H4 0.0} } # set psi 4YB & 4YB
impose branch {5 4} { {H1 C1 O3 C3 -60.0} } # set phi torsion and
impose branch {5 4} { {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 LEaP

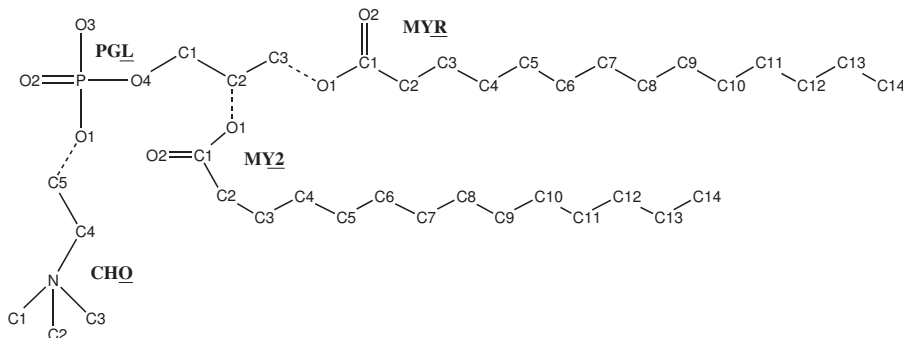


Figure 3.2: *DMPC*

### 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 phosph-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 3.2 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 the antechamber module. GLYCAM compatible charges were added to the prep files and a prep file database (GLYCAM\_06\_lipids.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\_06. 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_06 # source the leaprc for GLYCAM 06
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 3} { {C1 C2 C3 O1 163} } # set torsions for
impose lipid {2 3} { {C2 C3 O1 C1 -180} } # PGL & MYR
impose lipid {2 3} { {C3 O1 C1 C2 180} }
impose lipid {2 4} { {O4 C1 C2 O1 -60} } # set torsions for
impose lipid {2 4} { {C1 C2 O1 C1 -180} } # PGL & MY2
impose lipid {2 4} { {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 disulphide bonds (SSBOND) by looking at the header section of the pdb file. 3RN3 has four pairs of disulphide bonds between the following cysteine residues: 26 – 84, 40 – 95, 58 – 110, and 65 – 72. Change the names of these 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\_06 is sourced. See the lists of library files in 2.8 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 with that present in the experimental pdb file. 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 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 (for example 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.

```
source leaprc.GLYCAM_06 # load the GLYCAM 06 leaprc
source leaprc.ff99SB # load the (modified) ff99 force field
glyprot = loadpdb 3rn3_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 disulphide 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
```

## 3.6 Differences between *tleap* and *sleap*

The *sleap* program is a new text-based tool that is almost entirely compatible with *tleap*, and at some point in the future we will retire *tleap*. Below, we discuss the differences between the two codes. Please note that *sleap* is a new code, and has not been tested nearly as much as *tleap* has. We encourage people to use it – that’s the only way it will get better! – but be on the lookout for places where it might not do what it should. The “gleap” and “mort” foundations, on which *sleap* is built, will be the basis for a lot of new functionality in the future.

### 3.6.1 Limitations

For now, *sleap* has the following limitations:

**SaveAmberParm** won’t give the identical topology file as *tleap* does, while the energy should be identical.

**SolvateDontClip** has not been implemented.

**addions** won’t give the identical result as of *tleap* does due to the different set of vdw radii they are using.

### 3.6.2 Unsupported Commands

The following commands are not going to be implemented, since it is not clear to me why do they even exist.

**addAtomTypes:** It seems to me the only usage of it is designating the hybrid type of an atom, which is determined by chemical environment in *sleap*.

**logFile:** All the information are dumped to standard output now.

### 3.6.3 New Commands or New Features of old Commands

The following new commands have been introduced into *sleap*:

**loadsdf** allows users to read mdl’s sdf format file. The syntax is

### 3 LEaP

```
unitname = loadsdf filename
```

**savesdf** allows users to save mdl sdf format files. The syntax is

```
savesdf unitname filename
```

**loadmol2** can now load molecules that have more than one residue.

**savemol2** allows users to save tripos mol2 format files. The syntax is

```
savemol2 unitname filename
```

**fixbond** assigns bond orders automatically. Note that the input molecule should have only one residue. There is a test case showing how to use fixbond in *amber11/test/sleap/fastbld*. The syntax is

```
fixbond unitname
```

**addhydr** adds hydrogens to a molecule. The molecule should have only one residue and have correct bond order assigned. There is a test case showing how to use addhydr in *amber11/test/sleap/fastbld*. The syntax is

```
addhydr unitname
```

**setpchg** calls *antechamber* to set partial charges (AM1-BCC) and gaff atom types for a molecule. The molecule should have only one residue. There is a test case showing how to use setpchg in *amber11/test/sleap/fastbld*. The syntax is

```
setpchg unitname
```

**saveamoebaparm** save a topology file for the AMOEBA force field. The syntax is

```
saveamoebaparm unitname xxx.top xxx.xyz
```

There is a test case showing how to use saveamoebaparm in *amber11/test/sleap/amoeba*. The only difference is that the user should load *leaprc.amoeba* at startup which loads the AMOEBA force field parameters and AMOEBA specialized libraries.

**parmchk** calls parmchk on a molecule to get missing force field parameters and add them to the database. The syntax is

```
parmchk unitname
```

#### 3.6.4 New keywords

The following new keywords have been introduced into *sleap*:

**echo**: if set to "on", the input command will be echoed. This is very useful for the construction of test cases.

**disulfide** is used to control the behavior of loadpdb on disulfide bonds. if disulfide is set to "off", loadpdb will not create disulfide bonds unless they are specified in the CONECT records; if disulfide is set to "auto", loadpdb will create disulfide bonds between two sulfur atoms whose distance is less then the value specified by keyword "disulfcut" (by default the cutoff is 2.2 angstrom); if disulfide is set to "manu", loadpdb will ask the user if they want to create a disulfide bond when such a pair of sulfur atoms is found; by default it is set to off.

**disulfcut** is used as the cutoff of disulfide bonds.

**fastbld** is used to control the behavior of loadpdb for unknown residues. if fastbld is set to "on" and an unknown residue is encountered in the pdb file, loadpdb will try to run fixbond, addhydr, setpchg and parmchk on the unknown residue and put all the the necessary information together into the molecule. The resulting molecule will then be ready for SaveAmberParm.

**write14scale** instructs sleap to output SCEE and SCNB sections to any topology files it generates when set to "on" (default: "off"). For this directive to change values from the default, the parameter file must contain the new values. New values are specified in the comment field of each relevant line in the torsions section using keyword/value pairs *SCEE=value1* and *SCNB=value2* . These values must be present for each line of multi-term torsions. The exact position within the comment field is irrelevant, but there cannot be any whitespace before or after the equals sign (=). The information written into the topology file overrides both internal defaults for SCEE and SCNB as well as any specification made in the simulation input file. The following are sample parameter-file torsion entries that specify SCEE and SCNB. In this example, whitespace has been compressed for formatting purposes.

```
O2-S -N -H 1 -0.10 0.0 6. N-Sulfates SCEE=1.0 SCNB=1.0
O2-S -N -CG 1 0.11 0.0 -3. SCEE=1.0 SCNB=1.0 N-Sulfates
                1 0.0 0.0 -2. SCEE=1.0 and SCNB=1.0
                1 0.0 0.0 1. SCEE=1.0 & SCNB=1.0
```

### 3.6.5 The basic idea behind the new commands

As has been mentioned before, quite a few new commands have been introduced into sleap. The ultimate goal of these new commands is that users will be able to generate topology files right from pdb files without calling any other programs such as antechamber. The easiest way to prepare a topology from a pdb file is to use the new keyword fastbld. Ideally the script would look like the following:

```
source leaprc.ff03.r1
source leaprc.gaff
set default fastbld on
xxx = loadpdb xxx.pdb
saveamberparm xxx xxx.top xxx.xyz
quit
```

### 3 LEaP

However, real world cases can not always be that simple. There are several issues which could interrupt the procedure. First, the *fixbond* command could fail on distorted structures. Fixbond uses the geometrical evidence to determine the bond orders, and won't work for distorted structures. Second, the *addhydr* command might not give the proper answer since it does not consider protonation states. Third, the *setpchg* command only assigns AM1-BCC charges to the residue. Sometimes users might want to use resp charges.

In all, experienced users might want to customize the procedure. They might use some of the new commands but not all of them. That is the reason the separate commands are provided. A template is the following:

```
source leaprc.ff03.r1
source leaprc.gaff
res = loadpdb res.pdb
fixbond res
addhydr res
setpchg res
parmchk res
all = loadpdb all.pdb
savemaberparm all all.top all.xyz
quit
```

Users may make changes to this script. For instance, one can assign the bond orders manually, save the result in sdf format (or mol2 format), then reload in sleap and do the rest, or one could even add hydrogens manually. In all, it is a highly customizable procedure.

## 4 Antechamber

This is a set of tools to generate files for organic molecules, 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).[69] See Ref. [70] 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.

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 parameterizations were carried out with HF/6-31G\* RESP charges. However, in most cases, AM1-BCC, which was parameterized 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 statistics derived from the Cambridge Structural Database, and ab initio calculations at the MP2/6-31G\* level. 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 to the AMBER macromolecular force fields. It should be noted that GAFF atom types are in lowercase except metals, 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 even can merge the two kinds of force fields into one file. The combined force fields are capable to study 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 an 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 ....

## 4.1 Principal programs

The antechamber program itself is the main program of Antechamber: if your molecule falls in fairly broad categories, this should be all you need to convert an input pdb file into files ready for LEaP. Otherwise you may use molecular formats that having bond information, such as mol2, sdf to run antechamber programs. If there are missing parameters after antechamber is finished, you may want to run parmchk to generate a frcmmod 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: *mopac* (or optionally, *divcon*), *atomtype*, *am1bcc*, *bondtype*, *espgen*, *respgen* and *prepgen*. It may also generate a lot of intermediate files (all in capital letters). If there is a problem with 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, default is MOL
-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  
 -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 is no  
 -i -o -fi and -fo must appear in command lines and the others are optional

#### 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
Charmm		charmm	23				

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

## 4 Antechamber

### Examples:

```
antechamber -i g98.out -fi gout -o sustiva_resp.mol2 -fo mol2 -c resp
antechamber -i g98.out -fi gout -o sustiva_bcc.mol2 -fo mol2 -c bcc -j 5
antechamber -i g98.out -fi gout -o sustiva_gas.mol2 -fo mol2 -c gas
antechamber -i g98.out -fi gout -o sustiva_cm2.mol2 -fo mol2 -c cm2
antechamber -i g98.out -fi gout -o sustiva.ac -fo ac
antechamber -i sustiva.ac -fi ac -o sustiva.mpdb -fo mpdb
antechamber -i sustiva.ac -fi ac -o sustiva.mol2 -fo mol2
antechamber -i sustiva.mol2 -fi mol2 -o sustiva.gzmat -fo gzmat
antechamber -i sustiva.ac -fi ac -o sustiva_gas.ac -fo ac -c gas
antechamber -i mtz.pdb -fi pdb -o mtz.mol2 -fo mol2 -c rc -cf mtz.charge
```

The `-rn` line specifies the residue name to be used; thus, it must be one to three characters long. The `-at` flag is used to specify whether atom types are to be created for the general AMBER force field (`gaff`) or for atom types consistent with `parm94.dat` and `parm99.dat` (`amber`). 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 this to `gaff`, even if you plan to use this as a ligand bound to a macromolecule described by the AMBER force fields.

### 4.1.2 parmchk

`Parmchk` reads in an `ac` file as well as a force field file (the default is `gaff.dat` in `$AMBERHOME/dat/leap/parm`). It writes out a force field modification (`frcmmod`) file for the missing or all force field parameters. Problematic parameters are indicated with "ATTN, need revision". Such parameters are typically 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. For each atom type, an atom type corresponding file (`ATCOR.DAT`) lists its replaceable general atom types. By the default, only the missing parameters are written to the `frcmmod` file. When the `'-a Y'` flag is used, `parmchk` prints out all force field parameters used by the input molecule, no matter whether they are already in the `parm` file or not. This file can be used to prepare the `frcmmod` file used by thermodynamic integration calculations using `sander`.

```
parmchk -i input file name
        -o frcmmod 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, caution should be taken to examine the output. Considering the complicate nature of the problem, users should certainly be on the lookout for unusual or incorrect behavior of the suite program of Antechamber.

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
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/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
```

#### 4 Antechamber

```
1 CG      -1.9590    0.1020    0.7950  ca    1 TP    -0.118600
2 CD1     -1.2490    0.6020   -0.3030  ca    1 TP    -0.113500
3 CD2     -2.0710    0.8650    1.9630  ca    1 TP     0.016500
4 CE1     -0.6460    1.8630   -0.2340  ca    1 TP    -0.137200
5 C6      -1.4720    2.1290    2.0310  ca    1 TP    -0.145300
6 CZ      -0.7590    2.6270    0.9340  ca    1 TP    -0.112400
7 HE2     -1.5580    2.7190    2.9310  ha    1 TP     0.129800
8 S15     -2.7820    0.3650    3.0600  sh    1 TP    -0.254700
9 H19     -3.5410    0.9790    3.2740  hs    1 TP     0.191000
10 H29    -0.7870   -0.0430   -0.9380  ha    1 TP     0.134700
11 H30     0.3730    2.0450   -0.7840  ha    1 TP     0.133500
12 H31    -0.0920    3.5780    0.7810  ha    1 TP     0.133100
13 H32    -2.3790   -0.9160    0.9010  ha    1 TP     0.143100

@<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
 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 frcmmod
```

This yields the frcmmod file:

```
remark goes here
MASS
BOND
ANGLE
DIHE
IMPROPER
```

## 4.2 A simple example for antechamber

```
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 a 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:

## 4 Antechamber

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,
```

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
```

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 compl.ac -o compl_bcc.ac -f ac -j 4
```

This command reads in compl.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. compl\_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.

```

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 /bin/*.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.

```
Usage: 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.

```
Usage: espngen -i input file name
           -o output file name
```

Example:

```
espngen -i sustiva_g98.out -o sustiva.esp
```

The above command reads in sustiva\_g98.out and writes out sustiva.esp, which can be used by the resp program. 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.

```
Usage: respgen -i input file name(ac)
        -o output file name
        -f output file format (resp1 or resp2)
            resp1 - first stage resp fitting
            resp2 - second stage resp fitting
        -a additional input data (predefined charges, atom groups etc)
        -n number of conformations (default is 1)
```

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:

```
respgen -i sustiva.ac -o sustiva.respin1 -f resp1
respgen -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
```

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*.

## 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
```

### 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

```
Charmm          charmm  23
```

---

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.

#### 4.4.2 database

Database reads in a multiple sdf or mol2 file and a definition file to run a set of commands for each record sequentially. The commands are defined in the definition file. It is noted that the database program can handle other well-organized file formats exemplified by the all\_amino94.in file in dat/leap/parm as well. The definition file also describes how to dissect records and how to name them, as well as rules of selecting a subset of the database. A more detailed sample input file is in *\$AMBERHOME/test/antechamber/database*.

```
Usage: database -i database file name
          -d definition file name
```

Example:

```
database -i sample_database.mol2 -d mol2.def
```

This command reads in a multiple mol2 database - sample\_database.mol2 and a description file mol2.def to run a set of commands (defined in mol2.def) to generate prep input files and merge them to a single file called total.prepi. Both files are located in the following directory: *\$AMBERHOME/test/antechamber/database/mol2*.

#### 4.4.3 parmcal

Parmcal is an interactive program to calculate the bond length and bond angle parameters, according to the rules outlined in Ref. [69].

```
Please select:
```

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

#### 4.4.4 residuegen

It can be painful to prepare an amino-acid-like residues. In AMBER 10, a new program, *residuegen*, is developed to facilitate the residue topology generation. The program reads in an input file and applies a set of antechamber programs to generate residue topologies in prepri 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
```

#### 4.4.5 top2ff

Top2ff extracts the force field parameter information in a topology file and save it in `frmod` format. It is useful if one is not sure what kinds of force field parameters he used for his MD simulations.

Usage: `top2ff -p topology file name`  
`-o output file name in frmod format`

Example:

```
top2ff -i lbe9.prmtop -c lbe9.frcmod
```

## 4 Antechamber

This command reads in a topology file (1be9.prmtop) and write out all the force field parameters into a frcmod file. Examination on the output can reveal which force field is used in the topology file.

### 4.4.6 top2mol2

Top2mol2 essentially resembles ambpdb. It reads in an AMBER topology file and an AMBER restart file and writes out a mol2 file. This mol2 file can be read by many software packages, such as sybyl, chimera, pymol, etc. The atom types in the mol2 file can be either sybyl (the default) or amber (AMBER and GAFF atom types). The sybyl atom type information is obtained from an atom type corresponding file (ATOMTYPE\_CHECK.TAB). Similarly, a bond type corresponding file (BONDTYPE\_CHECK.TAB) is used to assign bond types (single, double, triple and aromatic). The two parameter files are located in the dat/antechamber sub-directory.

```
Usage: top2mol2 -p topology file name
          -c rst or crd file name
          -o output file name)
          -ac atom type corresponding file (optional)
          -bc bond type corresponding file (optional)
          -at atom type: sybyl (the default) or amber, optional
          -bt bond type: sybyl (the default) or amber (all set to 1), optional
          -wt keep water flag: 1 (including water) or 0 (removing water, the def
```

Example:

```
top2mol2 -i 1be9.prmtop -c 1be9_md.rst -c 1be9_md.mol2
top2mol2 -i 1be9.prmtop -c 1be9_md.rst -c 1be9_md.mol2 -at amber
```

This first command reads in a topology file (1be9.prmtop) and a MD restart file (1be9\_md.rst) then writes out a mol2 file with the sybyl atom types. The second command, on the other hand, keeps the original atom types in the topology file.

