AmberClassic Reference Manual

The starting point for this collection is *msander* ("modern sander"), an updated version of *sander*, the basic energy minimizer and molecular dynamics program in the *AmberTools* suite. A variety of constraints to be added to the basic force field, and *msander* has been designed especially for the types of calculations involved in NMR, Xray or cryo-EM structure refinement.

This collection also contains versions of a number of the "classic" (and most-used) parts of `AmberTools`: *tleap*, *antechamber*, *sqm*, *saxs*, *rism1d*, *metatwist* and *paramfit*. With these tools, many systems can be set up for simulation in *msander*. The tutorials at ambermd/org/tutorials illustrate the use of these programs. (Analysis of trajectories is most commonly carried out via *cpptraj*, which is distributed separately, at github.com/Amber-MD.) If you need more capabilities than these classic, stripped-down versions provide, consider installing the full `AmberTools` and `Amber` suites, available at ambermd.org.

One goal of this collection is to make compiling and installation as simple as possible. There is a pretty simple `configure` script, and minimal dependencies on external packages. If you have problems or questions, creating an issue at `github.com` is the recommended path, but you can also send email to dacase1@gmail.com.

Principal contributors to the current codes:

David A. Case (Rutgers)
Carlos Simmerling (Stony Brook)
Adrian Roitberg (Florida)
Tom Darden (OpenEye)
Junmei Wang (Pitt)
Chris Schafmeister (Temple)
Tom Macke
István Kolossváry (Budapest)
Celeste Sagui (NCSU)
Feng Pan (FSU)
Jason Swails (Entos)
David Cerutti (Psivant Discovery)
Tyler Luchko (CSU Northridge)
Ross C. Walker (Independent Consultant)
George Giambaşu (Merck)
Jian Liu (Peking Univ.)
Vinícius Wilian D. Cruzeiro (UC San Diego)
Nikolai R. Skrynnikov (Purdue, SPbU)
Oleg Mikhailovskii (Purdue, SPbU)
Yi Xue (Tsinghua)
Sergei A. Izmailov (SPbU)
Alexey Onufriev (Virginia Tech)
Saeed Izadi (Genentech)
Xiongwu Wu (NIH)
George Giambasu (Rutgers)
Jian Liu (Peking Univ.)
Andriy Kovalenko (NINT)

Contents

1 Introduction

This repository contains *msander*, a "modern" version of parts of the Amber molecular dynamics program sander. Also included are various NMR, X-ray and cryoEM-related code and utilities, as well as versions of a number of the “classic” (and most-used) parts of AmberTools: *tleap, antechamber, sqm, metatwist, rism1d, saxs, NAB, nabc* and *paramfit*. With these tools, many systems can be set up for simulation in *msander*.

This code is probably most useful to those who are already familiar with ‘AmberTools‘ (https://ambermd.org). Many of the basic tutorials there will also work with AmberClassic. The documentation and authorship credits are in the `doc/AmberClassic.pdf` file.

This package may also be of interest to those who want just the subset included here of the far-more-complex AmberTools package. Some other popular parts of *AmberTools* are not included here, but are available separately: *cpptraj* and *pytraj* (both at github.com/Amber-MD), and *parmed* (at github.com/ParmEd).

Warning

This is a work in progress, and may not always be in a stable state (although that is my goal for the main branch). I may not be able to respond to requests for support, but please create a github issue if you have comments or suggestions. (As an alternative, send email to *dacase1@gmail.com*.)

Design goals

This project began as a fork of the *sander* code in *AmberTools*. It tries to (greatly) simplify the code base, choosing the best and most useful parts of the code, and to serve as a test bed for how modern Fortran coding techniques can be used. Key application areas are expected to be in structure refinements using NMR, cryoEM or Xray diffraction information. This version has a fair amount of OpenMP support, especially for Xray and 3D-RISM calculations. Parts of the Xray code uses GPU acceleration.

One additional goal of this collection is to make compiling and installation as simple as possible. There is a pretty simple `configure` script, and minimal dependencies on external packages. I am (slowly) cleaning up and adding other parts of *AmberTools*, and I hope to create a conda package soon.

Key differences in functionality versus sander

Some pieces are missing from the sander program in AmberTools:

- Things that should be easy to re-introduce later: emil, sebomd, pbsa, APBS
- Things are are problably gone for good, but which don’t represent the best current practice: Path-integral methods, thermostats that don’t follow the "middle" scheme, Berendsen barostat
- Things that might be useful, but really complicate the code: evb potentials, QM/MM, nudged elastic band, constant pH and constant redox potential simulations. The API interface has also been removed.
- Non-periodic 3D-RISM has been removed for now, in an attempt to get the simplest possible RISM code, perhaps as a basis for future GPU work.

(If you need some of these deleted pieces, use *sander* from AmberTools instead.)

Key pieces of code that are still there, and being emphasized:

- Periodic and non-periodic simulations, with all of Amber's GB models
- 3D-RISM in periodic boundary conditions
- NMR, cryoEM and Xray restraints (including quite a bit of new code; Xray restraints include NVIDIA GPU-enabled capabilities)
- Thermodynamic integration and non-equilibrium sampling methods, including adaptively biassed sampling and self-guided Langevin dynamics
- Sampling and minimization using the lmod and xmin approaches; these can now be used in conjunction with SHAKE and SETTLE.
- Replica exchange capabilities, except for constant pH and redox potential simulations

Installation

In the top-level folder:

```
./configure --help      # then choose options
make install
source AmberClassic.sh  # sets up environment variables; consider adding to your SHELL
make test               # optional--more info is below
```

The test suite also serves as a source of example input files. Look especially in the *rism*, *xray* and *cryoem* folders to see sample inputs for these sorts of calculation.