### 4.4.7 translate

Translate reads a pdb, ac or mol2 file and writes out a file in the same format after an operation. The supported actions include, dimension check ('check'), centerization ('center'), translation in three dimensions ('translate'), rotation along an axis defined by two atoms ('rotate1') or two space points ('rotate2'), least-squares fitting ('match'), alignment to X, Y or Z-axis ('alignx', 'aligny' and 'alignz'). The manipulation of molecules with this program may be useful in manual docking and molecular complexes modeling, such as membrane protein construction.

```
translate -i input file name (pdb, ac or mol2)
          -o output file name
          -r reference file name
```

## 4.5 New Development of Antechamber And GAFF

```
-f file format
-c command (check, center, translate, rotate1, rotate2, match)
  center:      need -a1;
  translate:   need -vx, -vy and -vz;
  rotate1:    need -a1, -a2 and -d;
  rotate2:    need -x1, -y1, -z1, -x2, -y2, -z2 and -d;
  match:      need -r;
  alignx:     align to X-axis, need -x1, -y1, -z1, -x2, -y2, -z2;
  aligny:     align to Y-axis, need -x1, -y1, -z1, -x2, -y2, -z2;
  alignz:     align to Z-axis, need -x1, -y1, -z1, -x2, -y2, -z2;
  image:      reflect by (0,0,0)
  imagex:     reflect by Y-Z plane
  imagey:     reflect by X-Z plane
  imagez:     reflect by X-Y plane
-d degree to be rotated
-vx x vector
-vy y vector
-vz z vector
-a1 id of atom 1 (0 coordinate center)
-a2 id of atom 2
-x1 coord x for point 1
-y1 coord y for point 1
-z1 coord z for point 1
-x2 coord x for point 2
-y2 coord y for point 2
-z2 coord z for point 2
```

Example:

```
translate -i 2rh1.pdb -f pdb -c alignz -x1 -33.088 -x2 -33.088 -y1 -14.578
-y2 50.061 -z1 7.0287 -z2 7.0287 -o 2rh1_Z.pdb
translate -i 2rh1_Z.pdb -f pdb -c rotate2 -x1 0 -x2 0 -y1 0 -y2 0 -z1 -10
-z2 10 -o 2rh1_Z60.pdb -d 60
```

This first command align a GPCR crystal structure, *2rh1* from Y-axis to Z-axis to get protein *2rh1\_Z.pdb*. Then the second command rotates *2rh1\_Z* 60 degrees along the Z-axis to get *2rh1\_Z60.pdb*.

## 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

## 4 Antechamber

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 sp<sup>2</sup> 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 CH<sub>3</sub>Br (1.6755, 20 degree) and CH<sub>3</sub>I (2.2789, 20 degree): 1.642 for CH<sub>3</sub>Br and 2.25 for CH<sub>3</sub>I, in contrast, the old parameters give 1.31 and 1.84, respectively. (Junmei, unpublished result)
3. New van der Waals parameters have been suggested by David Mobley for 'c1', 'cg' and 'ch' atom types. The justification of the changes is discussed at <http://dx.doi.org/10.1021/ct800409d>

#### 4.5 New Development of Antechamber And GAFF

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.



# 5 Semiempirical quantum chemistry

## 5.1 Introduction

AmberTools now contains its own quantum chemistry program, called *sgm*. 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 energy minimizations for a wide variety of Hamiltonians, including many recent ones. Our plan is to add capabilities to subsequent versions.

Support currently exists for gas phase simulations. Available semi-empirical Hamiltonians are PM3,[77] AM1,[78] RM1,[79] MNDO,[80] PDDG/PM3,[81] PDDG/MNDO,[81] and PM3CARB1.[82] PM6 is supported for all elements which do not require d-orbitals. Support is also available for the Density Functional Theory-based-tight-binding (DFTB) Hamiltonian,[83–85] as well as the Self-Consistent-Charge version, SCC-DFTB.[86] DFTB/SCC-DFTB also supports approximate inclusion of dispersion effects,[87] as well as reporting CM3 charges [88] for molecules containing only the H, C, N, O, S and P atoms and third-order corrections[89].

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, Sn, I, Hg, Pb
AM1: H, C, N, O, F, 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
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
PM6: H, He, Li, Be, B, C, N, O, F, Ne, Na, Mg, Ar, K, Ca, Zn, Ga, Ge, Kr, Rb,
DFTB/SCC-DFTB: (Any atom set available from the www.dftb.org website)
```

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](http://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 parameterizations 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.

The semi-empirical support was written by Ross Walker, Mike Crowley, and Dave Case,[90] based originally on public-domain MOPAC codes of J.J.P. Stewart. The generalised Born implementation uses the model described by Pellegrini and Field[91]. SCC-DFTB support was written by Gustavo Seabra, Ross Walker and Adrian Roitberg,[83] and is based on earlier work of Marcus Elstner.[86, 92] Support for third-order SCC-DFTB was written by Gustavo Seabra and Josh McClellan.

## 5.2 General &qmmm Namelist Variables

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, PM6, and DFTB.

*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](http://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 parameterization, 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 parameterization, which was developed for phosphate hydrolysis reactions. The parameters will be read from the *\$AMBER-HOME/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.

## 5 Semiempirical quantum chemistry

- = **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 on 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  $0.05 * \sqrt{\text{scfconv}}$ .
- = **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.
- = **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 *ntpr* steps.

`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  $\phi$  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 14.4) 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.

## 5 *Semiempirical quantum chemistry*

- `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 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.

## 6 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 1 100 1
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

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.

## 6.1 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.

## 6.2 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* ]

*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.

### 6.3 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.

## 6.3 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 Angstroms) 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

## 6 ptraj

following command will set the x-component of the box size to be 100.0 Angstroms 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. 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 what the current solvent information is (or more generally get some 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.

## 6.4 ptraj *action* commands - short list

## 6.5 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 6.1. 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.

## 6.5 ptraj action commands - detailed discussion

<i>action</i>	<i>description of the action</i>
<b>analyze</b>	calculate summary properties for values accumulated during analysis, such as distances, etc.
<b>angle</b>	compute the angle in degrees between the center of mass of the three atom specifications listed
<b>atomicfluct</b>	compute the atomic positional fluctuations (RMSF) or B-factors: Note this does not implicitly perform an RNA fit prior
<b>average</b>	compute the straight coordinate average of the coordinate sets read in: Note this does not implicitly perform an RNA fit prior
<b>center</b>	move all the coordinates to the origin or box center
<b>checkoverlap</b>	look for atoms that are closer (and optionally further separated than) a specified distance
<b>closest</b>	strip the trajectories such that only the closest N (chosen by the user) solvent molecules are retained
<b>cluster</b>	group configurations from the MD trajectory into distinct sets (and report average, representative and trajectory) structures
<b>contacts</b>	find the number of atom interactions (and native contacts) within a given distance
<b>dihedral</b>	calculate the dihedral values from the center of mass of the four specified atom masks
<b>diffusion</b>	calculate the mean-squared displacements vs. frame number
<b>dipole</b>	construct a grid of solvent dipoles (note: this command is obsolete and the user should consult the code)
<b>distance</b>	calculated the distance in Angstroms between the center of mass of the two atom specifications listed
<b>grid</b>	create a grid in X-Plor density format that has a count of the number of selected atoms in each grid cell
<b>hbond</b>	described in a separate section, this command keeps track of distances and angles between pairs of atoms
<b>image</b>	with periodic boundary conditions, bring molecules outside the periodic box into the central unit cell
<b>matrix</b>	calculate 2D correlations and fluctuations; note that this is described in a separate section of the manual
<b>principal</b>	align to the principal coordinate axis
<b>pucker</b>	compute the pucker for 5-membered rings
<b>radial</b>	compute a radial distribution function
<b>radgyr</b>	determine the radius of gyration for each frame
<b>randomizeions</b>	move single atom ion residues to new positions by swapping with random water molecules
<b>rms</b>	compute the RMSd to the first frame or a reference frame
<b>secstruct</b>	determine the secondary structure of proteins
<b>strip</b>	remove atoms from the trajectory
<b>translate</b>	move the entire system in the x, y and/or z directions
<b>trunctoct</b>	create an old style AMBER truncated octahedron; this is obsolete
<b>watershell</b>	determine how many solvent models are within a given radius of the solute
<b>vector</b>	calculate and analyze various different vectors

Table 6.1: Summary of ptraj action commands

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  $(8/3)\pi^{**2}$ ). 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 (Angstroms) or B-factor ( $\text{Angstroms}^{**2} * (8/3)\pi$ ).

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:

```
angle 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

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 angstroms, `min` 0.95 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 Angstroms 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. [93], and users should consult that paper for background and details. A simple example is as follows:

```

trajin traj.1.gz
trajin traj.2.gz
cluster out testcluster representative pdb \
    average pdb average-linkage 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. [93] 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 *s*th 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 “FullPairwiseMatrix” 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.

**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
```

**dihedralcluster** *out filename cut clustersize\_cutoff framefile filename clusterinfo filename\*

[*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 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
dihedrall_atom1 dihedrall_atom2 dihedrall_atom3 dihedrall_atom4
...
#CLUSTERS
CLUSTERNUM1 CLUSTERSIZE1 DIHEDRALBINID1
...
```

If a cutoff is specified by CUT only clusters with population greater than CUT will be printed.

**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 angstroms\*\*2) 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 (*i.e.* 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 & Sundarlingam, *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/angstrom\*\*3 which corresponds to a density of water equal to ~1.0 g/mL. To convert a standard density in g/mL, multiply the density by "6.022 / (10 \* weight)" where "weight" 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 angstroms from residues 1-20 and no closer than 3.0 angstroms 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.[94] 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.

**truncocct** *mask distance* [ *prmtop filename* ]

Note: This command is largely obsolete.

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 **truncocct** 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).

**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.

## 6.6 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* | *corrired mask2* ] \n [ *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 ( $0 \leq \text{order} \leq 2$ ). 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 ( $0 \leq \text{order} \leq 2$ ). It defaults to 2.

**corrred *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 [95] can be calculated. *order* 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 corrred!) 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*), [96] or Isotropic Reorientational Eigenmode Dynamics (*ired*)

[95] 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  $\text{cm}^{-1}$  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  $\geq 1$ , 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 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*]  
 [`drc`] [`dplr`] [`norm`] `out` *filename* [`noefile` *noefilename*]

Calculates time correlation functions for vectors *vecname1* (*vecname2*) of type “`corr`” or “`corrired`”, 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 `drc` 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.  $C_i(t=0) = P_i(t=0) = 1.0$ . 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 “*corrired*” and when they represent an N-H bond. If *relax* is given, correlation times  $\tau_m$  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  $\tau_m$  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  $\tau_m$  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).

## 6.7 Parallel ptraj

Ptraj has been modified to include parallel support using MPI. To take advantage of this feature, run the *configure* script with the *-mpi* flag (e.g. *configure -mpi gnu*), then run *make* with the parallel command (e.g. *make -f Makefile\_at\_parallel*). This will produce an executable in the \$AMBERHOME/bin/ directory called *ptraj.MPI*. **Note that currently *ptraj.MPI* only performs well if run on a local disk** (i.e. on a single multi-processor machine). Running on network file systems such as NFS actually cause the performance of *ptraj.MPI* to be worse than *ptraj* due to the generally poor performance of these file systems in parallel.

The same ptraj source files are used to build ptraj in both serial and parallel, using preprocessor definitions where needed. There is no change to the syntax of the ptraj commands, so previous ptraj command scripts should work and produce the same output. Currently not all

actions are supported; a list of the ones that currently run in parallel is included below. Commands given to parallel ptraj which are not in the list below will be ignored by the program (after a warning message is printed).

A large change to both the parallel and serial versions of ptraj is in the way AMBER trajectory files are read in. In the old version, the whole file would be read in and checked for errors first, then read in a second time to perform the actions. On small trajectory files this was not a problem, but for larger files this could take significantly more time. Now the file is checked while the actions are being performed, which eliminates the need to read the whole file twice. Secondly, the way in which the coordinates are read has been optimized speeding up the read time. Finally, seeking is used with the trajectory file in order to skip past the frames that do not need to be processed. Before these frames would be read in regardless.

For example, if a trajectory file contains 10,000 frames, and we are only interested in frames 2000 to 5000, skipping every other frame (offset of 2), the old ptraj would read in all frames from 1 to 5000. The new ptraj will now seek to the 2000th frame, and then only read in every other frame by seeking to the new file position. So instead of reading in 5000 frames, it will read in 1500  $((5000 - 2000) / 2)$ . Compressed files are one limitation to this because of the way they are decompressed by the program. Thus compressed AMBER trajectory files have the benefit of being smaller, but have the downside of additional time needed to decompress the files, plus the inability to skip past unprocessed frames.

#### Supported Actions:

`angle, atomicfluct, average, center, checkoverlap, closest, contacts, dihedrals, distance, image, principal, pucker, radgyr, randomizeions, rms, strip, translate, watershell`

#### Known Issues:

- You cannot use the `previous` option with the **rms** action because frames are divided among processors and do not have previous frame information.
- **randomizeions** does not produce the same output trajectory as the serial version, even when using same seed, because frames are not analyzed in the same order.
- `ptraj.MPI` does not perform well on network file systems such as NFS due to underlying issues with the file system - `ptraj.MPI` should be run on the local disk of a multi-processor machine for optimal performance.

## 6.8 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.

### 6.8.1 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 quasiharmonic analysis, whereby 5 eigenvectors and eigenvalues are calculated and output to evecs.dat.

```
matrix mwcovar name mwcvmat out mwcvmat.dat
analyze matrix mwcvmat out evecs.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 evecs vecs 5
analyze modes fluct out rmsfluct.dat stack evecs beg 1 end 3
analyze modes displ out resdispl.dat stack evecs beg 1 end 3
```

Finally, dipole-dipole correlation functions for modes obtained from principle component analysis or quasiharmonic analysis can be computed.

```
analyze modes corr out cffromvec.dat stack evecs beg 1 end 3 ...
... maskp @1 @2 maskp @3 @4 maskp @5 @6
```

### 6.8.2 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 evecs.dat (which have been obtained by the "matrix mwcovar", "analyze matrix" commands from above).

```
projection modes evecs.dat out project.dat beg 1 end 2
```

### 6.8.3 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 timestep 1.0 tcorr 100.0 out v0.out
analyze timecorr vec1 v1 timestep 1.0 tcorr 100.0 out v1.out
analyze timecorr vec1 v0 vec2 v1 timestep 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.c
```

## 6.9 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.

**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.

The **acceptor** command is similar except that both the heavy atom and the hydrogen atom are specified. 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.

**acceptor** *resname atomname atomnameH* | *mask mask maskH* | *clear* | *print*

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* ]

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<sup>+</sup> ions (assuming the atom name was Na<sup>+</sup> and residue name was also Na<sup>+</sup>):

```
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.

## 6.10 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.

**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.

## 6.11 AMBER Trajectory NetCDF Format

*John Mongan (jmongan@mccammon.ucsd.edu)*

The file format described in this document was developed for storing data generated by molecular dynamics simulations. It was introduced in version 9 of the AMBER suite of programs (<http://amber.scripps.edu>).

The primary design goals of this format are:

- Efficient input and output
- Compact, high-precision representation of data
- Portability of data files across different machine architectures
- Extensibility of the format (ability to add additional data without re-writing parsers)
- Compatibility with existing tools and formats

The file format is based on the NetCDF (Network Common Data Form) developed by Unidata (<http://www.unidata.ucar.edu/software/netcdf/>). NetCDF is designed for representation of arbitrary array-based data. Unidata provides libraries with bindings in C, C++, Fortran (F77 and F90), Java, Python, Perl, Ruby and MATLAB for reading and writing NetCDF files. The design goals above are largely met by NetCDF and the libraries that implement it. It is expected that all I/O of the format described here will occur through these libraries; this specification describes the file format at a high level in terms of the API implemented by version 3.6 of these libraries. In NetCDF terms, this document is a “Convention,” describing the names, dimensions and attributes of the arrays that may be present in the file.

### 6.11.1 Program behavior

Programs creating trajectory files (“creators”) shall adhere strictly to the requirements of this document. Programs reading trajectory files (“readers”) shall be as permissive as possible in applying the requirements of this document. Readers may emit warnings if out-of-spec files are encountered; these warnings should include information about the program that originally created the file (see Global attributes, section 6.11.3). Readers shall not fail to read a file unless the required information cannot be located or interpreted. In particular, to ensure forward compatibility with later extensions of the format, readers shall not fail or emit warnings if elements not described in this document are present in the file.

### 6.11.2 NetCDF file encoding

- Trajectory files shall be encoded in the manner employed by NetCDF version 3.x.  
Those using NetCDF versions 4 or later should take care to ensure that files are read and written using this encoding, and not the HDF5 encoding.
- Trajectory files shall use 64 bit offsets  
This can be accomplished by setting a flag during file creation; refer to API docs for details.

### 6.11.3 Global attributes

Global attributes shall have type character string. Spelling and capitalization of attribute names shall be exactly as appears below. Creators shall include all attributes marked required and may include attributes marked optional. Creators shall not write an attribute string having a length greater than 80 characters. Readers may warn about missing required attributes, but shall not fail, except in the case of a missing or unexpected *Conventions* or *ConventionVersion* attributes.

- Conventions (required)

Contents of this attribute are a comma or space delimited list of tokens representing all of the conventions to which the file conforms. Creators shall include the string *AMBER* as one of the tokens in this list. In the usual case, where the file conforms only to this convention, the value of the attribute will simply be “AMBER”. Readers may fail if this attribute is not present or none of the tokens in the list are *AMBER*. Optionally, if the reader does not expect NetCDF files other than those conforming to the *AMBER* convention, it may emit a warning and attempt to read the file even when the *Conventions* attribute is missing.

- ConventionVersion (required)

Contents are a string representation of the version number of this convention. Future revisions of this convention having the same version number may include definitions of additional variables, dimensions or attributes, but are guaranteed to have no incompatible changes to variables, dimensions or attributes specified in previous revisions. Creators shall set this attribute to “1.0”. If this attribute is present and has a value other than “1.0”, readers may fail or may emit a warning and continue. It is expected that the version of this convention will change rarely, if ever.

- application (optional)

If the creator is part of a suite of programs or modules, this attribute shall be set to the name of the suite.

- program (required)

Creators shall set this attribute to the name of the creating program or module.

- programVersion (required)

Creators shall set this attribute to the preferred textual formatting of the current version number of the creating program or module.

- title (optional)

Creators may set use this attribute to represent a user-defined title for the data represented in the file. Absence of a title may be indicated by omitting the attribute or by including it with an empty string value.

#### 6.11.4 Dimensions

- frame (required, length unlimited)

Coordinates along the frame dimension will generally represent data taken from different time steps, but may represent arbitrary conformation numbers when the trajectory file does not represent a true trajectory but rather a collection of conformations (e.g. from clustering).

- spatial (required, length 3)

This dimension represents the three spatial dimensions (X,Y,Z), in that order.

- atom (required, length set as appropriate)

Coordinates along this dimension are the indices of particles for which data is stored in the file. The length of this dimension may be different (generally smaller) than the actual number of particles in the simulation if the user chooses to store data for only a subset of particles.

- cell\_spatial (optional, length 3)

This dimension represents the three lengths (a,b,c) that define the size of the unit cell.

- cell\_angular (optional, length 3)

This dimension represents the three angles (alpha,beta,gamma) that define the shape of the unit cell.

- label (optional, length set as appropriate)

This dimension is used for character strings in label variables where the label is longer than a single character. The length of this dimension is equal to the length of the longest label string.

#### 6.11.5 Label variables

Label variables shall be written by creators whenever their corresponding dimension is present. These variables are for self-description purposes, so readers may generally ignore them. Labels requiring more than one character per coordinate shall use the label dimension. Individual coordinate labels that are shorter than the length of the label dimension shall be space padded to the length of the label dimension.

- char spatial(spatial)

Creators shall write the string “xyz” to this variable, indicating the labels for coordinates along the spatial dimension.

- char cell\_spatial(cell\_spatial)

Creators shall write the string “abc” to this variable, indicating the labels for the three lengths defining the size of the unit cell.

- char cell\_angular(cell\_angular, label)

Creators shall write the strings “alpha”, “beta”, “gamma” to this variable, naming the angles defining the shape of the unit cell.

### 6.11.6 Data variables

All data variables are optional. Some data variables have dependencies on other data variables, as described below. Creators shall define a *units* attribute of type character string for each variable as described below. Creators may define a *scale\_factor* attribute of type float for each variable. Creators shall ensure that the units of data values, after being multiplied by the value of *scale\_factor* (if it exists) are equal to that described by the *units* attribute. If a *scale\_factor* attribute exists for a variable, readers shall multiply data values by the value of the *scale\_factor* attribute before interpreting the data. This scaling burden is placed on the reader rather than the creator, as writing data is expected to be a more time-sensitive operation than reading it.

It is left as an implementation detail whether creators create a separate file for each variable grouping (e.g. coordinates and velocities) or a single file containing all variables. Some creators may allow the user to select the approach. Readers should support reading both styles, that is, combining data from multiple files or reading it all from a single file.

- float time(frame) units = “picosecond”

When coordinates on the frame dimension have a temporal sequence (e.g. they form a molecular dynamics trajectory), creators shall define this dimension and write a float for each frame coordinate representing the number of picoseconds of simulated time elapsed since the start of the trajectory. When the file stores a collection of conformations having no temporal sequence, creators shall omit this variable.

- float coordinates(frame, atom, spatial) units = “angstrom”

This variable shall contain the Cartesian coordinates of the specified particle for the specified frame.

- float cell\_lengths(frame, cell\_spatial) units = “angstrom”

When the *coordinates* variable is included *and* the data in the *coordinates* variable come from a simulation with periodic boundaries, creators shall include this variable. This variable shall represent the lengths (a,b,c) of the unit cell for each frame. When each of the angles in *cell\_angles* is 90, a, b and c are parallel to the x, y and z axes, respectively. If the simulation has one or two dimensional periodicity, then the length(s) corresponding to spatial dimensions in which there is no periodicity shall be set to zero.

- float cell\_angles(frame, cell\_angular) units = “degree”

Creators shall include this variable if and only if they include the *cell\_lengths* variable. This variable shall represent the angles ( $\alpha, \beta, \gamma$ ) defining the unit cell for each frame.  $\alpha$  defines the angle between the a-b and a-c planes,  $\beta$  defines the angle between the a-b and b-c planes and  $\gamma$  defines the angle between the a-c and b-c planes. Angles that are undefined due to less than three dimensional periodicity shall be set to zero.

- float velocities(frame, atom, spatial) units = "angstrom/picosecond"

When the *velocities* variable is present, it shall represent the cartesian components of the velocity for the specified particle and frame. It is recognized that due to the nature of commonly used integrators in molecular dynamics, it may not be possible for the creator to write a set of velocities corresponding to exactly the same point in time as defined by the *time* variable and represented in the *coordinates* variable. In such cases, the creator shall write a set of velocities from the nearest point in time to that represented by the specified frame.

### 6.11.7 Example

The following is an example of the CDL for a trajectory file conforming to the preceding specification and containing most of the elements described in this document. This CDL was generated using `ncdump -h <trajectory file>`.