License

This project is licensed under the GNU General Public License, version 2, or (at your option) any later version. Some components use different, but compatible, open source licenses. See the `LICENSE` file for more information.

2 msander

2.1 Introduction

This is a guide to *msander*, an Amber module which carries out energy minimization, molecular dynamics, and NMR refinements. The acronym stands for **S**imulated **A**nnealing with **N**MR-**D**erived **E**nergy **R**estraints, but this module is used for a variety of simulations that have nothing to do with NMR refinement. Some general features are outlined in the following paragraphs:

1. *msander* provides direct support for several force fields for proteins and nucleic acids, and for several water models and other organic solvents. The basic force field implemented here has the following form, which is about the simplest functional form that preserves the essential nature of molecules in condensed phases:

$$\begin{aligned} V(\mathbf{r}) &= \sum_{bonds} K_b(b-b_0)^2 + \sum_{angles} K_\theta(\theta-\theta_o)^2 \\ &+ \sum_{dihedrals} (V_n/2)(1+\cos[n\phi-\delta] \\ &+ \sum_{nonb\,ij} (A_{ij}/r_{ij}^{12}) - (B_{ij}/r_{ij}^{6}) + (q_i q_j/r_{ij}) \end{aligned}$$

   In addition, charges that are not centered on atoms, but are off-center (as for lone-pairs or "extra points") can be included in the force field.

2. The particle-mesh Ewald (PME) procedure is used to handle long-range electrostatic interactions. Long-range van der Waals interactions are estimated by a continuum model. Biomolecular simulations in the NVE ensemble (*i.e.* with Newtonian dynamics) conserve energy well over multi-nanosecond runs without modification of the equations of motion.

3. Translational periodic (P1) symmetry is supported, but no other space-group symmetry is. The size of the unit cell can be coupled to a given external pressure, and velocities can be coupled to a given external temperature by several schemes. The external conditions and coupling constants can be varied over time, so various simulated annealing protocols can be specified in a simple and flexible manner.

4. It is also possible to carry out non-periodic simulations in which aqueous solvation effects are represented *implicitly* by a generalized Born/ surface area model by adding the following two terms to the "vacuum" potential function:

$$\Delta G_{sol} = \sum_{ij}(1-\frac{1}{\varepsilon})(q_i q_j/f_{GB}(r_{ij}) + A\sum_i \sigma_i$$

   The first term accounts for the polar part of solvation (free) energy, designed to provide an approximation for the reaction field potential, and the second represents the non-polar contribution which is taken to be proportional to the surface area of the molecule.

5. Users can define internal restraints on bonds, valence angles, and torsions, and the force constants and target values for the restraints can vary during the simulation. The relative weights of various

terms in the force field can be varied over time, allowing one to implement a variety of simulated annealing protocols in a single run.

6. Internal restraints can be defined to be "time-averaged", that is, restraint forces are applied based on the averaged value of an internal coordinate over the course of the dynamics trajectory, not only on its current value. Alternatively, restraints can be "ensemble-averaged" using the locally-enhanced-sampling (LES) option.

7. Restraints can be directly defined in terms of NOESY intensities (calculated with a relaxation matrix technique), residual dipolar couplings, scalar coupling constants and proton chemical shifts. There are provisions for handling overlapping peaks or ambiguous assignments. In conjunction with distance and angle constraints, this provides a powerful and flexible approach to NMR structural refinements.

8. Xray and cryoEM refinements can be carried out in reciprocal space, with GPU-accelerated structure factor calculations. Real-space electron density maps can also be used as restraints for refinement.

9. Replica exchange calculations can allow simultaneous sampling at a variety of conditions (such as temperature), and allow the user to construct Boltzmann samples in ways that converge more quickly than standard MD simulations. Other variants of biased MD simulations can also be used to improve sampling.

10. Restraints can also be defined in terms of the root-mean-square coordinate distance from some reference structure. This allows one to bias trajectories either towards or away from some target. Free energies can be estimated from non-equilibrium simulations based on targetting restraints.

11. Free energy calculations, using thermodynamic integration (TI) with a linear or non-linear mixing of the "unperturbed" and "perturbed" Hamiltonian, can be carried out. Alternatively, potentials of mean force can be computed using umbrella sampling.

2.2 Main changes from the *sander* code in AmberTools

1. Some pieces are missing from the sander program in AmberTools:
   a) Things that should be easy to re-introduce later: emil, sebomd, pbsa, APBS, the charge-relocation option.
   b) Things are are problably gone for good, but which don't represent the best current practice: Path-integral methods, thermostats that don't follow the "middle" scheme, Berendsen barostat
   c) Things that might be useful, but really complicate the code: EVB potentials, QM/MM, nudged elastic band, constant pH and constant redox potential simulations. The API interface to Fortran, C and Python codes has also been removed.
   d) Non-periodic 3D-RISM has been removed for now, in an attempt to get the simplest possible RISM code, perhaps as a basis for future GPU work.

2. Key pieces of code that are still there, and being emphasized:
   a) Periodic and non-periodic simulations, with all of Amber's GB models
   b) 3D-RISM in periodic boundary conditions
   c) NMR, cryoEM and Xray restraints (including quite a bit of new code; Xray restraints include NVIDIA GPU-enabled capabilities)
   d) Thermodynamic integration and non-equilibrium sampling methods

e) Sampling and minimization using the lmod and xmin approaches; these can now be used in conjunction with SHAKE and SETTLE.

f) Replica exchange capabilities, except for constant pH and redox potential simulations

2.3 File usage

```
msander [-help] [-O] [-A] -i mdin -o mdout -p prmtop -c inpcrd -r restrt
-ref refc -mtmd mtmd -x mdcrd -y inptraj -v mdvel -frc mdfrc -e mden
-inf mdinfo
-O  Overwrite output files if they exist.
-A  Append output files if they exist (used mainly for replica exchange).
```

Here is a brief description of the files referred to above; the first five files are used for every run, whereas the remainder are only used when certain options are chosen.

mdin *input* control data for the min/md run

mdout *output* user readable state info and diagnostics -o stdout will send output to stdout (to the terminal) instead of to a file.

mdinfo *output* latest mdout-format energy info

prmtop *input* molecular topology, force field, periodic box type, atom and residue names

inpcrd *input* initial coordinates and (optionally) velocities and periodic box size

refc *input* (optional) reference coords for position restraints; also used for targeted MD

mtmd *input* (optional) containing list of files and parameters for targeted MD to multiple targets

mdcrd *output* coordinate sets saved over trajectory

inptraj *input* coordinate sets in trajectory format, when imin=5

mdvel *output* velocity sets saved over trajectory

mdfrc *output* force sets saved over trajectory

mden *output* extensive energy data over trajectory (not synchronized with mdcrd or mdvel)

restrt *output* final coordinates, velocity, and box dimensions if any - for restarting run

suffix *output* this string will be added to all unspecified output files that are printed (for *multisander* runs, it will append this suffix to all output files)

2.4 Example input files

Here are a couple of sample files, just to establish a basic syntax and appearance. There are more examples of NMR-related files later in this chapter.

1. Simple restrained minimization

```
Minimization with Cartesian restraints
&cntrl
imin=1, maxcyc=200, (invoke minimization)
ntpr=5, (print frequency)
ntr=1, (turn on Cartesian restraints)
restraint_wt=1.0, (force constant for restraint)
restraintmask=':1-58', (atoms in residues 1-58 restrained)
/
```

2. "Plain" molecular dynamics run

```
molecular dynamics run
&cntrl
imin=0, irest=1, ntx=5, (restart MD)
ntt=3, temp0=300.0, gamma_ln=5.0, (temperature control)
ntp=1, taup=2.0, (pressure control)
ntb=2, ntc=2, ntf=2, (SHAKE, periodic bc.)
nstlim=500000, (run for 0.5 nsec)
ntwx=1000, ntpr=200, (output frequency)
```

2.5 Namelist Input Syntax

Namelist provides list-directed input, and convenient specification of default values, and is a part of the Fortran 90 standard, Namelist input groups take the form:

```
&name
var1=value, var2=value, var3(sub)=value,
var4(sub,sub,sub)=value,value,
var5=repeat*value,value,
/
```

The variables must be names in the Namelist variable list. The order of the variables in the input list is of no significance, except that if a variable is specified more than once, later assignments may overwrite earlier ones. Blanks may occur anywhere in the input, except embedded in constants (other than string constants, where they count as ordinary characters).

It is common in older inputs for the ending "/" to be replaced by "&end"; this is non-standard-conforming.

Letter case is ignored in all character comparisons, but case is preserved in string constants. String constants must be enclosed by single quotes (’). If the text string itself contains single quotes, indicate them by two consecutive single quotes, e.g. C1’ becomes ’C1”’ as a character string constant.

Array variables may be subscripted or unsubscripted. An unsubscripted array variable is the same as if the subscript (1) had been specified. If a subscript list is given, it must have either one constant, or exactly as many as the number in the declared dimension of the array. Bounds checking is performed for ALL subscript positions, although if only one is given for a multi-dimension array, the check is against the entire array size, not against the first dimension. If more than one constant appears after an array assignment, the values go into successive locations of the array. It is NOT necessary to input all elements of an array.

Any constant may optionally be preceded by a positive (1,2,3,..) integer repeat factor, so that, for example, 25*3.1415 is equivalent to twenty-five successive values 3.1415. The repeat count separator, *, may be preceded and followed by 0 or more blanks. Valid LOGICAL constants are 0, F, .F., .FALSE., 1, T, .T., and .TRUE.; lower case versions of these also work.

2.6 Overview of the information in the input file

General minimization and dynamics input One or more title lines, followed by the (required) &cntrl and (optional) &pb, &ewald, &qmmm, &amoeba or &debugf namelist blocks. Described in Sections 2.7 and 2.8.

Varying conditions Parameters for changing temperature, restraint weights, etc., during the MD run. Each parameter is specified by a separate &wt namelist block, ending with &wt type="END", /. Described in Section 2.9.

File redirection TYPE=*filename* lines. Section ends with the first non-blank line which does not correspond to a recognized redirection. Described in Section 2.10.

Group information Read if *ntr*, *ibelly* or *idecomp* are set to nonzero values, and if some other conditions are satisfied; see sections on these variables, below. Described in Appendix 6.2.

2.7 General minimization and dynamics parameters

Each of the variables listed below is input in a namelist statement with the namelist identifier &cntrl.cmmu can enter the parameters in any order, using keyword identifiers. Variables that are not given in the namelist input retain their default values. Support for namelist input is included in almost all current Fortran compilers, and is a standard feature of Fortran 90. A detailed description of the namelist convention is given in Appendix A.

In general, namelist input consists of an arbitrary number of comment cards, followed by a record whose first seven characters after a " &" (e.g. " &cntrl ") name a group of variables that can be set by name.cmsys is followed by statements of the form " maxcyc=500, diel=2.0, ... ", and is concluded by an " / " token. The first line of input contains a title, which is then followed by the &cntrl namelist. Note that the first character on each line of a namelist block must be a blank.