```
netcdf mdtrj {
dimensions:
    frame = UNLIMITED ; // (10 currently)
    spatial = 3 ;
    atom = 28 ;
    cell_spatial = 3 ;
    cell_angular = 3 ;
    label = 5 ;
variables:
    char spatial(spatial) ;
    char cell_spatial(cell_spatial) ;
    char cell_angular(cell_angular, label) ;
    float time(frame) ;
        time:units = "picosecond" ;
    float coordinates(frame, atom, spatial) ;
        coordinates:units = "angstrom" ;
    float cell_lengths(frame, cell_spatial) ;
        cell_lengths:units = "angstrom" ;
    float cell_angles(frame, cell_angular) ;
        cell_angles:units = "degree" ;
    float velocities(frame, atom, spatial) ;
        velocities:units = "angstrom/picosecond" ;
        velocities:scale_factor = 20.455f ;
// global attributes:
    :title = "netCDF output test" ;
    :application = "AMBER" ;
    :program = "sander" ;
    :programVersion = "9.0" ;
    :Conventions = "AMBER" ;
    :ConventionVersion = "1.0" ;
```

6 traj

}

## 7 PBSA

Several efficient finite-difference numerical solvers, both linear [97, 98] and nonlinear,[99] are implemented in *pbsa* for various applications of the Poisson-Boltzmann (PB) method. In the following, a brief introduction to the PB method, the numerical solvers, and numerical energy and force calculations is given first. This is followed by a detailed description of the usage and keywords. Finally example input files are explained for typical PB applications. For more background information and how to use the PB method, please consult cited references and online *Amber* tutorial pages.

### 7.1 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.[100] 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 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 [101]).

The Poisson-Boltzmann (PB) solvents are a class of widely used implicit solvents to model solvent-mediated electrostatic interactions.[100] 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.[100] 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.[102] 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:[103, 104]

$$\nabla \cdot [\epsilon(\mathbf{r})\nabla\phi(\mathbf{r})] = -4\pi\rho(\mathbf{r}) - 4\pi\lambda(\mathbf{r})\sum_i z_i c_i \exp(-z_i\phi(\mathbf{r})/k_B T) \quad (7.1)$$

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 solvation interactions are typically modeled with a term proportional to the solvent accessible surface area (SASA).[105] An alternative and more accurate method to model the non-polar solvation interactions is also implemented in this release.[106] 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 for branched and cyclic organic molecules.[107] 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.[105] In the new 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.[108]

### 7.1.1 Numerical solutions of the PB equation

In this release, both the linear form and the full nonlinear form of the PB equation are supported. Many numerical methods may be used to solve the PB equation, but only the finite-difference (FD) method [109–111] is supported in the current release 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 this release, four common linear FD solvers are implemented: modified ICCG, geometric multigrid, conjugate gradient, and successive over-relaxation (SOR).[98] 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.[99]

## 7.1.2 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  $\phi$ , which can be computed by solving the linear or nonlinear system.

It has been shown (see for example [101]) 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.[112]

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.[113]

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 this release. 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,[100] 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 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.[106] 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.[105] You can also use SASA to correlate the cavity term only and use a surface-integration approach to compute the dispersion term.[106] 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.[106] Otherwise, there is no guarantee of consistence between the implemented non-polar implicit solvent and the TIP3P explicit solvent. In this release, more options are added in the second approach, i.e. when INP = 2. See the discussion of keywords following INP below. These options are described in detail in Ref. [106].

### 7.1.3 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,[101] 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 when the weighted harmonic average of dielectric constants is used at the solute/solvent interface (when SMOOTHOPT = 1, see below).[114]

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.[106]

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.

## 7.2 Usage and keywords

### 7.2.1 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.

### 7.2.2 Basic input options

The layout of the input file is also 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. 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 the coming 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 or 2. The backward compatible flag IGB = 10 is equivalent to IPB = 1 and will be phased out in future releases. The numerical non-polar procedures can be turned on by setting INP to either 1 or 2. The backward compatible flag NPOPT will be phased out in future releases.

**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. Note the current release of *pbsa* does not support either minimization or dynamics.

**= 0** No minimization. Default.

**= 1** Same as 0, but with a slightly different output format

## 7 PBSA

- 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.
- = 0 No electrostatic solvation free energy is computed.
  - = 1 The dielectric interface between solvent and solute is defined to be the numerical solvent excluded surface. Default.
  - = 2 The dielectric interface is also the solvent excluded surface, but it is implemented with the level set function, which is the signed distance from the numerical solvent accessible surface. The solvent excluded surface is the isosurface where the function value equals to negative solvent probe radius. [Wang and Luo, Manuscript in preparation] Use of a level set function simplifies the calculation of the intersection points of the solvent excluded surface and grid edges (when SMOOTHOPT = 1, see below) and leads to more stable numerical calculations.
- igb** When set to 10 (the only allowed value), it instructs *pbsa* to set up calculations, equivalent to the IPB = 1 option. Other values will cause *pbsa* to issue a warning and exit. It will be over written by IPB if inconsistency between the two is detected.
- 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. See Introduction. Default for backward-compatibility.
  - = 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 [106] closely related to the PCM solvent for quantum chemical programs.[108] 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 prmtpop file. Otherwise, a warning will be issued in the output file.

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.

### 7.2.3 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 the solute region. i.e. only two dielectric regions are allowed in the current release.
- smoothopt** instructs PB how to set up dielectric values for finite-difference grid edges that are located across the solute/solvent dielectric boundary.
- = 0** The dielectric constants of the boundary grid edges are always set to the equal-weight harmonic average of EPSIN and EPSOUT. Default.
  - = 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.[115]
  - = 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 *Amber*. Note also that we are only dealing with symmetrical solution, so the ionic strength should be equal to the square of the valence of the symmetrical ions times the ion concentration (in mM).
- pbtemp** Temperature used for the PB equation, needed to compute the Boltzmann factor for salt effects; default is 300 K.
- radiopt** Option to set up atomic radii.
- = 0** Use radii from the prmtop file for both the PB calculation and for the NP calculation (see INP).
  - = 1** Use atom-type/charge-based radii by Tan and Luo [116] for the PB calculation. Note that the radii are optimized for *Amber* atom types as in standard residues from the Amber database. If a residue is build by *antechamber*, i.e. if GAFF atom types are used, radii from the prmtop file will be used. Please see [116] on how these radii are optimized. The procedure in [116] can also be used to optimize radii for non-standard 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 *pbsa* 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.

## 7 PBSA

dprob	Solvent probe radius for molecular surface used to define the dielectric boundary between solute and solvent. To be backward compatible with previous releases in <i>Amber</i> , DPROB = SPROB = 1.6 by default, <i>i.e.</i> it is set to be equal to SPROB if DPROB is not specified in the input file. In this release, SPROB has been reserved for the calculation of non-polar solvation energy. See below.
iprob	Mobile ion probe radius for ion accessible surface used to define the Stern layer. Default to 2.0 Å.
arcres	<i>pbsa</i> uses a numerical method to compute solvent accessible arcs,[Wang 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 0.0625 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.

### 7.2.4 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. = 2 Geometric multigrid. A four-level v-cycle implementation is applied. Each dimension of the finite-difference grid is $2^{4*n-1}$ . = 3 Conjugate gradient. This option requires a large MAXITN to converge. = 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 .
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.
npgrid	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.

### 7.2.5 Options to compute energy and forces

bcopt	Boundary condition options. <ul style="list-style-type: none"> <li>= 1 Boundary grid potentials are set as zero. Total electrostatic potentials and energy are computed.</li> <li>= 5 Computation of boundary grid potentials using all grid charges. Total electrostatic potentials and energy are computed. Default.</li> <li>= 6 Computation of boundary grid potentials using all grid charges. Reaction field potentials and energy are computed with the charge singularity free formalism.<a href="#">[113]</a></li> </ul>
eneopt	Option to compute total electrostatic energy and forces. <ul style="list-style-type: none"> <li>= 1 Compute total electrostatic energy and forces with the particle-particle particle-mesh procedure outlined in Lu and Luo.<a href="#">[101]</a> 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.<a href="#">[101]</a> This option requires a non-zero CUTNB and BCOPT = 5.</li> <li>= 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.</li> </ul>

## 7 PBSA

- 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* – .
  - = 2** Use dielectric boundary surface polarized charges to compute the electrostatic forces and dielectric boundary forces [Ye *et al.* Manuscript submitted]. The forces are output in the unit of *kcal/mol* – .
  - = 3** Reaction field forces are computed using dielectric boundary polarized charge. Dielectric boundary forces are computed using the electric field on dielectric boundary [Ye *et al.* Manuscript submitted]. The forces are output in the unit of *kcal/mol* – .
- dbfopt** This keyword is equivalent to ENEOPT, and will be phased out in future releases.
- = 0** equivalent to ENEOPT = 1.
  - = 1** equivalent to ENEOPT = 2.
- 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.
- cutres** Residue-based cutoff distance; default is 24 Å. The residue-based non-bonded list is used to make the generation of the atom-based pair list efficient. A value longer than the largest residue in the system is recommended for correct nonbonded list generation.
- cutfd** Atom-based cutoff distance to remove short-range finite-difference Coulombic interactions, and to add pairwise Coulombic interactions, default is 5 Å. See Eqn (20) in Lu and Luo.[101]
- 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.
- nsnbr** Sets how often residue-based pairlist is generated; default is 1 step. For molecular dynamics simulations, a value of 25 is recommended.
- nsnba** Sets how often atom-based pairlist is generated; default is 1 step. For molecular dynamics simulations, a value of 5 is recommended.

### 7.2.6 Options for visualization and output

- phiout** *pbsa* 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 *pbsa.phi* 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 ( $kT/mol \cdot e$ ) is printed in the *Delphi* binary format. Default.
  - = 1** The electrostatic potential ( $kcal/mol \cdot e$ ) is printed in the *Amber* ASCII format.
- npbverb** When set to 1, turns on verbose mode in *pbsa*; default is 0.

### 7.2.7 Options to select a non-polar solvation treatment

- npopt** This keyword is equivalent to INP, and will be phased out in future releases. It will be over written by INP if inconsistency between the two is detected.
- decompopt** Option to select different decomposition schemes when  $INP = 2$ . See [106] for a detailed discussion of the different schemes. The default is 1, to be backward compatible with previous releases in *Amber*. However, the recommended option is  $DECOMPOPT = 2$ , the  $\sigma$  decomposition scheme, which is the best of the three schemes studied.[106] As discussed in Ref. [106],  $DECOMPOPT = 1$  is not a very accurate approach even if it is more straightforward to understand the decomposition.
- = 1** The 6/12 decomposition scheme. Default.
  - = 2** The  $\sigma$  decomposition scheme.
  - = 3** The WCA decomposition scheme.
- use\_rmin** The option to set up van der Waals radii for  $INP = 2$ . The default is not to use *rmin* to be backward compatible with previous releases in *Amber*. However, use of *rmin* improves the agreement with TIP3P [106], so it is recommended.
- = 0** Use atomic van der Waals  $\sigma$  values. Default.
  - = 1** Use atomic van der Waals *rmin* values.
- sprob** Solvent probe radius for solvent accessible surface area (SASA) used to compute the dispersion term, default to 1.600 Å, the sigma value of the TIP3P OW atom. The recommended value is 0.557 Å in the  $\sigma$  decomposition scheme as optimized in Ref. [106] with respect to the TIP3P solvent and the PME treatment. Recommended values for other decomposition schemes can be found in Table 4 of [106]. If  $USE\_SAV = 0$  (see below), **SPROB** can be used to compute SASA for the cavity

## 7 PBSA

term as well. Unfortunately, the recommended value is different from that used in the dispersion term calculation as documented in Ref. [106] 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. [106] 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 [106] 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. [106, 116]

- `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 optimized in [106] with respect to the TIP3P solvent. Recommended values for other decomposition schemes can be found in Tables 1-3 of [106]. See the discussion in `SPROB` above.
- `rhov_effect` Effective water density used in the non-polar dispersion term calculation, default to 1.000. The recommended value is 1.129 for `DECOMPOPT = 2`, the  $\sigma$  scheme. This was optimized in [106] with respect to the TIP3P solvent in PME. Optimized values for other decomposition schemes can be found in Table 4 of [106].
- `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 SASA to be backward compatible with previous releases in *Amber*. Recent study shows that the molecular volume approach transfers better from small training molecules to biomacromolecules (Tan and Luo, In Preparation).
- `= 0` Use SASA to estimate cavity free energy. Default.
  - `= 1` Use the molecular volume enclosed by SASA.
- `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 be backward compatible with previous releases in *Amber*, but not for `INP = 1`. The recommended value is 0.0378 when `DECOMPOPT = 2`, `USE_RMIN = 1`, and `USE_SAV = 1`. See recommended values in Tables 1-3 for other combinations of options.
- `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 be backward compatible with previous releases in *Amber*, but not for `INP = 1`. The recommended value is -0.5692 when `DECOMPOPT = 2`, `USE_RMIN = 1`, and `USE_SAV = 1`. See recommended values in Tables 1-3 for other combinations of options.
- `maxsph` *pbsa* uses a numerical method to compute solvent accessible surface area.[106] `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.

## 7.3 Example inputs

### 7.3.1 Single-point calculation of solvation free energies

Here is a sample input file that might be used to perform single structure calculations.

```
Sample single point PB calculation
&cntrl
ntx=1,
imin=1,
igb=10,
inp=2
/
&pb
npbverb=1, istrng=0, epsout=80.0, epsin=1.0,
radiopt=1, dprob=1.6
space=0.5, nbuffer=0, fillratio=4.0,
accept=0.001,
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
/
```

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 set 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 sample input options above for the NP calculation are set to the recommended values for the  $\sigma$  decomposition scheme and to use molecular volume to correlate with cavity free energy. You can find recommended values for other decomposition schemes

## 7 PBSA

and other options in Tables 1-4 of [106]. If INP is set to 1, the ECAVITY term would give the total non-polar solvation free energy.

### 7.3.2 Visualization of electrostatic potentials

You can also use *pbsa* to 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
ntx=1,
imin=1,
ipb=1,
inp=0
/
&pb
npbverb=1, istrng=0, epsout=80.0, epsin=1.0,
space=1., accept=0.001,
sprob=1.4, cutnb=9,
phiout=1, phiform=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 FD grid is also set to be cubic as in *Delphi*. The `SPROB` value should be set to that used in *PyMol*, 1.4 Å. A large grid spacing, e.g. 1 Å or higher, is recommended for visualization purposes. Otherwise, the potential file would be very large.

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.

In principle, it is possible to visualize the potential file in *VMD*, but we have not validated this program. More detailed information on static single-point PB calculations can be found on online *Amber* tutorial pages.

### 7.3.3 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
imin=1,
ntx=1,
ipb=2,
inp=0
/
&pb
npbverb=1,
istrng=0, epsout=80.0, epsin=1.0, sprob=0.6, radiopt=1,
space=0.5, fillratio=2.0, accept=0.001,
eneopt=2, cut=0, frcopt=2
/
```

Note that INP is set to 0 to turn off non-polar solvation interactions. The molecular surface computed with the level set function is used. ENEOPT and FRCOPT are both set to 2, *i.e.* induced surface charges are used to compute the electrostatic energy and forces. Since CUTNB is set to the default value of zero, an infinite cutoff distance is used for both Coulombic and van der Waals interactions.

### 7.3.4 Comparing with Delphi results

Under identical condition, PBSA is highly consistent with Delphi in term of computed re-action field energies. In this subsection, we briefly go over the details on how you can obtain comparable energies from both program. 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
```

## 7 PBSA

```
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 ntx=1, imin=1, ipb=1, inp=0
/
&pb npbverb=0, istrng=150, epsout=80.0, epsin=1.0,
ivalence=1, iprob=2.0, space=0.5, accept=1e-3,
dprob=1.5, radiopt=0, fillratio=2, bcopt=6,
smoothopt=2, nfocus=1, dbfopt=1, cutnb=0,
maxitn=10000
/
```

The above sample input files are also provided in the current release under \$AMBERHOME/test/pbsa\_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 Angstrom, i.e. *scale*=2.0, while in PBSA the grid spacing is set straight as *space*=0.5 in Angstrom. 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 *bndcnd*=4 and PBSA sets the boundary condition as *bcopt*=6; 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 7.1.3); *arcrs* is used to set the resolution of numerical dot representation of the solvent accessible arcs, the default is 1/16 Å, which is similar to the numerical resolution of solvent accessible arc dots used in Delphi.

## 7.4 PBSA in SANDER

All PBSA functionalities are available in SANDER and all input options are exactly the same as in the standalone PBSA. Apparent exceptions are *ipb* and *inp*: you need to really set *ipb*/*inp* to nonzero in order to invoke PBSA functionalities.

## 7.5 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 14.1 for the list of available

PBSA input options. The structures and parameters are supplied by NAB's facility. An extra feature for PBSA in NAB is the availability of electrostatic forces or gradients. We describe special consideration in its applications in the following section.

### 7.5.1 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. [117] 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, there is a partial solution in the mapping of dielectric boundary force as described by Gilson et al[117] when the SES definition is used. This is currently adopted in the PBSA module. Note that the method is rigorously correct only for two overlapping atoms. On solvent-exposed surface with well-packed and highly charged atoms, such as nucleic acid backbones, the error in the force-mapping procedure can be very large. Our tests show that the deviations from the numerical finite-difference forces can be in the orders of several tenths of kcal/mol-Å. In other situations, the errors are in the orders of several hundredths of kcal/mol-Å.

### 7.5.2 Example

Here is a sample of calls in a NAB program to the *mm\_options()* routine, in order to run pbsa:

```
mm_options("ntpr=1, cut=99.0"); // No solute-solute cutoff
mm_options("ipb=1"); // Use PBSA
mm_options("accept=0.000001"); // Convergence criterion
mm_options("sprob=1.6"); // Solvent probe radius for SASA
mm_options("radiopt=1"); // Useatom-type/charge-based radii
mm_options("fillratio=4"); // Coarse/Fine ratio of electrostatic focusing
```



# 8 Reference Interaction Site Model of Molecular Solvation

In addition to explicit and implicit solvation models, Amber also has a third class of solvation model for molecular mechanics simulations, the reference interaction site model (RISM) of molecular solvation[118–131]. RISM is available in AmberTools as `rism1d` (1D-RISM) and an option in NAB (3D-RISM). In Amber, 3D-RISM is an option in `sander.RISM` and `sander.RISM.MPI`. Details specific to using `sander.RISM` and `sander.RISM.MPI` can be found in the Amber manual.

## 8.1 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[119, 128, 129, 132, 133]

$$h(r_{12}, \Omega_1, \Omega_2) = c(r_{12}, \Omega_1, \Omega_2) + \rho \int d\mathbf{r}_3 d\Omega_3 c(r_{13}, \Omega_1, \Omega_3) h(r_{32}, \Omega_3, \Omega_2), \quad (8.1)$$

where  $r_{12}$  is the separation between particles 1 and 2 while  $\Omega_1$  and  $\Omega_2$  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  $a$ . Orientationally averaging over site  $\alpha$  of species  $a$  and site  $\gamma$  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 (8.1), in the limit of infinite dilution, as a set of

three equations:

$$h^{VV}(r_{12}, \Omega_1, \Omega_2) = c^{VV}(r_{12}, \Omega_1, \Omega_2) + \rho^V \int d\mathbf{r}_3 d\Omega_3 c^{VV}(r_{13}, \Omega_1, \Omega_3) h^{VV}(r_{32}, \Omega_3, \Omega_2), \quad (8.2)$$

$$h^{UV}(r_{12}, \Omega_1, \Omega_2) = c^{UV}(r_{12}, \Omega_1, \Omega_2) + \rho^V \int d\mathbf{r}_3 d\Omega_3 c^{UV}(r_{13}, \Omega_1, \Omega_3) h^{VV}(r_{32}, \Omega_3, \Omega_2), \quad (8.3)$$

$$h^{UU}(r_{12}, \Omega_1, \Omega_2) = c^{UU}(r_{12}, \Omega_1, \Omega_2) + \rho^V \int d\mathbf{r}_3 d\Omega_3 c^{UV}(r_{13}, \Omega_1, \Omega_3) h^{UV}(r_{32}, \Omega_3, \Omega_2). \quad (8.4)$$

Equation (8.3) 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 (8.3) can be obtained using 3D-RISM. However, a solution to Equation (8.2) 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 [132]

$$g(r_{12}, \Omega_1, \Omega_2) = \exp[-\beta u(r_{12}, \Omega_1, \Omega_2) + h(r_{12}, \Omega_1, \Omega_2) - c(r_{12}, \Omega_1, \Omega_2) + b(r_{12}, \Omega_1, \Omega_2)] \quad (8.5)$$

$u$  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  $h$  [132]). 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 two are implemented in 3D-RISM: hypernetted-chain approximation (HNC) and Kovalenko-Hirata (KH).

In the case of HNC  $b = 0$ , giving [132]

$$g(r_{12}, \Omega_1, \Omega_2) = \exp[-\beta u(r_{12}, \Omega_1, \Omega_2) + h(r_{12}, \Omega_1, \Omega_2) - c(r_{12}, \Omega_1, \Omega_2)] \quad (8.6)$$

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 [129].

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 [128, 129]

$$g(r_{12}, \Omega_1, \Omega_2) = \begin{cases} \exp(-\beta u(r_{12}, \Omega_1, \Omega_2) + h(r_{12}, \Omega_1, \Omega_2) - c(r_{12}, \Omega_1, \Omega_2)) & \text{for } g(r_{12}, \Omega_1, \Omega_2) \leq 1 \\ 1 - \beta u(r_{12}, \Omega_1, \Omega_2) + h(r_{12}, \Omega_1, \Omega_2) - c(r_{12}, \Omega_1, \Omega_2) & \text{for } g(r_{12}, \Omega_1, \Omega_2) > 1 \end{cases} \quad (8.7)$$

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 does converge to stable solutions for all stable parts of the phase diagram. The reliability of the KH closure makes it particularly suitable for molecular mechanics calculations.

### 8.1.1 1D-RISM

1D-RISM is used to calculate bulk properties of the solvent and is a prerequisite for 3D-RISM, where 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 via the fundamental RISM approximation[119–122, 132, 133], which produces the intramolecular pair correlation matrix,

$$\omega_{\alpha\gamma}(k) = \sin(kr_{\alpha\gamma})/(kr_{\alpha\gamma}) \quad (8.8)$$

where  $\alpha$  and  $\gamma$  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,  $\text{C}_3\text{H}_8$ , may have two carbon types and two hydrogen types. Equation (8.2) then becomes

$$\begin{aligned} h_{\alpha\gamma}(r) &= \sum_{\mu\nu} \int d\mathbf{r}' d\mathbf{r}'' \omega_{\alpha\mu}(|r-r'|) c_{\mu\nu}(|r'-r''|) [\omega_{\nu\gamma}(r'') + \rho_\nu h_{\nu\gamma}(r'')] \\ &= \frac{1}{(2\pi)^3} \int e^{i\mathbf{k}\cdot\mathbf{r}} d\mathbf{k} \left[ \omega\mathbf{c}[\mathbf{1} - \rho\omega\mathbf{c}]^{-1} \omega \right]_{\alpha\gamma} \\ &= \sum_0^\infty \omega(k)\mathbf{c}(k)\omega(k) [\rho\mathbf{c}(k)\omega(k)]^n. \end{aligned} \quad (8.9)$$

Equation (8.9) is complemented with either the 1D-HNC closure, a one-dimensional, site-site version of Equation (8.6),

$$g_{\alpha\gamma}(r) = \exp[-\beta u_{\alpha\gamma}(r) + h_{\alpha\gamma}(r) - c_{\alpha\gamma}(r)], \quad (8.10)$$

or the 1D-KH closure, a one-dimensional version of Equation (8.7) [129],

$$g_{\alpha\gamma}(r) = \begin{cases} \exp(-\beta u_{\alpha\gamma}(r) + h_{\alpha\gamma}(r) - c_{\alpha\gamma}(r)) & \text{for } g_{\alpha\gamma}(r) \leq 1 \\ 1 - \beta u_{\alpha\gamma}(r) + h_{\alpha\gamma}(r) - c_{\alpha\gamma}(r) & \text{for } g_{\alpha\gamma}(r) > 1 \end{cases}. \quad (8.11)$$

Equations (8.9)-(8.11) are 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 (8.8) set equal to zero for sites  $\alpha, \gamma$  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 [134].

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) [129, 135]. All correlation functions are represented as one-dimensional grids and the convolution integrals in Equation (8.9) 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 [129–131].

### 8.1.2 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 (8.3) becomes [127, 128]

$$h_{\gamma}^{\text{UV}}(\mathbf{r}) = \sum_{\alpha} \int d\mathbf{r}' c_{\alpha}^{\text{UV}}(\mathbf{r} - \mathbf{r}') \chi_{\alpha\gamma}^{\text{VV}}(r'), \quad (8.12)$$

where  $\chi_{\alpha\gamma}^{\text{VV}}(R)$  is the site-site susceptibility of the solvent, obtained from 1D-RISM and given by

$$\chi_{\alpha\gamma}^{\text{VV}}(r) = \omega_{\alpha\gamma}^{\text{VV}}(r) + \rho_{\alpha} h_{\alpha\gamma}^{\text{VV}}(r).$$

The 3D-KH closure is constructed using the same KH approximation but for the 3D site correlation functions [128, 129],

$$g_{\gamma}^{\text{UV}}(\mathbf{r}) = \begin{cases} \exp\left(-\beta u_{\gamma}^{\text{UV}}(\mathbf{r}) + h_{\gamma}^{\text{UV}}(\mathbf{r}) - c_{\gamma}^{\text{UV}}(\mathbf{r})\right) & \text{for } g_{\gamma}^{\text{UV}}(\mathbf{r}) \leq 1 \\ 1 - \beta u_{\gamma}^{\text{UV}}(\mathbf{r}) + h_{\gamma}^{\text{UV}}(\mathbf{r}) - c_{\gamma}^{\text{UV}}(\mathbf{r}) & \text{for } g_{\gamma}^{\text{UV}}(\mathbf{r}) > 1 \end{cases}. \quad (8.13)$$

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 [129, 130, 136]. 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 [136]. The convolution of the short-range terms in the integral equation (8.12) is calculated using 3D fast Fourier transform [137, 138]. Accordingly, the electrostatic asymptotics terms in the thermodynamics integral (8.14) below are handled analytically and reduced to one-dimensional integrals easy to compute [136].

With a converged 3D-RISM solution for  $h^{\text{UV}}$  and  $c^{\text{UV}}$  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),  $\Delta\mu^{\text{ex}}$  and the mean solvation force,  $\mathbf{f}_i^{\text{UV}}(\mathbf{R}_i)$ , on each solute atom,  $i$ . For both the HNC and KH closures, the thermodynamic integration of solvation free energy can be performed analytically. The KH closure gives

$$\Delta\mu^{\text{ex}} = k_{\text{B}}T \sum_{\alpha} \rho_{\alpha}^{\text{V}} \int d\mathbf{r} \left[ \frac{1}{2} (h_{\alpha}^{\text{UV}}(\mathbf{r}))^2 \Theta(-h_{\alpha}^{\text{UV}}(\mathbf{r})) - c_{\alpha}^{\text{UV}}(\mathbf{r}) - \frac{1}{2} h_{\alpha}^{\text{UV}}(\mathbf{r}) c_{\alpha}^{\text{UV}}(\mathbf{r}) \right] \quad (8.14)$$

while for HNC the Heaviside function is replaced with 1 for all  $\mathbf{r}$  [128, 129]. The force equation

$$\mathbf{f}_i^{\text{UV}}(\mathbf{R}_i) = -\frac{\partial \Delta\mu^{\text{ex}}}{\partial \mathbf{R}_i} = -\sum_{\alpha} \rho_{\alpha} \int d\mathbf{r} g_{\alpha}^{\text{UV}}(\mathbf{r}) \frac{\partial u_{\alpha}^{\text{UV}}(\mathbf{r} - \mathbf{R}_i)}{\partial \mathbf{R}_i}$$

is valid for both the HNC and KH closure [118, 139, 140].

Equation(8.14) gives the total solvation free energy,  $\Delta G_{\text{sol}}$ , but it is often useful to decompose this into electrostatic (solvent polarization),  $\Delta G_{\text{pol}}$ , and non-electrostatic (dispersion and cavity formation),  $(\Delta G_{\text{cav}} + \Delta G_{\text{dis}})$ , 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,

$$\Delta\mu^{\text{ex}} = \Delta G_{\text{sol}} = \Delta G_{\text{pol}} + \Delta G_{\text{cav}} + \Delta G_{\text{dis}}.$$

$\Delta G_{\text{sol}}$  is produced by a 3D-RISM calculation on the charged solute.  $\Delta G_{\text{pol}}$  is then the difference of the two calculations. Note that GB and PB methods calculate only  $\Delta G_{\text{pol}}$ .

## 8.2 Practical Considerations

### 8.2.1 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. The memory required can be estimated as

$$\text{Total memory} \approx N_{\text{box}} \left[ \underbrace{4}_{\text{asymptotics}} + N^{\text{V}} \left\{ \underbrace{2N_{\text{MDIIS}}}_{c,\text{residual}} + \underbrace{2}_{g,h} \right\} \right] \times 8 \text{ bytes} \quad (8.15)$$

where  $N_{\text{box}} = N_x \times N_y \times N_z$  is the total number of grid points,  $N^{\text{V}}$  is the number of solvent atom species and  $N_{\text{MDIIS}}$  is the number of MDIIS vectors used to accelerate convergence. A full grid for  $g$  and  $h$  is required for each solvent species and four grids are required to compute the long range asymptotics. Memory, therefore, scales linearly with  $N_{\text{box}}$  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 step methods (see the Amber manual) are applicable to molecular dynamics only while parallelization is limited by system size and computational resources.

Both `sander.RISM.MPI` and `NAB` have MPI implementations of 3D-RISM (see Section 8.5.4 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 2.1.5 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 ( $\geq 7$ ).

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.RISM.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.

### 8.2.2 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 OpenDX file format (Section 8.6.3) 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 capable of displaying both molecular and volumetric trajectories.

### 8.2.3 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[118], minimization and trajectory post-processing[141] 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.

## 8.3 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,  $\chi^{VV}$ , and properties such as the temperature and partial charges. This is read in as a `.xvv` file, which is produced by a 1D-RISM calculation. If another 3D-RISM calculation is to be performed 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/rismld`.

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 8.6.1 for format details.

## 8.4 *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.

### 8.4.1 Parameters keywords

There are no default parameters for `rism1d`. All of the following parameters should be specified in the input file.

#### 8.4.1.1 Theory

`theory`    [] The 1D-RISM theory to use.

**DRISM** Dielectrically consistent RISM (recommended).

**XRISM** Extended RISM.

`closur`    [] The type of closure to use.

**KH** Kovalenko-Hirata (recommended).

**HNC** Hyper-netted chain equation.

**PY** Percus-Yevick.

**VM** Verlet modified.

**VM0** Verlet modified.

#### 8.4.1.2 Grid Size

`dr`        [] Grid spacing in real space. 0.025 Å is recommended.

`nr`        [] Number of grid points. Should be a product of small prime factors (2, 3 and 5). 16384 is recommended.

### 8.4.1.3 Output

- outlst     [] Indicates what output files to produce. 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.
  - X**  $\chi^{VV}(k)$  Solvent site-site susceptibility in reciprocal space. Required input for 3D-RISM.
  - G**  $G^{VV}(r)$  Solvent site-site pair distribution function in real-space.
  - B**  $B^{VV}(r)$  Solvent site-site bridge correction in real space for VM and VM0 closures.
  - T** Thermodynamic properties of the solvent.
  - E**  $exN^{VV}(r)$  Solvent site-site running excess coordination numbers in real space.
  - N**  $N^{VV}(r)$  Solvent site-site excess coordination numbers in real space.
  - S**  $S^{VV}(k)$  Solvent site-site structure factor in reciprocal space.
- routup     [] Largest real space separation in Å for output files. If 0 then all grid points will be output.
- toutup     [] Largest reciprocal space separation in Å<sup>-1</sup> for output files. If 0 then all grid points will be output.
- ksave      [] Output an intermediate solution every *ksave* steps. If *ksave* ≤ 0 then no intermediate restart files are written. If any restart files are present at run time (*.sav* suffix) they are automatically used. However, such files are non-portable binary files.
- kshow      [] Write the current residue to standard output every *kshow* iteration. If *kshow* ≤ 0 then residue is not reported.

### 8.4.1.4 Species keywords

For each molecular species in the solvent mixture, a *species* name list should be provided. There are only two parameters for this name list and no default values.

- density    [] Density of the species in M.
- model      [] Relative or absolute path to and name of the *.mdl* file with the parameters for this solvent molecule.

### 8.4.1.5 Solution Convergence

- nis        [] Number of MDIIS vectors to use. 20 is recommended.
- delvv      [] MDIIS step size. 0.3 is recommended.
- tolvv      [] Target residual tolerance for the self-consistent solution. 10<sup>-12</sup> is recommended.
- maxste     [] Maximum number of iterations to converge to a solution. 10000 is reasonable.

**8.4.1.6 Solvent Description**

- temper     [] Temperature in Kelvin.
- dieps      [] Dielectric constant of the solvent.
- nsp        [] Number of species (molecules) in the solutions. Also indicates the number of  
species name lists to follow.

**8.4.1.7 Other**

- smear      [] Electrostatic smear parameter. 1.0 is recommended.
- adbcor     [] 0.5 is recommended.

**8.4.2 Example**

Mixed ionic solvent.

```

&PARAMETERS
  THEORY='DRISM', CLOSUR='KH',           !Theory
  NR=16384, DR=0.025,                   !Grid size and spacing
  OUTLST='x', routup=384, toutup=0,    !Output
  NIS=20, DELVV=0.3, TOLVV=1.e-12,     !MDIIS
  KSAVE=-1,                             !Check pointing
  KSHOW=1,                              !Output frequency
  maxste=10000,                         !Maximum iterations
  SMEAR=1, ADCOR=0.5,                  !Electrostatics
  TEMPER=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/rism1d/model/SPC.mdl"
/
&SPECIES
  !Sodium
  DENSITY=.1d0,
  MODEL="../../../../dat/rism1d/model/Na+.mdl"
/
&SPECIES
  !Chloride
  DENSITY=.1d0,
  MODEL="../../../../dat/rism1d/model/Cl-.mdl"
/

```

## 8.5 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 8.5.4. At this time, standard molecular dynamics and minimization with non-polarizable force fields is supported.

### 8.5.1 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 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 molecular dynamics or minimization calculation.

### 8.5.2 I/O

All 3D-RISM options, including input and output files, are specified using `mm_options()` (see Section 14.1). 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. 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`.

### 8.5.3 Examples

#### 8.5.3.1 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
:

```

#### 8.5.3.2 Minimization

```

:
mm_options("ntpr=1, cut=999.0"); //No solute-solute

```

```

mm_options("rism=1"); //cut off
mm_options("xvvhfile=../rismld/spc/spc.xvv.save"); //Use 3D-RISM-KH
mm_options("tolerance=1e-11"); //1D-RISM input
mm_options("solvcut=999.0"); //Low tolerance
mm_options("centering=2"); //No solute-solvent
//cut off
//Center solute
//using center-
//of-geometry
:

```

### 8.5.4 Compiling MPI 3D-RISM

Executables compiled with MPI NAB 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 NAB. The procedure to include 3D-RISM in MPI NAB is a simple two step process.