Some of the options and variables are much more important, and commonlycmrdified, than are others. We have denoted the "common" options by printing them in **boldface** below. In general, you can skip reading about the non-bold options on a first pass, and you should change these from their defaults only if you think you know what you are doing.

2.7.1 General flags describing the calculation

imin Flag to run minimization.

= 0 (default) Run molecular dynamics without any minimization.

= 1 Perform an energy minimization.

= 5 Read in a trajectory for analysis.

Although sander will write energy information in the output files (using *ntpr*), it is often desirable to calculate the energies of a set of structures at a later point. In particular, one may wish to post-process a set of structures using a different energy function than was used to generate the structures. An example of this is MM-PBSA analysis, where the explicit water is removed and replaced with a continuum model.

If *imin* is set to 5, sander will read a trajectory file (the “inptraj” argument, specified using `-y` on the command line), and will perform the functions described in the mdin file (e.g., an energy minimization) for each of the structures in this file. The final structure from each minimization will be written out to the normal mdcrd file. If you wish to read in a binary (i.e., NetCDF format) trajectory, be sure to set *ioutfm* to 1 (see below). Note that this will result in the output trajectory having NetCDF format as well.

For example, when *imin* = 5 and *maxcyc* = 1000, sander will minimize each structure in the trajectory for 1000 steps and write a minimized coordinate set for each frame to the mdcrd file. If *maxcyc* = 1, the output file can be used to extract the energies of each of the coordinate sets in the inptraj file.

Trajectories containing box coordinates can be post-processed. In order to read trajectories with box coordinates, *ntb* should be greater than 0.

IMPORTANT CAVEAT: The initial coordinates input file used (`-c` <inpcrd>) should be the same as the initial coordinates input file used to generate the original trajectory. This is because sander sets up parameters for PME from the box coordinates in the initial coordinates input file.

nmropt **= 0** (default) No nmr-type analysis will be done.

= 1 NMR restraints and weight changes will be read.

= 2 NMR restraints, weight changes, NOESY volumes, chemical shifts and residual dipolar restraints will be read.

2.7.2 Nature and format of the input

ntx Option to read the initial coordinates, velocities, and box size from the inpcrd file. Option 1 must be used when one is starting from minimized or model-built coordinates. If an MD restrt file is specified for inpcrd then option 5 is generally used (unless you explicitly wish to ignore the velocities that are present).

= 1 (default) Coordinates, but no velocities, will be read; either formatted (ASCII) files or NetCDF files can be used, as the input file type will be auto-detected.

= 5 Coordinates and velocities will be read from either a NetCDF or a formatted (ASCII) coordinate file. Box information will be read if $ntb > 0$. The velocity information will only be used if *irest* = 1 (see below).

irest Flag to restart a simulation.

= 0 (default) Do not restart the simulation; instead, run as a new simulation. Velocities in the input coordinate file, if any, will be ignored, and the time step count will be set to 0 (unless overridden by *t*; see below).

= 1 Restart the simulation, reading coordinates and velocities from a previously saved restart file. The velocity information is necessary when restarting, so *ntx* (see above) must be 5 (for Amber versions much older than 20, ntx must be greater than or equal to 4), if *irest* = 1.

2.7.3 Nature and format of the output

ntxo Format of the final coordinates, velocities, and box size (if a constant volume or pressure run) written to file "restrt".

= 1 Formatted (ASCII)

= 2 (default) NetCDF file (recommended, unless you have a workflow that requires the formatted form.)

ntpr Every *ntpr* steps, energy information will be printed in human-readable form to files "mdout" and "mdinfo". "mdinfo" is closed and reopened each time, so it always contains the most recent energy and temperature. Default 50.

ntwr Every *ntwr* steps during dynamics, the “restrt” file will be written, ensuring that recovery from a crash will not be so painful. No matter what the value of *ntwr*, a restrt file will be written at the end of the run, i.e., after *nstlim* steps (for dynamics) or *maxcyc* steps (for minimization). If $ntwr < 0$, a unique copy of the file, “restrt_<nstep>”, is written every abs(*ntwr*) steps. This option is useful if for example one wants to run free energy perturbations from multiple starting points or save a series of restrt files for minimization. Default = nstlim.

ntwx Every *ntwx* steps, the coordinates will be written to the mdcrd file. If *ntwx* = 0, no coordinate trajectory file will be written. Default = 0.

ntwv Every *ntwv* steps, the velocities will be written to the mdvel file. If *ntwv* = 0, no velocity trajectory file will be written. If *ntwv* = -1, velocities will be written to mdcrd, which then becomes a combined coordinate/velocity trajectory file, at the interval defined by *ntwx*. This option is available only for binary NetCDF output (*ioutfm* = 1). Most users will have no need for a velocity trajectory file and so can safely leave *ntwv* at the default. Default = 0. Note that dumping velocities frequently, like forces or coordinates, will introduce potentially significant I/O and communication overhead, hurting both performance and parallel scaling.

`ionstepvelocities` Controls whether to print the half-step-ahead velocities (0, default) or on-step velocities (1). The half-step-ahead velocities can potentially be used to restart calculations, but the on-step velocities correspond to calculated kinetic energy/temperature.