1. 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. For OpenMPI and MPICH2 these files are `libmpi_f77.a` and `libfmpich.a` respectively (the suffix may vary). Set the `XTRA_FLIBS` environment variable to contain the compiler directive to link the library. For example:

```

OpenMPI  export XTRA_FLIBS=-lmpi_f77
MPICH2   export XTRA_FLIBS=-lfmpich

```

2. Run `configure` and specify both `-mpi` and `-rismmpi`. For example:

```

./configure -mpi -rismmpi gnu

```

## 8.6 File Formats

### 8.6.1 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.

## 8 Reference Interaction Site Model of Molecular Solvation

**%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 ( $e$ ).

**%FLAG LJEPSILON**  
**%FORMAT (5e16.8)**

REAL\*8 (NSITE)    Lennard-Jones  $\epsilon$  for each solvent site (kcal/mol).

**%FLAG LJSIGMA**  
**%FORMAT (5e16.8)**

REAL\*8 (NSITE)    Lennard-Jones  $r_{\min}(\sigma^*)$  for each solvent site (Å). Note that this is not  $\sigma$ .

**%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 (Å).

## 8.6.2 XVV

Solvent site-site susceptibility,  $\chi^{VV}$ , 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.

```
%VERSION VERSION_STAMP = Vxxxx.yyy DATE = mm:dd:yy hh:mm:ss
```

The current version of the format is 0000.001. 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  $\chi_{ab}^{VV}(k)$ .

NSITE        Number of solvent sites.

NSPECIES    Number of solvent species (molecules).

```
%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 RHOV
%FORMAT (5E16.8)
```

REAL\*8 (NSITE) Number density of each solvent site (Å<sup>-3</sup>).

```
%FLAG QV
%FORMAT (5E16.8)
```

REAL\*8 (NSITE) Partial charge for each solvent site (*e*).

## 8 Reference Interaction Site Model of Molecular Solvation

**%FLAG QSPV**

**%FORMAT (5E16.8)**

REAL\*8 (NSPECIES) Net charge for each solvent species ( $e$ ).

**%FLAG EPSV**

**%FORMAT (5E16.8)**

REAL\*8 (NSITE) Lennard-Jones  $\epsilon$  for each solvent site (kcal/mol).

**%FLAG SIGV**

**%FORMAT (5E16.8)**

REAL\*8 (NSITE) Lennard-Jones  $r_{\min}$  ( $\sigma^*$ ) for each solvent site ( $\text{\AA}$ ). Note that this is not  $\sigma$ .

**%FLAG DELHVO**

**%FORMAT (5E16.8)**

REAL\*8 (NSITE) Long range Coulomb correction for each solvent site.

**%FLAG XVV**

**%FORMAT (5E16.8)**

REAL\*8 (NR, NSITE, NSITE)  $\chi_{ab}^{VV}(k)$ . This array is stored in column major order. I.e., the NR index varies fastest.

### 8.6.3 OpenDX

3D correlation functions from 3D-RISM calculations use the ASCII version of the OpenDX file format for volumetric data on regular grids as defined in the OpenDX 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

```

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 Miscellaneous utilities

### 9.1 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

## 9 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.[142, 143] 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 parameterization derived for RNA structures. Details about the RNA parameterization can be found in ref.[144] .

*-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 ID’s 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 hard-wired into the code, requiring a re-compilation 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.

## 9.2 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.

### 9.2.1 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, NH<sub>3</sub><sup>+</sup>, Met-CH<sub>3</sub>, 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 NH<sub>3</sub><sup>+</sup> 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. Non-standard atom names are the primary cause of missing or misplaced

hydrogens. If reduce tries to process a file which contains hydrogens with non-standard 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.

### 9.2.2 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

```

### 9.2.3 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).

### 9.2.4 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 sub-cliques or singletons.

An alternative way to break up cliques is to rotate all the methionine CH<sub>3</sub>s and lysine/N-terminus NH<sub>3</sub><sup>+</sup>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 NH<sub>3</sub><sup>+</sup> 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.

### 9.2.5 Contact

If you use reduce, I would appreciate any comments you send my way.

J. Michael Word

e-mail: [mike.word@duke.edu](mailto:mike.word@duke.edu) voice: (919)483-3522

Richardson Lab, Biochemistry Department, Duke University, Durham, NC USA 27710

## 9.3 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. [145] 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.



## 10 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.

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.

## 10.1 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.

### 10.1.1 Conformation build-up procedures

MC-SYM[146–148] 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.*[149] 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  $\chi$ . 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[150, 151] 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.

### 10.1.2 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.[152] 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,[153] which decomposes

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.*[154] 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[155] 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.

## 10.2 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.

### 10.2.1 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  $4 \times 4$  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.

## 10.2.2 Distance geometry

nab’s second initial structure-creation method is *metric matrix distance geometry*,<sup>[156, 157]</sup> 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.

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[158] 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.

### 10.2.3 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,[159] calyculin dimers,[160] HMG-protein/DNA complexes,[161] active sites of Rieske iron-sulfur proteins,[162] and supercoiled DNA.[163] The Examples chapter below provides a number of other sample applications.

## 10.3 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.

## 10.4 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.

## 10.5 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.

### 10.5.1 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'-gcgtaacgc -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.

## 10.5.2 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.

### 10.5.3 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 10.1. 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

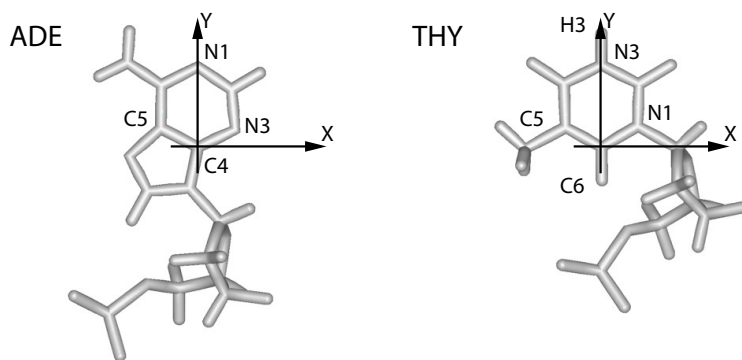


Figure 10.1: *ADE and THY after execution of Program 3.*

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.

## 10.6 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.

## 10.7 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 addressidue( 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 end1,
             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  addressidue( 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.

## 10.8 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()` 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 map1to3[ hashed ];           // convert residue names
6
7     map1to3["A"] = "ADE";      map1to3["C"] = "CYT";
8     map1to3["G"] = "GUA";      map1to3["T"] = "THY";
9     map1to3["U"] = "URA";
10
11    map1to3["a"] = "ADE";      map1to3["c"] = "CYT";
12    map1to3["g"] = "GUA";      map1to3["t"] = "THY";
13    map1to3["u"] = "URA";
14
15    if( r in map1to3 ) {
16        res = getresidue( map1to3[ 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.

## 10.9 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 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

## 10 NAB: Introduction

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. *i.e.*, 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.

## 10.10 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 `addressidue()` 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  $\leq$  cutoff writes the information out using the C-like I/O function *printf()*.

## 10.11 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.

### 10.11.1 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.

### 10.11.2 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 a 0 on success and 1 on failure. In both functions, the special atom expression NULL selects all atoms in the input residue or molecule.

### 10.11.3 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,
             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.

## 10.12 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 3.13. 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 3 at this point.

### 10.12.1 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 3.13.

### 10.12.2 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 `rtl` 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.

### 10.12.3 `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

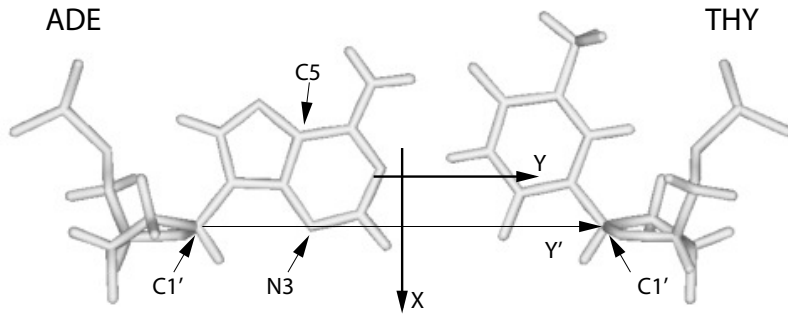
the "anti" strand. `seq` and `aseq` are required to be complementary although this is not checked. `wc_helix()` creates the molecule one base pair at a time. `seq` is read from left to right, `aseq` is read from right to left and corresponding letters are extracted and converted to residues by `getres()`. 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.[164]

`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.

#### 10.12.4 `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, `addressidue()` adds `sres` to the strand `sense` of `m_sense`. In line 84, `addressidue()` adds `ares` to the strand `anti` of `m_anti`. Lines 86 and 87 align the molecules containing the sense residue and anti residue so that `sres` and `ares` are on top of each other. Line 88 creates a transformation matrix that rotates `m_anti` (containing `ares`) 180° 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

Figure 10.2: *ADE:THY* from *wc\_basepair()*.

helix generation. Initially the base as shown in Figure 10.2 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( ( sname == "CYT" ) || ( sname =~ "[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( ( sname == "GUA" ) || ( sname =~ "[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( ( sname == "THY" ) || ( sname =~ "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( ( sname == "URA" ) || ( sname =~ "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",sname );
60         exit( 1 );
61     }
62     addressidue( 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     addressidue( 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 };

```

### 10.12.5 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 be 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 residue of the sense strand of `m2`. The residue of `m2` is added after the "last" residue 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 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;
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;

```

## 10 NAB: Introduction

```
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     srname = "D" + loup[ substr( seq, i, 1 ) ];
68     setreslibkind( sreslib, snatype );
69     sres = getresidue( srname, sreslib_use );
70     arname = "D" + loup[ substr( aseq, i, 1 ) ];
71     setreslibkind( areslib, anatype );
```

```

72     ares = getresidue( arname, 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     srname = substr( seq, i, 1 );
93     srname = "D" + loup[ substr( seq, i, 1 ) ];
94     setreslibkind( sreslib, snatype );
95     if( opts =~ "s3" )
96         sres = getres( srname + "3", sreslib_use );
97     else
98         sres = getres( srname, sreslib_use );
99     arname = "D" + loup[ substr( aseq, i, 1 ) ];
100    setreslibkind( areslib, anatype );
101    if( opts =~ "a5" )
102        ares = getres( arname + "5", areslib_use );
103    else
104        ares = getres( arname, 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 };

```

## 10.13 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. The text version of *LEaP*, called *tleap* is distributed as a part of *NAB*.

### 10.13.1 Creating a Parallel DNA Triplex

Parallel DNA triplexes are thought to be intermediates in homologous DNA recombination. These triplexes, investigated by Zhurkin *et al.*[165] 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.

### 10.13.2 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

```

## 10 NAB: Introduction

```
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 addressidue( 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         fprintf( ef, "%10.3f %10.3f %10.3f %10.3fn",
67                 x, y, brz, be );
68     }
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.

### 10.13.3 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 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.

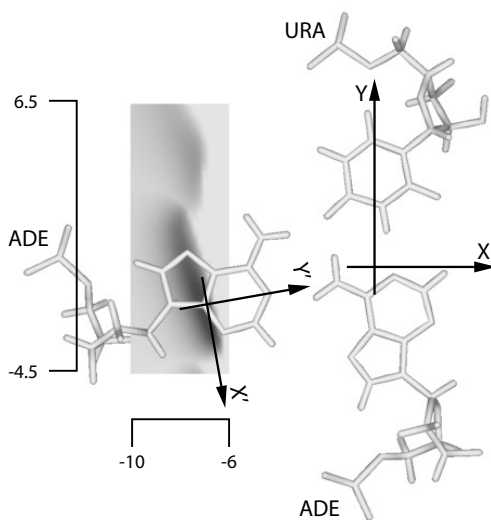


Figure 10.3: *Minimum energy AUA triad and the potential energy surface.*

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 10.3 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 10.3 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 10.3 shows the best AU:A found.

#### 10.13.4 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 );
27        setpoint( m, "C:THY:C1'", p3 );
28    }
29    circle( p1, p2, p3, pc );

```

## 10 NAB: Introduction

```
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 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 forcefield 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 terms of forcefield—things like 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.



# 11 NAB: Language Reference

## 11.1 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.

## 11.2 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.

### 11.2.1 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
```

### 11.2.2 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

### 11.2.3 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.

### 11.2.4 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 <i>or</i> atom in molecule	4	
&&	and	3	
	or	2	
=	assignment	1	right to left

### 11.2.5 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.

## 11.3 Higher-level constructs

### 11.3.1 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 makes 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`.

### 11.3.2 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 <b>Residue attributes</b> 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.

### 11.3.3 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 ];
....
deallocate 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.

### 11.3.4 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.

### 11.3.5 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.

### 11.3.6 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.

### 11.3.7 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 <code>%</code>

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.

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 to 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.

<b>Ouput format options</b>	
<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 &amp; 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.

## 11.4 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.

### 11.4.1 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.
```

### 11.4.2 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.

### 11.4.3 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;
```

### 11.4.4 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;
}
```

### 11.4.5 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.

### 11.4.6 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  
    ...  
}
```

### 11.4.7 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  
    ...  
}
```

### 11.4.8 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.
```

### 11.4.9 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.

## 11.5 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 othe form of a struct declaration is this short form:

```
struct cmplx_t cv1, cv2;
```

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, ctab[ 10 ]; // define some vars
int f( int s, struct cmplx_t ct[1] ) // func taking array of
                                   // struct cmplx_t { ... };
```

## 11.6 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.

### 11.6.1 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
};

```

### 11.6.2 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;
```

## 11.7 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>	-p	8	Vector negation, same as -1 * p.
^	p^q	7	Compute the cross or vector product of p, q.
@	p@q	6	Compute the scalar or dot product of p, q.
*	s * p	6	Multiply p by s, same as p * s.
/	p / s	6	Divide p by s, s / p not allowed.
+	p + q	5	Vector addition
<i>Binary -</i>	p - q	5	Vector subtraction
==	p == q	4	Test if p and q equal.
!=	p != q	4	Test if p and q are different.
=	p = q	1	Set the value of p to q.

## 11.8 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 t that matches the regular expression r. gsub() replaces all non overlapping substrings of t that match the regular expression r with the string s.

## 11.9 Math Functions

nab provides the following builtin mathematical functions. 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.

<b>nab Builtin Mathematical Functions</b>	
<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 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, $x \geq 0$ .

## 11.10 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.

## 11.11 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.

### 11.11.1 Ordinary I/O Functions

nab provides these functions for stream or FILE \* I/O of int, float and string objects.

```
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.

<b>fopen() mode values</b>	
“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.

### 11.11.2 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) -> (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.

## 11.12 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 addressidue( 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 `addressidue()` is used to add residues to a strand. The first residue is numbered 1 and subsequent residues are numbered 2, 3, etc. `addressidue()` 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.

<b>end1</b>	<b>end2</b>	<b>Action</b>
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> : $1, 2, \dots, N : M, \dots, 2, 1$

## 11.13 Creating Biopolymers

```

molecule linkprot( string strandname, string seq, string reslib );

```

```

molecule link_na( string strandname, string seq, string reslib, string natype,
                    string opts );
molecule getpdb_prm( string pdb-
                    file, string leaprc, string leap_cmd2, int savef )

```

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 *link\_prot()* 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  $\phi$ ,  $\psi$  and  $\omega$  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  $\delta$  position; "h" for histidine with the proton at the  $\epsilon$  position; "3" for protonated histidine; "n" for an acetyl blocking group; "c" for an HNMe blocking group, "a" for an NH<sub>2</sub> 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 *getpdb\_prm()* function described in Chapter 6.

## 11.14 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 fd_helix()	
<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.; Chandrasekharan, 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.

## 11.15 Reduced Representation DNA Modeling Functions

`nab` provides several functions for creating the reduced representation models of DNA described by R. Tan and S. Harvey.[166] 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`.

## 11.16 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()`.

`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.

<b>Options flags for putpdb</b>	
<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>i.e.</i> 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>i.e.</i> the residue numbers are consecutive across chains, as required by some force-field programs like Amber.

putbnd() writes the bonds of mol into fname. 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.

```
num1 rname1 aname1 num2 rname2 aname2 distance
```

## 11.17 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 atom 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.* [167]

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`.

## 11.18 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, 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 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.

## 11.19 Time and date routines

NAB incorporates a few interfaces to time and date routines:

```
string date();  
string timeofday();  
string ftime( string fmt );  
float second();
```

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.

The `second()` routine returns the number of seconds of CPU utilization since the beginning of the process. It is really just a wrapper for the C function `clock()/CLOCKS_PER_SEC`, and so the meaning and precision of the output will depend upon the implementation of the underlying C compiler and libraries. Generally speaking, you should be able to time a certain section of code in the following manner:

```
t1 = second();  
..... // code to be timed  
t2 = second();  
elapsed = t2 - t1
```

## 12 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.

### 12.1 Transformation Matrix Functions

nab uses  $4 \times 4$  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  $4 \times 4$  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.

### 12.2 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).

## 12.3 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.

## 12.4 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.

### 12.4.1 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

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.

### 12.4.2 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  $4 \times 4$  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.

## 12.5 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.

### 12.5.1 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:	$IM_1 \times R_1, IM_2 \times R_1, IM_3 \times R_1, IM_1 \times R_2, IM_2 \times R_2, IM_3 \times R_2$

### 12.5.2 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

## 12 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  $\mathbf{O} \rightarrow \text{axis1}$ ,  $\mathbf{O} \rightarrow \text{axis2}$  and  $\mathbf{O} \rightarrow \text{axis3}$  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  $360/\text{count}$ . 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.

<b>symmetry</b>	<b>name</b>	<b>noid</b>	<b>axestype</b>	<b>center</b>	<b>axes</b>	<b>angles</b>	<b>dist</b>	<b>count</b>
cube	<i>mPid</i>	false	relative	Required	1,2	-	-	-
cyclic	<i>mPid</i>	false	relative	Required	1	-	-	D=1
dihedral	<i>mPid</i>	false	relative	Required	1,2	-	-	D=1
dodeca	<i>mPid</i>	false	relative	Required	1,2	-	-	-
helix	<i>mPid</i>	false	relative	Required	1	1,D=0	D=0	D=1
ico	<i>mPid</i>	false	relative	Required	1,2	-	-	-
octa	<i>mPid</i>	false	relative	Required	1,2	-	-	-
tetra	<i>mPid</i>	false	relative	Required	1,2	-	-	-
<b>transform</b>	<b>name</b>	<b>noid</b>	<b>axestype</b>	<b>center</b>	<b>axes</b>	<b>angles</b>	<b>dist</b>	<b>count</b>
orient	<i>mPid</i>	-	relative	Required	All	All,D=0	-	-
rotate	<i>mPid</i>	-	relative	Required	1	1,D=0	-	-
translate	<i>mPid</i>	-	relative	Required	1	-	D=0	-

### 12.5.3 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.

### 12.5.4 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, fourth and fifth matrices of `5.mat`.

### 12.5.5 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
<code>matextract 2</code>	Extract matrix number 2.
<code>matextract 2,5</code>	Extract matrices number 2 and 5.
<code>matextract 2 5</code>	Extract matrices number 2 and 5.
<code>matextract 2-5</code>	Extract matrices number 2 up to and including 5.
<code>matextract -5</code>	Extract matrices 1 to 5.
<code>matextract 2-</code>	Extract all matrices beginning with number 2.
<code>matextract -</code>	Extract all matrices.
<code>matextract 2-4,7 13 15,19-</code>	Extract matrices 2 to 4, 7, 13, 15 and all matrices numbered 19 or higher.

### 12.5.6 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
<code>transform -pdb X.pdb</code>	Transform a PDB format file.
<code>transform -point X.pts</code>	Transform a set of points.

## 13 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.

### 13.1 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.[156, 168] 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:

$$g_{ij} \equiv \mathbf{x}_i \cdot \mathbf{x}_j \quad (13.1)$$

These matrix elements can be expressed in terms of the distances  $d_{ij}$ :

$$g_{ij} = 2(d_{i0}^2 + d_{j0}^2 - d_{ij}^2) \quad (13.2)$$

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  $\lambda$ , the metric matrix can be written in two ways:

$$g_{ij} = \sum_{k=1}^3 x_{ik}x_{jk} = \sum_{k=1}^3 w_{ik}w_{jk}\lambda_k \quad (13.3)$$

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:

$$x_{ik} = \lambda_k^{1/2} w_{ik} \quad (13.4)$$

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.[169, 170] 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.[158] 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.[171] 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,[172] 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.

## 13.2 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 and-
bounds( bounds b, molecule mol, string aex1, string aex2, float lb, float ub );
int or-
bounds( bounds b, molecule mol, string aex1, string aex2, float lb, float ub );
int set-
bounds( 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,
```

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 13.1: 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 13.1.

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`) `andbounds()` sets the lower bound to  $\max(\text{current\_lb}, \text{lb})$  and the upper bound to the  $\min(\text{current\_ub}, \text{ub})$ . If  $\text{ub} < \text{current\_lb}$  or if  $\text{lb} > \text{current\_ub}$ , 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.

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  $O(N^3)$ . 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>i.e.</i> , <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.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
<code>pembed</code>	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.
<code>shuffle</code>	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.
<code>rbox</code>	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.
<code>riter</code>	1000	Maximum number of cycles for random-embed procedure. Each cycle selects 1000 pairs for adjustment.
<code>slearn</code>	1.0	Starting value for the learning parameter in proximity embedding; see [27] for details.
<code>kchi</code>	1.0	Force constant for enforcement of chirality constraints.
<code>k4d</code>	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 <i>w</i> is the value of the fourth dimensional coordinate.
<code>sqviol</code>	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.
<code>lbpen</code>	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.
<code>ntr</code>	10	Frequency at which the bounds matrix violations will be printed in subsequent refinements.
<code>pencut</code>	-1.0	If <code>pencut</code> $\geq$ 0.0, individual distance and chirality violations greater than <code>pencut</code> will be printed out (along with the total energy) every <code>ntr</code> 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.

### 13.3 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.[173] 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?`T]", get-  
pdb( PATH + "gc.bdna.pdb" ), " : : ??, H? [ `T ]", 0.1 );
```

where `PATH` is `$NABHOME/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 13.1 and 13.2.

13 NAB: Distance Geometry

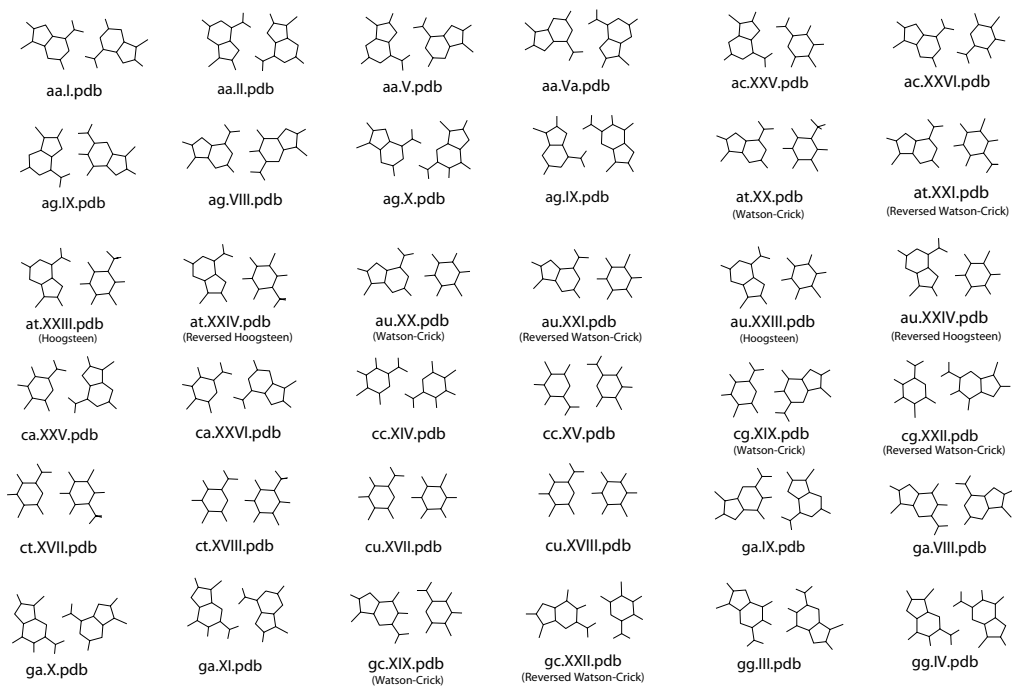


Figure 13.1: *Basepair templates for use with useboundsfrom(), (aa-gg)*

### 13.3 Distance geometry templates

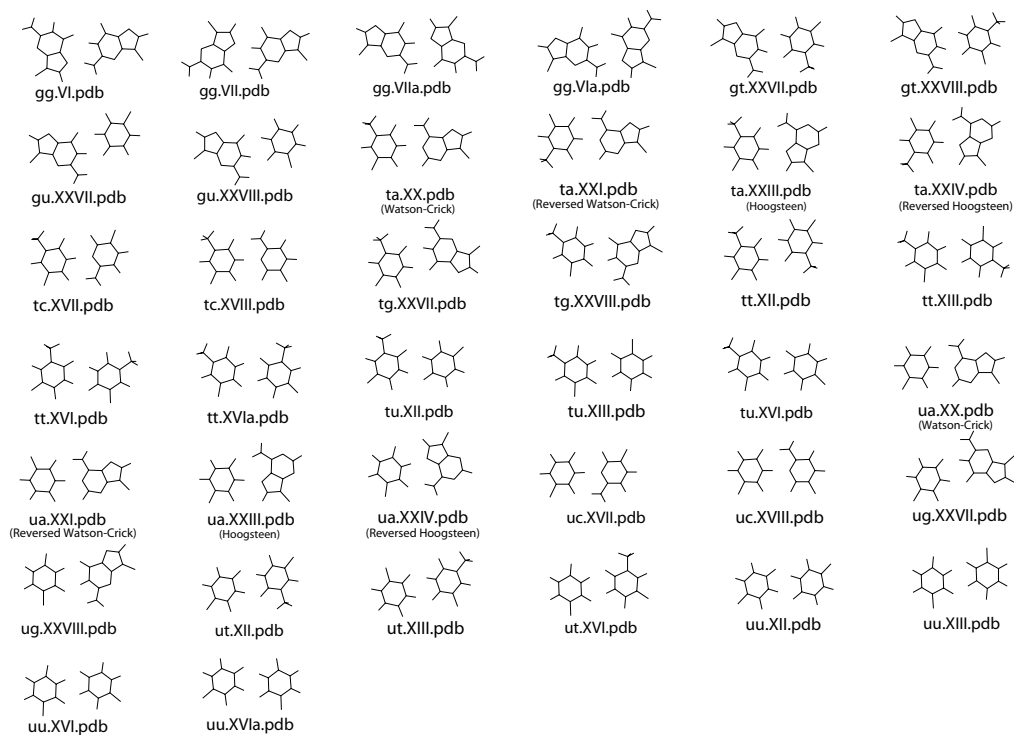


Figure 13.2: Basepair templates for use with `useboundsfrom()`, (gg-uu)

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.

## 13.4 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, \$NABHOME/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 `$NABHOME/coords` directory. The databases are made by issuing the command `$NABHOME/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. i.e., all `bdna` `pdb` files must begin with `bd`.

The `nab` functions used to create the databases are located in `$NABHOME/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.



# 14 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.

## 14.1 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[] );  
int getxyz( string filename, int natom, float xyz[] );  
int putxyz( string filename, int natom, float xyz[] );  
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 \$NABHOME/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 Amber-style restart files that have coordinates and velocities. The `getxyz()` and `putxyz()` routines read and write restart files that have coordinates only (and not velocities). 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 `putxyz()` routine is used in conjunction with the `mm_set_checkpoint()` routine to write checkpoint or restart files. 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.

## 14.1 Basic molecular mechanics routines

<i>keyword</i>	<i>default</i>	<i>meaning</i>
npr	10	Frequency of printing of the energy and its components.
e_debug	0	If non-zero printout additional components of the energy.
gb_debug	0	If non-zero printout information about Born first derivatives.
gb2_debug	0	If non-zero printout information about Born second derivatives.
nchk	10000	Frequency of writing checkpoint file during first derivative calculation, i.e., in the <i>mme()</i> routine.
nchk2	10000	Frequency of writing checkpoint file during second derivative calculation, i.e., in the <i>mme2()</i> routine.
nsnb	25	Frequency at which the non-bonded list is updated.
cut	8.0	Non-bonded cutoff, in angstroms.
wcons	0.0	Restraint weight for keeping atoms close to their positions in <i>xyz_ref</i> (see <i>mme_init</i> ).
dim	3	Number of spatial dimensions; supported values are 3 and 4.
k4d	1.0	Force constant for squeezing out the fourth dimensional coordinate, if <i>dim</i> =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 <i>w</i> is the value of the fourth dimensional coordinate.
dt	0.001	Time step, ps.
t	0.0	Initial time, ps.
rattle	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 <i>mme_rattle()</i> function to <i>conjgrad()</i> .
tautp	999999.	Temperature coupling parameter, in ps. The time constant determines the strength of the weak-coupling ("Berendsen") temperature bath.[174] Set <i>tautp</i> 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 <i>gamma_In</i> remains at its default value of zero; if <i>gamma_In</i> is not zero, Langevin dynamics is assumed, as discussed below.
gamma_In	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.[175] 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.[176]
temp0	300.0	Target temperature, K.
vlimit	20.0	Maximum absolute value of any component of the velocity vector.
npr_md	10	Printing frequency for dynamics information to stdout.
ntwx	0	Frequency for dumping coordinates to <i>traj_file</i> .
zerov	0	If non-zero, then the initial velocities will be set to zero.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
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 <i>md()</i> 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 <i>prmtop</i> 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 <b>85</b> , 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 <i>mme_init()</i> by scaling all of the charges by $\sqrt{\text{dielc}}$ . This means that you need to set this (if desired) in <i>mm_options()</i> before calling <i>mme_init()</i> .
gb	0	If set to 0 then GB is off. Setting <i>gb</i> =1 turns on the Hawkins, Cramer, Truhlar (HCT) form of pairwise generalized Born model for solvation. See ref [66] for details of the implementation; this is equivalent to the <i>igb=1</i> option in Amber. Set <i>diel</i> to "C" if you use this option. Setting <i>gb</i> =2 turns on the Onufriev, Bashford, Case (OBC) variant of GB,[67, 177] 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 <i>gb</i> =5 just changes the values of $\alpha$ , $\beta$ and $\gamma$ to 1.0, 0.8, and 4.85, respectively, corresponding to the <i>igb=5</i> option in <i>sander</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.
gbsa	0	If set to 1, add a surface-area dependent energy equal to <i>surften</i> *SASA, where <i>surften</i> is discussed below, and SASA is an approximate surface area term. NAB uses the "LCPO" approximation developed by Weiser, Shenkin, and Still.[178]
surften	0.005	Surface tension (see <i>gbsa</i> , above) in kcal/mol/Å <sup>2</sup> .
epsxt	78.5	Exterior dielectric for generalized Born; interior dielectric is always 1.
kappa	0.0	Inverse of the Debye-Hueckel length, if <i>gb</i> is turned on, in Å <sup>-1</sup> . 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 $\epsilon=78.5$ , $\kappa = (0.10806I)^{1/2}$ , where <i>I</i> is in [M].

## 14.1 Basic molecular mechanics routines

<i>keyword</i>	<i>default</i>	<i>meaning</i>
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: <b>0</b> , <b>1</b> , and <b>2</b> . See PBSA chapter for more information.
inp	1	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: <b>0</b> , <b>1</b> , and <b>2</b> . 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. i.e. only two dielectric regions are allowed in the current release.
smoothopt	0	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: <b>0</b> , and <b>1</b> . See PBSA chapter for more information.
dprob	0.0	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.
npbopt	0	Option to select the linear or the full nonlinear PB equation. = <b>0</b> Linear PB equation is solved. = <b>1</b> 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: <b>1</b> , <b>2</b> , <b>3</b> , <b>4</b> , <b>5</b> , and <b>6</b> . 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: <b>1</b> and <b>2</b> .
fscale	8	Set the ratio between the coarse and fine grid spacings in an electrostatic focussing calculation.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
bcopt	5	Boundary condition options. This is equivalent to the bcopt option in <i>pbsa</i> . Possible values: <b>1</b> , <b>5</b> , and <b>6</b> . 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: <b>1</b> , and <b>2</b> . See PBSA chapter for more information.
dbfopt	n/a	This keyword is equivalent to ENEOPT, and will be phased out in future releases. = <b>0</b> equivalent to ENEOPT = 1. = <b>1</b> equivalent to ENEOPT = 2.
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: <b>0</b> , <b>1</b> , <b>2</b> , and <b>3</b> . See PBSA chapter for more information.
cutres	99.0	Residue-based cutoff distance. The residue-based non-bonded list is used to make the generation of the atom-based pair list efficient.
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	1.6	Solvent probe radius for solvent accessible surface area (SASA) used to compute the dispersion term.
irism	0	Use 3D-RISM. = <b>0</b> Off. = <b>1</b> On.
xvvfile	n/a	.xvv file which describes bulk solvent properties. Required for 3D-RISM calculations. Produced by <code>rism1d</code> .
guvfile	n/a	Root name for solute-solvent 3D pair distribution function, $G^{UV}(\mathbf{R})$ , output in ASCII OpenDX format. 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, $H^{UV}(\mathbf{R})$ , output in ASCII OpenDX format. 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, $C^{UV}(\mathbf{R})$ , output in ASCII OpenDX format. This will produce one file for each solvent atom type for each frame requested.

## 14.1 Basic molecular mechanics routines

<i>keyword</i>	<i>default</i>	<i>meaning</i>
closure	1	Select closure approximation.  = 0 Hyper-netted chain equation (HNC). = 1 Kovalenko-Hirata (KH).
solvcut	buffer	Cut-off distance for solvent-solute potential and force calculations. If <code>buffer &lt; 0</code> and <code>rism_cut</code> is not explicitly set, <code>rism_cut =  buffer </code> . For minimization it is recommended to not use a cut-off (e.g. <code>solvcut=9999</code> ).  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.
buffer	14	
grdspcX ngX	0.5 n/a	Linear grid spacing in Å. X=x, y or z. E.g., <code>grdspcy=0.5</code> . Sets the number of grid points for a fixed size solvation box. X=x, y or z. E.g., <code>ngy=64</code> .
solvboxX	n/a	Sets the size in Å of the fixed size solvation box. X=x, y or z. E.g., <code>solvboxy=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_vec	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	1	Specify implementation of the MDIIS routine.  = 0 Original. For small systems (e.g. < 64 <sup>3</sup> grid points) this implementation may be faster than the BLAS optimized version.  = 1 BLAS 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. Note that the only format is ASCII OpenDX which does not support multiple grids.</p> <p>= 0 No files written.</p> <p>&gt;= 1 Output every <code>ntwrism</code> time steps.</p>

<i>keyword</i>	<i>default</i>	<i>meaning</i>
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>kshow</code> iterations. 0 indicates this information will not be displayed. Must be used with <code>verbose &gt; 1</code> .
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 blocksize as small as 1 results in better load balancing during parallel execution. For OpenMP, blocksize 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 blocksize is not know, although a value of 1 is probably too small.

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 `npr` 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, send `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 progress. 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.

## 14.2 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, ntr_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 `$NABHOME/test` directory.

## 14.3 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 mme2( float x[], float g[], float h[], float mass[], int iter );
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 hrmx, 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  $3 \times \text{natom}$  vector of coordinates `x[]`, and returns a gradient vector `g[]`, a Hessian matrix, stored columnwise in a  $3 \times \text{natom} \times 3 \times \text{natom}$  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 will generally not 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  $3*natom$ ). 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.<sup>[179]</sup>

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. 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

`hrmax` 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 `hrmax` 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, 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);
```

## 14.4 Low-MODe (LMOD) optimization methods

István Kolossváry has contributed new functions, which implement the LMOD methods for minimization, conformational searching, and flexible docking.[180–183] 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()*.

### 14.4.1 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, 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,

$$Hv = [\nabla(x_{min} + h) - \nabla(x_{min})] / h \quad (14.1)$$

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  $\nabla x$  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.

### 14.4.2 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. <b>NOTE:</b> 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 14.2: 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.

### 14.4.3 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 14.2; the parameters in the `xmin_opt` structure are described in Table 14.3; 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 [185].

## 14.4 Low-MODE (LMOD) optimization methods

<i>Parameter list for xmin_opt</i>		
<i>keyword</i>	<i>default</i>	<i>meaning</i>
mol_struct_opt	1	<i>l</i> = 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. 14.1 in section 14.4.1). 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	<i>l</i> = modified Armijo [184](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 14.3: Options for xmin\_opt.

- 2: L-BFGS Limited-memory Broyden-Fletcher-Goldfarb-Shanno quasi-Newton algorithm [186]. 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 [185, 187]. 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.

#### 14.4.4 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 Kolossvary.
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;

```

```

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.

## 14.4.5 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 "l\_o.", 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 "x\_o." parameters via the call to *lmod\_opt\_int()* 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()*.

<i>keyword</i>	<i>default</i>	<i>meaning</i>
natm		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.