`ntwf` Every *ntwf* steps, the forces will be written to the mdfrc file. If *ntwf* = 0, no force trajectory file will be written. If *ntwf* = -1, forces will be written to the mdcrd, which then becomes a combind coordinate/force trajectory file, at the interval defined by *ntwx*. This option is available only for binary NetCDF output (*ioutfm* = 1). Most users will have no need for a force trajectory file and so can safely leave *ntwf* at the default. Default = 0. Note that dumping forces frequently, like velocities or coordinates, will introduce potentially significant I/O and communication overhead, hurting both performance and parallel scaling.

ntwe Every *ntwe* steps, the energies and temperatures will be written to file "mden" in a compact form. If *ntwe* = 0 then no mden file will be written. Note that energies in the mden file are not synchronized with coordinates or velocities in the mdcrd or mdvel file(s). Assuming identical *ntwe* and *ntwx* values the energies are one time step before the coordinates (as well as the velocities which are synchronized with the coordinates). Consequently, an mden file is rarely written. Default = 0.

ioutfm The format of coordinate and velocity trajectory files (mdcrd, mdvel and inptraj). As of Amber 9, the binary format used in previous versions is no longer supported; binary output is now in NetCDF trajectory format. Binary trajectory files have many advantages: they are smaller, higher precision, much faster to read and write, and able to accept a wider range of coordinate (or velocity) values than formatted trajectory files.

= 0 Formatted ASCII trajectory

= 1 (default) Binary NetCDF trajectory

ntwprt The number of atoms to include in trajectory files (mdcrd and mdvel). This flag can be used to decrease the size of the these files, by including only the first part of the system, which is usually of greater interest (for instance, one might include only the solute and not the solvent). If *ntwprt* = 0, all atoms will be included.

= 0 (default) Include all atoms of the system when writing trajectories.

> 0 Include only atoms 1 to *ntwprt* when writing trajectories.

2.7.4 Frozen or restrained atoms

ibelly — Flag for belly type dynamics. If set to 1, a subset of the atoms in the system will be allowed to move, and the coordinates of the rest will be frozen. The *moving* atoms are specified with *bellymask*. This option is not available when *igb>0*. When belly type dynamics is in use, bonded energy terms, vdW interactions, and direct space electrostatic interactions are *not calculated* for pairs of frozen atoms. Note that this does *not* provide any significant speed advantage. Freezing atoms can be useful for some applications but is maintained primarily for backwards compatibility with older versions of Amber. Most applications should use the *ntr* variable instead to restrain parts of the system to stay close to some initial configuration. Default = 0.

ntr — Flag for restraining specified atoms in Cartesian space using a harmonic potential, if ntr > 0. The restrained atoms are determined by the *restraintmask* string. The force constant is given by *restraint_wt*. The coordinates are read in "restrt" format from the "refc" file. Default = 0.

restraint_wt — The weight ($\text{kcal} \cdot \text{mol}^{-1} \cdot \text{Å}^{-2}$) for the positional restraints. The restraint is of the form $k(\Delta x)^2$, where k is the value given by this variable, and Δx is the difference between one of the Cartesian coordinates of a restrained atom and its reference position. There is a term like this for each Cartesian coordinate of each restrainted atom.

restraintmask — String that specifies the *restrained* atoms when *ntr=1*.

bellymask — String that specifies the *moving* atoms when *ibelly=1*.

The syntax for both *restraintmask* and *bellymask* is given in Section 6.1.1. Note that these mask strings are limited to a maximum of 256 characters.

2.7.5 Energy minimization

`maxcyc` — The maximum number of cycles of minimization. Default = 1.

`ntmin` — Flag for the method of minimization.

= 3 The XMIN method is used, see Section 7.4.1. (default)

= 4 The LMOD method is used, see Section 7.4.2.

`dx0` — The initial step length. If the initial step length is too big then will give a huge energy; however the minimizer is smart enough to adjust itself. Default 0.01.

`drms` — The convergence criterion for the energy Derivative: minimization will halt when the Root-Mean-Square of the Cartesian elements of the gradient of the energy is less than this. Default is $10^{-4}\text{kcal} \cdot \text{mol}^{-1} \cdot \text{Å}^{-1}$.

2.7.6 Molecular dynamics

The “middle” scheme offers a unified framework to develop efficient thermostatting algorithms for configurational sampling for the canonical ensemble, as described in Refs. [1–5]. It is the only MD scheme implemented in *msander*. It can be implemented for performing molecular dynamics (MD) or path integral molecular dynamics (PIMD), either with or without holonomic constraints. The “middle” scheme allows the use of much larger time intervals (i.e., time stepsizes) Δt to maintain the same accuracy, which significantly improves the configurational sampling efficiency. That is, it is efficient for calculating structural properties and thermodynamic observables that depend on coordinate variables. Most thermostats control the temperature by updating momenta of the system. Some prevailing thermostats include stochastic ones (such as the Andersen thermostat and Langevin dynamics) and deterministic ones (such as the Nosé-Hoover thermostat and Nosé-Hoover chain). In the "middle" scheme, immediately after the coordinate-updating step for half a time interval, the thermostat process for a full time interval takes place, which is then followed by the coordinate-updating step for another half time interval[1, 5].

`nstlim` Number of MD-steps to be performed. Default 1.

`nscm` Flag for the removal of translational and rotational center-of-mass (COM) motion at regular intervals (default is 1000). For non-periodic simulations, after every NSCM steps, translational and rotational motion will be removed. For periodic systems, just the translational center-of-mass motion will be removed. This flag is ignored for belly simulations.

For Langevin dynamics, the *position* of the center-of-mass of the molecule is reset to zero every NSCM steps, but the velocities are not affected. Hence there is no change to either the translation or rotational components of the momenta. (Doing anything else would destroy the way in which temperature is regulated in a Langevin dynamics system.) The only reason to even reset the coordinates is to prevent the molecule from diffusing so far away from the origin that its coordinates overflow the format used in restart or trajectory files.

`t` The time at the start (psec) this is for your own reference and is not critical. Start time is taken from the coordinate input file if IREST=1. Default 0.0.