<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 14.4.2.). 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 14.4.7, note (5).
nconf	10	The maximum number of low-energy conformations stored in conflib[]. Note that the calling program is responsible for allocating memory for conflib[].
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.

#### 14.4 Low-MODE (LMODe) optimization methods

<i>keyword</i>	<i>default</i>	<i>meaning</i>
<code>lmod_restart</code>	10	The frequency, in LMOD iterations, of updating the conffib 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 = 1</code> 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 <code>RT</code> 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 14.4.2.
<code>lmod_step_size_max</code>	5.0	The maximum length of a single LMOD ZIG move in Å. See section 14.4.2.
<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 14.4.2. <code>nof_lmod_steps &gt; 0</code> 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 14.4.2.
<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>
apply_rigdock	2	The frequency, measured in LMOD iterations, of the application of rigid-body rotational and translational motions to the ligand(s). At each apply_rigdock-th LMOD iteration nof_pose_to_try rotations and translations are applied to the ligand(s).
nof_poses_to_try	10	The number of rigid-body rotational and translational motions applied to the ligand(s). Such applications occur at each apply_rigdock-th LMOD iteration. In case nof_pose_to_try > 1, it is always the lowest energy pose that is kept, all other poses are discarded.
random_seed	314159	The seed of the random number generator. A value of zero requests hardware seeding based on the system clock.
print_level	0	Amount of debugging printout. 0= No output. 1= Basic output. 2= Detailed output. 3= Copious debugging output including ARPACK details.
lmod_time	N/A	CPU time in seconds used by LMOD itself.
aux_time	N/A	CPU time in seconds used by auxiliary routines.
error_flag	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 eigenvector 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  $n_{\text{mod}} + 1 \leq \text{ndim\_arnoldi} \leq 3 * n_{\text{atm}}$ . 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.

#### 14.4.6 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

```

## 14.4 Low-MODE (LMOD) optimization methods

```

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          = %12.3lf 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 14.4.2). The number of lines is equal to 2 times *kmod* (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 *nmod* 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 14.4.2) 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 *confib[]* array is saved periodically in a binary disk file called *confib.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 *confib.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.

### 14.4.7 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  $A_{ij}$  and  $B_{ij}$  coefficients in the PRMTOPT 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 PRMTOPT 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  $A_{ij}$  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 conffib 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.

#### 14.4 Low-MODE (LMOD) optimization methods

However, the eigenvector calculations are likely to converge faster with the tethered systems.



## 15 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.

### 15.1 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 rtl );  
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 xoff-  
set, 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 `rtl` 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 the 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 they 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.

## 15.2 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 molecule's 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 `useboundsfrom()` 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, `conjgrad()` 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 `conjgrad()` 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.

### 15.2.1 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 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 }
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 );

```

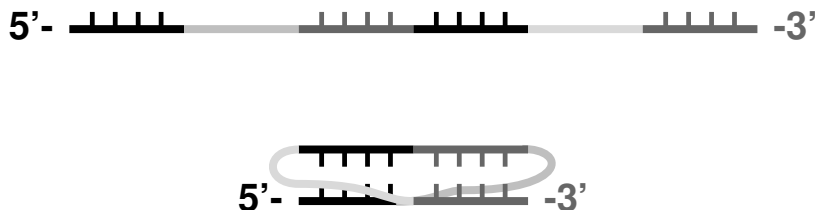


Figure 15.1: *Single-stranded RNA (top) folded into a pseudoknot (bottom). The black and dark grey base pairs can be stacked.*

```
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 `congrad()` 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.

## 15.2.2 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 15.1. 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 = "gcggaacgccgcguaagcg";
10
```

```

11 seqlen = length(seq);
12
13 m = link_na("1", seq, "rna.amber94.rlb", "rna", "35");
14
15 allocate xyz[ 4*m.natoms ];
16 allocate f[ 4*m.natoms ];
17 allocate v[ 4*m.natoms ];
18
19
20 b = newbounds(m, "");
21
22 for ( i = 1; i <= seqlen; i = i + 1 ) {
23     useboundsfrom(b, m, sprintf("1:%d:??,H?[^'T]", i), m,
24     sprintf("1:%d:??,H?[^'T]", i), 0.0 );
25 }
26
27 setboundsfromdb(b, m, "1:1:", "1:2:", "arna.stack.db", 1.0);
28 setboundsfromdb(b, m, "1:2:", "1:3:", "arna.stack.db", 1.0);
29 setboundsfromdb(b, m, "1:3:", "1:18:", "arna.stack.db", 1.0);
30 setboundsfromdb(b, m, "1:18:", "1:19:", "arna.stack.db", 1.0);
31 setboundsfromdb(b, m, "1:19:", "1:20:", "arna.stack.db", 1.0);
32
33 setboundsfromdb(b, m, "1:8:", "1:9:", "arna.stack.db", 1.0);
34 setboundsfromdb(b, m, "1:9:", "1:10:", "arna.stack.db", 1.0);
35 setboundsfromdb(b, m, "1:10:", "1:11:", "arna.stack.db", 1.0);
36 setboundsfromdb(b, m, "1:11:", "1:12:", "arna.stack.db", 1.0);
37 setboundsfromdb(b, m, "1:12:", "1:13:", "arna.stack.db", 1.0);
38
39 setboundsfromdb(b, m, "1:1:", "1:13:", "arna.basepair.db", 1.0);
40 setboundsfromdb(b, m, "1:2:", "1:12:", "arna.basepair.db", 1.0);
41 setboundsfromdb(b, m, "1:3:", "1:11:", "arna.basepair.db", 1.0);
42
43 setboundsfromdb(b, m, "1:8:", "1:20:", "arna.basepair.db", 1.0);
44 setboundsfromdb(b, m, "1:9:", "1:19:", "arna.basepair.db", 1.0);
45 setboundsfromdb(b, m, "1:10:", "1:18:", "arna.basepair.db", 1.0);
46
47 tsmooth(b, 0.0005);
48
49 opt = "seed=571, gdist=0, ntp=50, k4d=2.0, randpair=5.";
50 dg_options( b, opt );
51 embed(b, xyz );
52
53 for ( i = 3000; i > 2800; i = i - 100 ){
54     conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 500 );
55
56     dg_options( b, "ntp=1000, k4d=0.2" );
57     mm_options( "ntp_md=50, zerov=1, temp0="+sprintf("%d.",i));
58     md( 4*m.natoms, 1000, xyz, f, v, db_viol );
59

```

## 15 NAB: Sample programs

```
60     dg_options( b, "ntpr=1000, k4d=4.0" );
61     mm_options( "zerov=0, temp0=0., tautp=0.3" );
62     md( 4*m.natoms, 8000, xyz, f, v, db_viol );
63 }
64
65 setmol_from_xyz( m, NULL, xyz );
66 putpdb( "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 15-18 define arrays that will be used in the simulated annealing of the structure. The bounds object is created in line 20 which automatically sets the 1-2, 1-3, and 1-4 distance bounds in the molecule. The loop in lines 22-25 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 27-45, 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 27 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 39 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 47 smooths the bounds matrix, by attempting to adjust any sets of bounds that violate the triangle equality. Lines 49-50 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 51.

The coordinates `xyz` are subject to a series of conjugate gradient refinements and simulated annealing in lines 53-63. Line 65 replaces the old molecular coordinates with the new refined ones, and lastly, on line 66, the molecule is saved as "pseudoknot.pdb".

The resulting structure of Program 8 is shown in Figure 15.2. This structure had an final total energy of 9.41 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.



## 15 NAB: Sample programs

```
13 string iresname, jresname, iat, jat, aex1, aex2, aex3, aex4, line, dgopts, seq;
14
15 // sequence of the mrf2 protein:
16 seq = "RADEQAFLVALYKYMKERKTPIERIPYLGFKQINLWTFQAAQKLGGYETITARRQWKHIY"
17     + "DELGGNPGSTSAATCTRRHYERLILPYERFIKGEEDKPLPPIKPRK";
18
19 // build this sequence in an extended conformation, and construct a bounds
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 // (hardware 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:

```

## 15 NAB: Sample programs

```
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.

## 15.3 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.

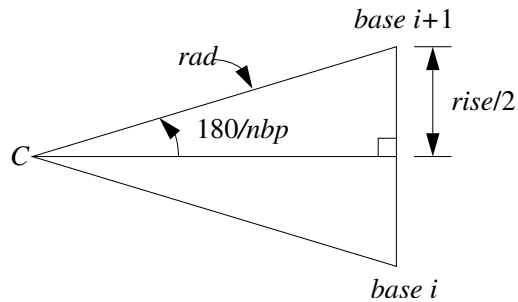
### 15.3.1 Closed Circular DNA

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 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  $\Delta linknumber$  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;
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  $\Delta$ linking number 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 a 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  $\Delta$ linking number (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.

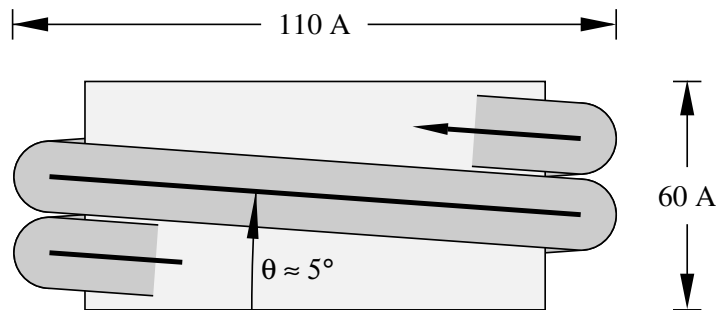
After the loop exit, since this is a *closed* circular molecule the first and last bases of each strand must be bonded and this is done with the two calls to `connectres()` in lines 67-68. The last step is to save the molecule's coordinates and connectivity in lines 71-72. The `nab` builtin `putpdb()` 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.

### 15.3.2 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  $\theta$  (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:

$$d = 3.38 \sin(\theta); \phi = 360t/(n-1) \quad (15.1)$$

$$r = \frac{3.38(n-1)\cos(\theta)}{2\pi t} \quad (15.2)$$

where  $d$  and  $\phi$  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  $360t\phi$ . The duplex contains  $(n-1)$  steps, resulting in  $360t/(n-1)\phi$  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. 15.2.

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 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()`.

## 15.4 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.

### 15.4.1 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  $x_i$ s is that they be monotonically increasing we can simply use the sequence  $1, 2, \dots, n$  for the  $x_i$ , 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 ] );

```

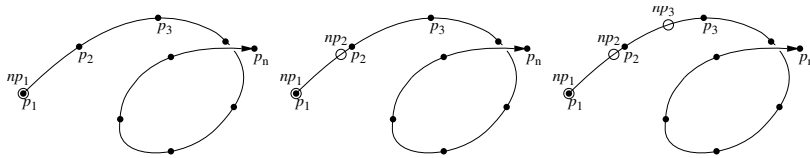
## 15 NAB: Sample programs

```
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];
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 numbers/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 sec-

ond 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  $np_{pts}$  sweeps out the curve from  $(x_1, y_1, z_1)$  to  $(x_{np_{pts}}, y_{np_{pts}}, z_{np_{pts}})$ . 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 is 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.

## 15.4.2 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 );