`dt` The time step (psec). Recommended MAXIMUM is .002 if SHAKE is used, or .001 if it isn't. Note that for temperatures above 300K, the step size should be reduced since greater temperatures mean increased velocities and longer distance traveled between each force evaluation, which can lead to anomalously high energies and system blowup. Default 0.001.
The use of Hydrogen Mass Repartitioning (HMR) (see [6] and references therein for more information), together with SHAKE, allows the time step to be increased in a stable fashion by about a factor of two (up to .004) by slowing down the high frequency hydrogen motion in the system. To use HMR, the masses in the topology file need to be altered before starting the simulation. ParmEd can do this automatically with the HMassRepartition option; see Section **??** .

`nrespa` This variable allows the user to evaluate slowly-varying terms in the force field less frequently. For PME, "slowly-varying" (now) means the reciprocal sum. For generalized Born runs, the "slowly-varying" forces are those involving derivatives with respect to the effective radii, and pair interactions whose distances are greater than the "inner" cutoff, currently hard-wired at 8 Å. If NRESPA$>$1 these slowly-varying forces are evaluated every *nrespa* steps. The forces are adjusted appropriately, leading to an impulse at that step. If *nrespa*dt* is less than or equal to 4 fs then the energy conservation is not seriously compromised. However if *nrespa*dt* $>$ 4 fs then the simulation becomes less stable. Note that energies and related quantities are only accessible every *nrespa* steps, since the values at other times are meaningless.

2.7.7 Temperature regulation

Note: Flag "ntt" is used for the temperature regulation in the default thermostat scheme as shown below. The "middle" thermostat scheme [Section **??**] is much more efficient than the default scheme to accurately sample the configuration/conformation space in the molecular dynamics simulation for the NVT ensemble. Please read Section **??** for more details.

ntt Switch for temperature scaling. Note that setting *ntt*=0 corresponds to the microcanonical (NVE) ensemble (which should approach the canonical one for large numbers of degrees of freedom). Some aspects of the "weak-coupling ensemble" (*ntt*=1) have been examined, and roughly interpolate between the microcanonical and canonical ensembles.[7, 8] The *ntt*=2 and 3 options correspond to the canonical (constant T) ensemble.

= 0 Constant total energy classical dynamics (assuming that *ntb*<2, as should probably always be the case when *ntt=0*).

= 3 Use Langevin dynamics with the collision frequency γ given by *gamma_ln*, discussed below. Note that when γ has its default value of zero, this is the same as setting *ntt* = 0. Since Langevin simulations are highly susceptible to "synchronization" artifacts,[9, 10] you should explicitly set the *ig* variable (described below) to a different value at each restart of a given simulation. (Setting $ig = -1$ will randomize this starting seed, based on the system clock.)

= 10 Stochastic Isokinetic Nose-Hoover RESPA integrator [11]. A novel isokinetic integrator developed by Tuckerman and co-workers that invokes an isokinetic constraint on the particle velocities combined with *nkija* (see below) auxiliary thermostat velocities *v1* and *v2*. The integrator includes a stochastic component in the equations of motion, which introduces white noise into the system, for the purpose of minimizing resonance instabilities in the velocities, ultimately allowing for larger RESPA steps. The isokinetic constraint has the form $mv^2 + \frac{L}{L+1}\sum_{i=1}^{L} Q_1 v_{1i}^2 = Lk_BT$. Here L is the number of additional thermostat degrees of freedom, defined in AMBER as *nkija* (see below), and $Q_1$is the thermostat mass, determined from *sinrtau* (below), v is the particle velocity and v_1 is one of two auxiliary velocities (e.g. thermostat velocities), and m, k_B, and T, are the particle mass, Boltzmann constant, and system temperature (*temp0*), respectively. In using this integrator, the system is placed in the isokinetic ensemble, as such the velocities are NOT canonical and no thermodynamic observables can be derived from them. This will lead to anomalous temperature readings throughout the simulation - for 1 thermostat degree of freedom (L = *nkija* = 1) the temperature will appear about one-half the specified temperature (*temp0*), and with additional thermostat DOF, the temperature will approach, but never exceed, the desired temperature, *temp0.* However, the particle coordinates ARE canonical and it can be said the configurations obtained from a simulation were sampled from a Boltzmann distribution at the specified temperature (*temp0*).

temp0 Reference temperature at which the system is to be kept, if $ntt > 0$. Note that for temperatures above 300K, the step size should be reduced since increased distance traveled between evaluations can lead to SHAKE and other problems. Default 300.

tempi Initial temperature. For the initial dynamics run, (*ntx* = 1 or for Amber versions much older than 20, ntx $<$ 3) the velocities are assigned from a Maxwellian distribution at *tempi* K. If *tempi* = 0.0, the velocities will be calculated from the forces instead. *tempi* has no effect if *ntx* = 5 (for Amber versions much older than 20, if ntx $>$ 3). Default 0.0.

ig The seed for the pseudo-random number generator. The MD starting velocity is dependent on the random number generator seed if *tempi* is nonzero and *ntx* = 1 (for Amber versions much older than 20, if ntx $<$ 3). The value of this seed also affects the set of pseudo-random values used for Langevin dynamics or Andersen-like coupling, and hence should be set to a different value on each restart if *ntt* = 2 or 3. If *ig* = -1 (the default) then the random seed will be based on the current date and time, and hence will be different for every run. Unless you specifically desire reproducibility, it is recommended that you set *ig* = -1 for all runs involving *ntt* = 2 or 3.

gamma_ln The collision frequency γ, in ps^{-1}, when *ntt* = 3. Default is 0. A. See Section **??** for details. Note that it is not necessary that γ approximate the physical collision frequency, which is about 50 ps^{-1} for liquid water. In fact, it is often advantageous, in terms of sampling[12, 13] or stability of integration[14], to use much smaller values, around 2 to 5 ps^{-1}.[12, 14] For implicit solvent (GB), even much lower values may be useful: for example, setting gamma_ln to 0.01 ps^{-1} can lead to significant, up to 100-fold in some cases, speedup of conformational sampling.[15]

The recommended value for *gamma_ln* is related to the characteristic frequency ($\tilde{\omega}$) of the specific system. The characteristic time of the potential energy autocorrelation function is

$$\tau_{UU} = \int_0^\infty \frac{\langle U(0)U(t)\rangle - \langle U\rangle^2}{\langle U^2\rangle - \langle U\rangle^2}\,\mathrm{d}t \tag{2.1}$$

The optimal value of the thermostat parameter that produces the minimum correlation time of the potential is $\xi^{opt} \approx \tilde{\omega}$ for Langevin dynamics and $\xi^{opt} \approx \sqrt{2}\tilde{\omega}$ for the Andersen thermostat, as the time interval Δt approaches zero. E.g. for a HO molecule, the frequency of the O-H stretch is around 3600 cm^{-1}(680 ps^{-1}), so one can choose 680 ps^{-1} as the value of *therm_par* when Langevin dynamics is used, or 960 ps^{-1} when the Andersen thermostat is employed. When the time interval Δt is finite in the two thermostatting methods, while the characteristic correlation time goes to infinity as the thermostat parameter approaches zero, the characteristic correlation time gradually reaches a plateau as the thermostat parameter increases. (Please see Refs. [3–5] for more discussion.) When condensed phase systems are simulated, it is not straightforward to estimate the optimal thermostat parameter(s) that could be related to the mixing of frequencies (or time scales) of the system [16]. Some numerical tests are necessary for obtaining the reasonable region for the thermostat parameter such that the characteristic time divided by the time interval is relatively small. (This is true not only for the “middle” scheme but for all thermostat algorithms.) For a liquid water system (216 water molecules in a cell with periodic boundary conditions) with no holonomic constraints, the thermostat parameter is usually chosen to be $2 - 50$ ps^{-1}.

vlimit If not equal to 0.0, then any component of the velocity that is greater than abs(VLIMIT) will be reduced to VLIMIT (preserving the sign). This can be used to avoid occasional instabilities in molecular dynamics runs. VLIMIT should generally be set to a value like 20 (the default), which is well above the most probable velocity in a Maxwell-Boltzmann distribution at room temperature. A warning message will be printed whenever the velocities are modified. Runs that have more than a few such warnings should be carefully examined.

nkija For use with *ntt=10*. This specifies the number of additional auxiliary velocity variables *v1* and *v2*, which will total nkija$\times$*v1* +nkija$\times$*v2* *[11].* Default is 1.

sinrtau For the SINR (Stochastic Isokinetic Nose-Hoover RESPA) integrator (*ntt=10*), this specifies the time scale for determining the masses associated with the two auxiliary velocity variables *v1* and *v2* (e.g. thermostat velocities) and hence the magnitude of the coupling of the physical velocities with the auxiliary velocities. Generally this should be related to the time scale of the system. See [11] for more explanation. Default is 1.0.

2.7.8 Pressure regulation

In "constant pressure" dynamics, the volume of the unit cell is adjusted (by small amounts on each step) to make the computed pressure approach the target pressure, *pres0*. Equilibration with $ntp > 0$ is generally necessary to adjust the density of the system to appropriate values. Pressure regulation only applies when Constant Pressure periodic boundary conditions are used (*ntp > 0*). The Monte Carlo barostat samples rigorously from the isobaric-isothermal ensemble and does not necessitate computing the virial. Please note: in general you will need to equilibrate the temperature to something like the final temperature using constant volume (*ntp*=0) *before* switching on constant pressure simulations to adjust the system to the correct density. If you fail to do this, the program will try to adjust the density too quickly, and bad things (such as SHAKE failures) are likely to happen.

ntp Flag for constant pressure dynamics. This option should be set to 1 or 2 when Constant Pressure periodic boundary conditions are used.

- **= 0** No pressure scaling (Default)
- **= 1** md with isotropic position scaling

barostat Flag used to control which barostat to use in order to control the pressure.

- **= 2** Monte Carlo barostat

mcbarint Number of steps between volume change attempts performed as part of the Monte Carlo barostat. Default is 100.

pres0 Reference pressure (in units of bars, where 1 bar $\approx$ 0.987 atm) at which the system is maintained (when NTP $>$ 0). Default 1.0.

Surface tension regulation

Constant surface tension is used in statistical ensembles for simulating liquid interfaces. This is primarily intended for lipid membrane simulations with two or more interfaces. Constant surface tension is only available for simulations with anisotropic pressure or semiisotropic scaling. This algorithm is an extension to the Berendsen pressure scaling algorithm that adjusts the tangential pressure evaluation in order to maintain a “constant” surface tension.[17] Since the surface tension is a function of the pressure tensor, fluctuations of the surface tension will be large.

In order to use constant surface tension, periodic boundary conditions (ntb = 2), anisotropic or semi-isotropic pressure scaling (ntp = 2 or ntp =3), and an orthogonal box must be used.

csurften Flag for constant surface tension dynamics.

- **= 0** No constant surface tension (default)
- **= 1** Constant surface tension with interfaces in the yz plane
- **= 2** Constant surface tension with interfaces in the xz plane
- **= 3** Constant surface tension with interfaces in the xy plane

gamma_ten Surface tension value in units of dyne/cm. Default value is 0.0 dyne/cm.

ninterface Number of interfaces in the periodic box. There must be at least two interfaces in the periodic box. Two interfaces is appropriate for a lipid bilayer system and is the default value.