```

### 15.4.3 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.

## 15 NAB: Sample programs

```

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 {
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.

## 15.5 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] Geney, R.; Layten, M.; Gomperts, R.; Simmerling, C. Investigation of salt bridge stability in a generalized Born solvent model. *J. Chem. Theory Comput.*, **2006**, *2*, 115–127.
- [3] Okur, A.; Wickstrom, L.; Simmerling, C. Evaluation of salt bridge structure and energetics in peptides using explicit, implicit and hybrid solvation models. *J. Chem. Theory Comput.*, **2008**, *4*, 488–498.
- [4] Okur, A.; Wickstrom, L.; Layten, M.; Geney, R.; Song, K.; Hornak, V.; Simmerling, C. Improved efficiency of replica exchange simulations through use of a hybrid explicit/implicit solvation model. *J. Chem. Theory Comput.*, **2006**, *2*, 420–433.
- [5] Roe, D.R.; Okur, A.; Wickstrom, L.; Hornak, V.; Simmerling, C. Secondary Structure Bias in Generalized Born Solvent Models: Comparison of Conformational Ensembles and Free Energy of Solvent Polarization from Explicit and Implicit Solvation. *J. Phys. Chem. B*, **2007**, *111*, 1846–1857.
- [6] Ren, P.; Ponder, J.W. Consistent treatment of inter- and intramolecular polarization in molecular mechanics calculations. *J. Comput. Chem.*, **2002**, *23*, 1497–1506.
- [7] Ren, P.Y.; Ponder, J.W. Polarizable atomic multipole water model for molecular mechanics simulation. *J. Phys. Chem. B*, **2003**, *107*, 5933–5947.
- [8] Ren, P.; Ponder, J.W. Temperature and pressure dependence of the AMOEBA water model. *J. Phys. Chem. B*, **2004**, *108*, 13427–13437.
- [9] Ren, P.Y.; Ponder, J.W. Tinker polarizable atomic multipole force field for proteins. *to be published.*, **2006**.
- [10] 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.
- [11] 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.

## BIBLIOGRAPHY

- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] García, A.E.; Sanbonmatsu, K.Y.  $\alpha$ -helical stabilization by side chain shielding of backbone hydrogen bonds. *Proc. Natl. Acad. Sci. USA*, **2002**, *99*, 2782–2787.
- [17] Sorin, E.J.; Pande, V.S. Exploring the helix-coil transition via all-atom equilibrium ensemble simulations. *Biophys. J.*, **2005**, *88*, 2472–2493.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] Cieplak, P.; Dupradeau, F.-Y.; Duan, Y.; Wang, J. Polarization effects in molecular mechanical force fields. *J. Phys.: Condens. Matter*, **2009**, *21*, 333102.
- [22] 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.
- [23] 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.
- [24] 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.

- [25] 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.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] Kirschner, K.N.; Woods, R.J. Solvent interactions determine carbohydrate conformation. *Proc. Natl. Acad. Sci. USA*, **2001**, *98*, 10541–10545.
- [30] Woods, R.J. Restrained electrostatic potential charges for condensed phase simulations of carbohydrates. *J. Mol. Struct (Theochem)*, **2000**, *527*, 149–156.
- [31] Woods, R.J. Derivation of net atomic charges from molecular electrostatic potentials. *J. Comput. Chem.*, **1990**, *11*, 29–310.
- [32] 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.
- [33] 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.
- [34] DeMarco, M.L.; Woods, R.J. Bridging computational biology and glycobiology: A game of snakes and ladders. *Glycobiology, in press*, **2008**.
- [35] Åqvist, J. Ion-water interaction potentials derived from free energy perturbation simulations. *J. Phys. Chem.*, **1990**, *94*, 8021–8024.
- [36] 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.
- [37] 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.
- [38] Joung, S.; Cheatham, T.E. III. Determination of alkali and halide monovalent ion parameters for use in explicitly solvated biomolecular simulations. *J. Chem. Phys. Bo*, **2008**, *112*, 9020–9041.
- [39] 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.

## BIBLIOGRAPHY

- [40] Price, D.J.; Brooks, C.L. A modified TIP3P water potential for simulation with Ewald summation. *J. Chem. Phys.*, **2004**, *121*, 10096–10103.
- [41] Jorgensen, W.L.; Madura, J.D. Temperature and size dependence for Monte Carlo simulations of TIP4P water. *Mol. Phys.*, **1985**, *56*, 1381–1392.
- [42] Horn, H.W.; Swope, W.C.; Pitera, 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.
- [43] Horn, H.W.; Swope, W.C.; Pitera, J.W. Characterization of the TIP4P-Ew water model: Vapor pressure and boiling point. *J. Chem. Phys.*, **2005**, *123*, 194504.
- [44] 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.
- [45] 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.
- [46] Berendsen, H.J.C.; Grigera, J.R.; Straatsma, T.P. The missing term in effective pair potentials. *J. Phys. Chem.*, **1987**, *91*, 6269–6271.
- [47] 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.
- [48] 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.
- [49] 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.
- [50] 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.
- [51] 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.
- [52] 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.
- [53] Higo, J.; Ito, N.; Kuroda, M.; Ono, S.; Nakajima, N.; Nakamura, H. Energy landscape of a peptide consisting of  $\alpha$ -helix,  $3_{10}$  helix,  $\beta$ -turn,  $\beta$ -hairpin and other disordered conformations. *Prot. Sci.*, **2001**, *10*, 1160–1171.

- [54] 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.
- [55] 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.
- [56] 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.
- [57] 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.
- [58] 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.
- [59] 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.
- [60] 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.
- [61] Brooks, B.R.; Brucoleri, 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.
- [62] 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.; Hodoseck, 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.
- [63] 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.
- [64] 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.

## BIBLIOGRAPHY

- [65] Bondi, A. van der Waals volumes and radii. *J. Phys. Chem.*, **1964**, *68*, 441–451.
- [66] 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.
- [67] 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.
- [68] 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.
- [69] 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.
- [70] 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.
- [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] Stewart, J.J.P. Optimization of parameters for semiempirical methods I. Method. *J. Comput. Chem.*, **1989**, *10*, 209–220.
- [78] 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.
- [79] 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.
- [80] 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.

- [81] Repasky, M.P.; Chandrasekhar, J.; Jorgensen, W.L. PDDG/PM3 and PDDG/MNDO: Improved semiempirical methods. *J. Comput. Chem.*, **2002**, *23*, 1601–1622.
- [82] 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.
- [83] 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.
- [84] 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.
- [85] 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.
- [86] 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.
- [87] 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.
- [88] 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.
- [89] 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.
- [90] 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.
- [91] Pellegrini, E.; J. Field, M. A generalized-Born solvation model for macromolecular hybrid-potential calculations. *J. Phys. Chem. A*, **2002**, *106*, 1316–1326.
- [92] Kruger, T.; Elstner, M.; Schiffels, P.; Frauenheim, T. Validation of the density-functional based tight-binding approximation. *J. Chem. Phys.*, **2005**, *122*, 114110.
- [93] 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.

## BIBLIOGRAPHY

- [94] Kabsch, W.; Sander, C. Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers*, **1983**, *22*, 2577–2637.
- [95] Prompers, J.J.; Brüschweiler, R. General framework for studying the dynamics of folded and nonfolded proteins by NMR relaxation spectroscopy and MD simulation. *J. Am. Chem. Soc.*, **2002**, *124*, 4522–4534.
- [96] Prompers, J.J.; Brüschweiler, R. Dynamic and structural analysis of isotropically distributed molecular ensembles. *Proteins*, **2002**, *46*, 177–189.
- [97] Luo, R.; David, L.; Gilson, M.K. Accelerated Poisson-Boltzmann calculations for static and dynamic systems. *J. Comput. Chem.*, **2002**, *23*, 1244–1253.
- [98] Wang, J.; Luo, R. Assessment of Linear Finite-Difference Poisson-Boltzmann Solvers. *J. Comput. Chem.*, **2010**, *in press*.
- [99] Cai, Q.; Hsieh, M.-J.; Wang, J.; Luo, R. Performance of Nonlinear Finite-Difference Poisson-Boltzmann Solvers. *J. Chem. Theory Comput.*, **2010**, *in press*.
- [100] Honig, B.; Nicholls, A. Classical electrostatics in biology and chemistry. *Science*, **1995**, *268*, 1144–1149.
- [101] Lu, Q.; Luo, R. A Poisson-Boltzmann dynamics method with nonperiodic boundary condition. *J. Chem. Phys.*, **2003**, *119*, 11035–11047.
- [102] 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.
- [103] Warwicker, J.; Watson, H.C. Calculation of the electric potential in the active site cleft due to. *J. Mol. Biol.*, **1982**, *157*, 671–679.
- [104] Klapper, I.; Hagstrom, R.; Fine, R.; Sharp, K.; Honig, B. Focussing of electric fields in the active site of Cu, Zn superoxide dismutase. *Proteins*, **1986**, *1*, 47–59.
- [105] Sitkoff, D.; Sharp, K.A.; Honig, B. Accurate calculation of hydration free energies using macroscopic solvent models. *J. Phys. Chem.*, **1994**, *98*, 1978–1988.
- [106] Tan, C. H.; Tan, Y. H.; Luo, R. Implicit nonpolar solvent models. *J. Phys. Chem. B*, **2007**, *111*, 12263–12274.
- [107] 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.
- [108] 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.
- [109] 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.

- [110] 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.
- [111] Bashford, D. An object-oriented programming suite for electrostatic effects in biological molecules. *Lect. Notes Comput. Sci.*, **1997**, *1343*, 233–240.
- [112] 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.
- [113] Cai, Q.; Wang, J.; Zhao, H.; Luo, R. On removal of charge singularity in Poisson-Boltzmann equation. *J. Chem. Phys.*, **2009**, *130*, 145101.
- [114] 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.
- [115] Davis, M.E.; McCammon, J.A. Electrostatics in biomolecular structure and dynamics. *Chem. Rev.*, **1990**, *90*, 509–521.
- [116] 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.
- [117] 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.
- [118] 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.
- [119] Chandler, D.; Andersen, H. C. Optimized cluster expansions for classical fluids. ii. theory of molecular liquids. *J. Chem. Phys.*, **1972**, *57*(5), 1930–1937.
- [120] Hirata, F.; Rossky, P. J. An extended RISM equation for molecular polar fluids. *Chem. Phys. Lett.*, **1981**, pp 329–334.
- [121] 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.
- [122] 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.
- [123] Chandler, D.; McCoy, J.; Singer, S. Density functional theory of nonuniform polyatomic systems. i. general formulation. *J. Chem. Phys.*, **1986**, *85*(10), 5971–5976.

## BIBLIOGRAPHY

- [124] 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.
- [125] 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.
- [126] 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.
- [127] 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.
- [128] 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.
- [129] Kovalenko, A. In Hirata [188], chapter 4.
- [130] 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.
- [131] 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.
- [132] Hansen, J.-P.; McDonald, I. R. *Theory of simple liquids*. Academic Press, London, 1990.
- [133] Hirata, F. In *Molecular Theory of Solvation* [188], chapter 1.
- [134] Perkyns, J. S.; Pettitt, B. M. A site-site theory for finite concentration saline solutions. *J. Chem. Phys.*, **1992**, 97(10), 7656–7666.
- [135] 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.
- [136] Kaminski, J. W.; Gusarov, S.; Wesolowski, T. A.; Kovalenko, A.; Andiry. Modeling solvatochromic shifts using the orbital-free embedding potential at statistically mechanically averaged solvent density. *J. Phys. Chem. A*, **2010**, *in press*.
- [137] 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.
- [138] 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.

- [139] 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.
- [140] 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.
- [141] Genheden, S.; Luchko, T.; Gusarov, S.; Kovalenko, A.; Ryde, U. An MM/3D-RISM approach for ligand-binding affinities. *J. Phys. Chem.*, **2010**. Accepted.
- [142] 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.
- [143] Ahmed, A.; Gohlke, H. Multiscale modeling of macromolecular conformational changes combining concepts from rigidity and elastic network theory. *Proteins*, **2006**, *63*, 1038–1051.
- [144] Fulle, S.; Gohlke, H. Analyzing the flexibility of RNA structures by constraint counting. *Biophys. J.*, **2008**, DOI:10.1529/biophysj.107.113415.
- [145] Sigalov, G.; Fenley, A.; Onufriev, A. Analytical electrostatics for biomolecules: Beyond the generalized Born approximation. *J. Chem. Phys.*, **2006**, *124*, 124902.
- [146] 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.
- [147] 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.
- [148] Turcotte, M.; Lapalme, G.; Major, F. Exploring the conformations of nucleic acids. *J. Funct. Program.*, **1995**, *5*, 443–460.
- [149] 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.
- [150] 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.
- [151] 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.
- [152] Zhurkin, V.B.; Lysov, Yu.; Ivanov, V.I. Different Families of Double Stranded Conformations of DNA as Revealed by Computer Calculations. *Biopolymers*, **1978**, *17*, 277–312.
- [153] Lavery, R.; Zakrzewska, K.; Skelnar, H. JUMNA (junction minimisation of nucleic acids). *Comp. Phys. Commun.*, **1995**, *91*, 135–158.

## BIBLIOGRAPHY

- [154] Gabarro-Arpa, J.; Cagnet, 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.
- [155] 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.
- [156] Crippen, G.M.; Havel, T.F. *Distance Geometry and Molecular Conformation*. Research Studies Press, Taunton, England, 1988.
- [157] 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.
- [158] 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.
- [159] 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.
- [160] 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 Ca<sup>2+</sup>-binding proteins. *Nature Struct. Biol.*, **1995**, *2*, 790–796.
- [161] 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.
- [162] 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.
- [163] Macke, T.J. *NAB, a Language for Molecular Manipulation*. 1996.
- [164] Dickerson, R.E. Definitions and Nomenclature of Nucleic Acid Structure Parameters. *J. Biomol. Struct. Dyn.*, **1989**, *6*, 627–634.
- [165] Zhurkin, V.B.; Raghunathan, G.; Ulynaov, 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.
- [166] Tan, R.; Harvey, S. Molecular Mechanics Model of Supercoiled DNA. *J. Mol. Biol.*, **1989**, *205*, 573–591.
- [167] Babcock, M.S.; Pednault, E.P.D.; Olson, W.K. Nucleic Acid Structure Analysis. *J. Mol. Biol.*, **1994**, *237*, 125–156.

- [168] Havel, T.F.; Kuntz, I.D.; Crippen, G.M. The theory and practice of distance geometry. *Bull. Math. Biol.*, **1983**, *45*, 665–720.
- [169] 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.
- [170] 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.
- [171] 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.
- [172] Agrafiotis, D.K. Stochastic Proximity Embedding. *J. Computat. Chem.*, **2003**, *24*, 1215–1221.
- [173] Saenger, W. in *Principles of Nucleic Acid Structure*, p 120. Springer-Verlag, New York, 1984.
- [174] 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.
- [175] 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.
- [176] Brooks, C.; Brünger, A.; Karplus, M. Active site dynamics in protein molecules: A stochastic boundary molecular-dynamics approach. *Biopolymers*, **1985**, *24*, 843–865.
- [177] Onufriev, A.; Bashford, D.; Case, D.A. Modification of the generalized Born model suitable for macromolecules. *J. Phys. Chem. B*, **2000**, *104*, 3712–3720.
- [178] 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.
- [179] Nguyen, D.T.; Case, D.A. On finding stationary states on large-molecule potential energy surfaces. *J. Phys. Chem.*, **1985**, *89*, 4020–4026.
- [180] Kolossvary, 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.
- [181] Kolossvary, I.; Guida, W.C. Low-mode conformatinoal search elucidated: Application to C<sub>39</sub>H<sub>80</sub> and flexible docking of 9-deazaguanine inhibitors into PNP. *J. Comput. Chem.*, **1999**, *20*, 1671–1684.
- [182] Kolossvary, I.; Keseru, 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.

## BIBLIOGRAPHY

- [183] 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.
- [184] Shi, Z.J.; Shen, J. New inexact line search method for unconstrained optimization. *J. Optim. Theory Appl.*, **2005**, *127*, 425–446.
- [185] 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.
- [186] Liu, D.C.; Nocedal, J. On the limited memory method for large scale optimization. *Math. Programming B*, **1989**, *45*, 503–528.
- [187] Nocedal, J.; L. Morales, J. Automatic preconditioning by limited memory quasi-Newton updating. *SIAM J. Opt.*, **2000**, *10*, 1079–1096.
- [188] Hirata, F., Ed. *Molecular Theory of Solvation*. Kluwer Academic Publishers, 2003.

# Index

1D-RISM, [157](#), [159](#), [163](#)  
3D-RISM, [157](#), [160](#), [166](#)

## A

[accept](#), [275](#)  
[acdoctor](#), [89](#)  
[acos](#), [239](#)  
[adbcor](#), [165](#)  
[add](#), [48](#)  
[addAtomTypes](#), [49](#)  
[addIons](#), [50](#)  
[addIons2](#), [50](#)  
[addPath](#), [50](#)  
[addPdbAtomMap](#), [50](#)  
[addPdbResMap](#), [51](#)  
[addressidue](#), [192](#), [243](#)  
[addstrand](#), [192](#), [243](#)  
[alias](#), [52](#)  
[alignframe](#), [200](#), [251](#)  
[allatom\\_to\\_dna3](#), [245](#)  
[allocate](#), [227](#)  
[am1bcc](#), [85](#)  
[andbounds](#), [260](#)  
[angle](#), [247](#)  
[anglep](#), [248](#)  
[antechamber](#), [78](#)  
[asin](#), [239](#)  
[assert](#), [249](#)  
[atan](#), [239](#)  
[atan2](#), [239](#)  
[atof](#), [239](#)  
[atoi](#), [239](#)  
[atomtype](#), [84](#)

## B

[basepair](#), [200](#)  
[bcopt](#), [147](#)

[bdna](#), [200](#), [301](#)  
[bdna\(\)](#), [201](#)  
[blocksize](#), [279](#)  
[bond](#), [52](#)  
[bondByDistance](#), [52](#)  
[bondtype](#), [86](#)  
[break](#), [234](#)  
[bridge function](#), [158](#)  
[buffer](#), [277](#)

## C

[ceil](#), [239](#)  
[centering](#), [278](#)  
[check](#), [52](#)  
[closures](#), [163](#)  
[closure](#), [277](#)  
[combine](#), [53](#)  
[complement](#), [200](#)  
[conjgrad](#), [271](#)  
[connectres](#), [192](#), [243](#)  
[continue](#), [234](#)  
[copy](#), [53](#)  
[copymolecule](#), [243](#)  
[cos](#), [239](#)  
[cosh](#), [239](#)  
[countmolatoms](#), [248](#)  
[createAtom](#), [54](#)  
[createResidue](#), [54](#)  
[createUnit](#), [54](#)  
[cut](#), [273](#)  
[cutfd](#), [148](#)  
[cutnb](#), [148](#)  
[cutres](#), [148](#)  
[cuvfile](#), [276](#)

## D

[database](#), [90](#)

## INDEX

date, 250  
dbfopt, 148  
deallocate, 227  
debug, 249  
decompopt, 149  
del, 277  
delete, 232  
deleteBond, 54  
delvv, 164  
density, 164  
desc, 54  
dftb\_3rd\_order, 98  
dftb\_chg, 99  
dftb\_disper, 98  
dftb\_maxiter, 99  
dftb\_telec, 99  
dg\_helix, 301  
dg\_options, 261  
diag\_routine, 100  
diel, 274  
dielc, 274  
dieps, 165  
dim, 273  
direct correlation function, 157  
dist, 248  
distp, 248  
dna3, 245  
dna3\_to\_allatom, 245  
dprob, 146, 275  
dr, 163  
dumpatom, 249  
dumpbounds, 249  
dumpboundsviolations, 249  
dumpmatrix, 249  
dumpmolecule, 249  
dumpresidue, 249

### E

e\_debug, 273  
embed, 261  
eneopt, 147  
epsxt, 274  
epsin, 145, 275  
epsout, 145, 275  
errconv, 102

espgen, 87  
excess chemical potential, 160  
exit, 240  
exp, 239

### F

fabs, 239  
fclose, 240  
fd\_helix, 244  
fillratio, 146, 275  
floor, 239  
fmod, 239  
fopen, 240  
force, 278  
fprintf, 240  
frcopt, 148  
freemolecule, 243  
freeresidue, 243  
fscanf, 240  
ftime, 250

### G

gamma\_ln, 273  
gauss, 239  
gb, 274  
gb2\_debug, 273  
gb\_debug, 273  
gbsa, 274  
genmass, 274  
geodesics, 261  
getchivol, 261  
getchivolp, 261  
getcif, 246  
getline, 240  
getmatrix, 242  
getpdb, 246  
getpdb\_prm, 244, 271  
getres, 193, 200  
getresidue, 192, 193, 246  
gettriad(), 217  
getxv, 271  
getxyz, 271  
grdspc, 277  
grms\_tol, 102  
groupSelectedAtoms, 55

gsub, 238  
guvfile, 276

**H**

helix, 200  
helixanal, 248  
HNC, 158, 160  
huvfile, 276  
hypernetted-chain approximation, 158

**I**

imin, 143  
impose, 56  
index, 238  
inp, 144, 275  
intramolecular pair correlation matrix, 159  
ipb, 275, 276  
iprob, 275  
irism, 276  
istrng, 145, 275  
itrmax, 101

**K**

k4d, 273  
kappa, 274  
KH, 158, 160  
Kovalenko-Hirata, 158  
ksave, 164  
kshow, 164

**L**

length, 238  
link\_na, 244  
linkprot, 243  
list, 57  
lmod, 291  
loadAmberParams, 57  
loadAmberPrep, 57  
loadMol2, 58  
loadOff, 57  
loadPdb, 58  
loadPdbUsingSeq, 58  
log, 239  
log10, 239  
logFile, 58

**M**

match, 238  
MAT\_cube, etc, 253  
matextract, 258  
MAT\_fprint, etc, 254  
matgen, 255  
matmerge, 257  
maxcyc, 101  
maxitn, 146, 275  
maxsph, 146, 150  
maxste, 164  
maxstep, 277  
md, 271  
MDIIS, 160  
MDL, 167  
mean solvation force, 160  
measureGeom, 59  
mergestr, 192, 243  
method, 277  
mk\_dimer(), 217  
mme, 271  
mme2, 281  
mme\_init, 271  
mme\_rattle, 271  
mm\_options, 271  
mm\_set\_checkpoint, 271  
model, 164  
modified direct inversion of the iterative sub-  
space, 159  
molsurf, 248  
MPI, 187

**N**

NAB, 166, 167  
nbuffer, 147  
nchk, 273  
nchk2, 273  
ndiis\_attempts, 102  
ndiis\_matrices, 102  
newbounds, 260  
newmolecule, 192, 243  
newton, 281  
newtransform, 198, 251  
nfocus, 275  
ng3, 277

## INDEX

nis, 164  
nmode, 281  
npbgrid, 147  
npbopt, 275  
npbverb, 149  
npopt, 149  
npropagate, 278  
nr, 163  
nsnb, 273  
nsnba, 148  
nsnbr, 148  
nsp, 165  
ntpr, 101, 273  
ntpr\_md, 273  
ntwrism, 278  
ntwx, 273  
ntx, 144

### O

offset, 150  
OMP\_NUM\_THREADS, 187  
OpenDX, 170  
orbounds, 260  
Ornstein-Zernike, 157  
outlst, 164

### P

pair-distribution function, 157  
parmcals, 90  
parmchk, 80  
pbtemp, 145  
peptide\_corr, 101  
phiform, 149  
plane, 248  
point, 237  
pow, 239  
prepgen, 87  
printcharges, 101  
printf, 240  
progress, 279  
pseudo\_diag, 100  
pseudo\_diag\_criteria, 100  
putbnd, 246  
putcif, 246  
putdist, 246

putmatrix, 242  
putpdb, 246  
putxv, 271  
putxyz, 271

### Q

qmcharge, 99  
qmcmdx, 99  
qm\_theory, 98

### R

radiopt, 145, 275  
rand2, 239  
rattle, 273  
readparm, 271  
remove, 59  
residuegen, 90  
respgen, 88  
rgbmax, 274  
rhow\_effect, 150  
RISM, 157  
rismld, 157, 163  
rmsd, 247  
rot4, 199, 251  
rot4p, 199, 251  
routup, 164  
rseed, 239

### S

saveAmberParm, 60  
saveMol2, 60  
saveOff, 60  
savePdb, 60  
ScaLAPACK, 187  
scalec, 146, 148  
scanf, 240  
scfconv, 100  
second, 250  
sequence, 61  
set, 61  
setbounds, 260  
setboundsfromdb, 261  
setchiplane, 261  
setchivol, 261  
setframe, 199, 251

setframep, 200, 251  
 setmol\_from\_xyz, 252  
 setmol\_from\_xyzw, 252  
 setpoint, 252  
 setseed, 239  
 setxyz\_from\_mol, 252  
 setxyzw\_from\_mol, 252  
 showbounds, 260  
 sin, 239  
 sinh, 239  
 smear, 165  
 smoothopt, 144, 145, 275  
 solvateBox, 63  
 solvateCap, 63  
 solvateOct, 63  
 solvateShell, 64  
 solvation, 157, 160  
 solvation free energy, 160  
 solvbox, 277  
 solvcut, 277  
 solvopt, 275  
 space, 147, 275  
 spin, 99  
 split, 238  
 sprintf, 240  
 sprob, 149  
 sqrt, 239  
 sscanf, 240  
 static\_arrays, 279  
 sub, 238  
 substr, 238  
 sugarpuckeranal, 248  
 superimpose, 247  
 surfTen, 150, 274  
 system, 240

**T**

t, 273  
 tan, 239  
 tanh, 239  
 tautp, 273  
 temp0, 273  
 temper, 165  
 tempi, 274  
 theory, 163

thermodynamics, 160  
 tight\_p\_conv, 100  
 timeofday, 250  
 tolerance, 277  
 tolvv, 164  
 torsion, 248  
 torsionp, 248  
 total correlation function, 157  
 toutup, 164  
 trans4, 199  
 trans4p, 199  
 transform, 64, 258  
 transformmol, 199, 252  
 transformres, 192, 193, 199, 252  
 translate, 65, 91, 92  
 tsmooth, 261

**U**

unlink, 240  
 useboundsfrom, 260  
 use\_rmin, 149  
 use\_sav, 150

**V**

vec, 277  
 verbose, 279  
 verbosity, 65, 99  
 vlimit, 273  
 vprob, 150

**W**

wc\_basepair, 301  
 wc\_basepair(), 203  
 wc\_complement, 301  
 wc\_complement(), 201  
 wc\_helix, 301  
 wc\_helix(), 206  
 wcons, 273

**X**

xmin, 286  
 XVV, 169  
 xvfile, 276

**Z**

zerofrc, 278

## *INDEX*

zerov, [273](#)  
zMatrix, [65](#)