2.7.9 SHAKE bond length constraints

ntc Flag for SHAKE to perform bond length constraints.[18] (See also NTF in the **Potential function** section. In particular, typically NTF = NTC.) The SHAKE option should be used for most MD calculations. The size of the MD timestep is determined by the fastest motions in the system. SHAKE removes the bond stretching freedom, which is the fastest motion, and consequently allows a larger timestep to be used. For water models, a special "three-point" algorithm is used.[19] Consequently, to employ TIP3P set NTF = NTC = 2.

Since SHAKE is an algorithm based on dynamics, the minimizer is not aware of what SHAKE is doing; for this reason, minimizations generally should be carried out without SHAKE. One exception is short minimizations whose purpose is to remove bad contacts before dynamics can begin.

For parallel versions of *sander* only intramolecular atoms can be constrained. Thus, such atoms must be in the same chain of the originating PDB file.

= 1 SHAKE is not performed (default)

= 2 bonds involving hydrogen are constrained

= 3 all bonds are constrained (not available for parallel or qmmm runs in *sander*)

`tol` Relative geometrical tolerance for coordinate resetting in shake. Recommended maximum: <0.00005 Angstrom Default 0.00001.

`jfastw` Fast water definition flag. By default, the system is searched for water residues, and special routines are used to SHAKE these systems.[19]

= 0 Normal operation. Waters are identified by the default names (given below), unless they are redefined, as described below.

= 4 Do not use the fast SHAKE routines for waters.

The following variables allow redefinition of the default residue and atom names used by the program to determine which residues are waters.

WATNAM The residue name the program expects for water. Default 'WAT '.

OWTNM The atom name the program expects for the oxygen of water. Default 'O '.

HWTNM1 The atom name the program expects for the 1st H of water. Default 'H1 '.

HWTNM2 The atom name the program expects for the 2nd H of water. Default 'H2 '.

`noshakemask` String that specifies atoms that are not to be shaken (assuming that ntc>1). Any bond that would otherwise be shaken by virtue of the *ntc* flag, but which involves an atom flagged here, will *not* be shaken. The syntax for this string is given in Chap. 13.5. Default is an empty string, which matches nothing. A typical use would be to remove SHAKE constraints from all or part of a solute, while still shaking rigid water models like TIPnP or SPC/E. Another use would be to turn off SHAKE constraints for the parts of the system that are being changed with thermodynamic integration, or which are the EVB or quantum regions of the system.

If this option is invoked, then all parts of the potential must be evaluated, that is, *ntf* must be one. The code enforces this by setting *ntf* to 1 when a *noshakemask* string is present in the input.

If you want the *noshakemask* to apply to all or part of the water molecules, you must also set *jfastw=4*, to turn off the special code for water SHAKE. (If you are not shaking waters, you presumably also want to issue the "set default FlexibleWater on" command in LEaP; see that chapter for more information.)

2.7.10 NMR refinement options

(Users to should consult the section NMR refinement to see the context of how the following parameters would be used.)

`iscale` Number of additional variables to optimize beyond the 3N structural parameters. (Default = 0). At present, this is only used with residual dipolar coupling and CSA or pseudo-CSA restraints.

`noeskp` The NOESY volumes will only be evaluated if mod(nstep, noeskp) = 0; otherwise the last computed values for intensities and derivatives will be used. (default = 1, i.e. evaluate volumes at every step) `ipnlty` This parameter determines the the functional form of the penalty function for NOESY volume and chemical shift restraints.

- **= 1** the program will minimize the sum of the absolute values of the errors; this is akin to minimizing the crystallographic R-factor (default).
- **= 2** the program will optimize the sum of the squares of the errors.
- **= 3** For NOESY intensities, the penalty will be of the form $awt[I_c^{1/6} - I_o^{1/6}]^2$. Chemical shift penalties will be as for *ipnlty=1*.

`mxsub` Maximum number of submolecules that will be used. This is used to determine how much space to allocate for the NOESY calculations. Default 1.

`scalm` "Mass" for the additional scaling parameters. Right now they are restricted to all have the same value. The larger this value, the slower these extra variables will respond to their environment. Default 100 amu.

`pencut` In the summaries of the constraint deviations, entries will only be made if the penalty for that term is greater than PENCUT. Default 0.1.

`tausw` For noesy volume calculations (*NMROPT* = *2*), intensities with mixing times less that TAUSW (in seconds) will be computed using perturbation theory, whereas those greater than TAUSW will use a more exact theory. See the theory section (below) for details. To always use the "exact" intensities and derivatives, set TAUSW = 0.0; to always use perturbation theory, set TAUSW to a value larger than the largest mixing time in the input. Default is TAUSW of 0.1 second, which should work pretty well for most systems.

2.7.11 EMAP restraints

EMAP restraints are used to perform targeted conformational search (TCS)[20]. EMAP uses maps to define restraints to maintain conformations and/or to induce simulation systems to the target conformations. The restraint map can be either obtained from electron microscopy experiments or derived from known protein structures, or defined from initial simulation coordinates. EMAP can be used to do rigid docking of molecules into maps and to do flexible fitting to obtain conformations defined by experimental maps. EMAP can also be used to maintain conformations of protein domains when studying large scale conformational change. Users should consult the section 10.1 to see how to define EMAP restraints.

`iemap` Turn on EMAP restrained simulation when *iemap*>0. (Default = 0). EMAP restraint information must be input from *&emap* namelists in the input file.

`gammamap` Friction constant for the EMAP restraint maps when allowed to move. (Default=1/ps). (See Section 10.1)

2.8 Potential function parameters

The parameters in this section generally control what sort of force field (or potential function) is used for the simulation.

2.8.1 Generic parameters

`ntf` Force evaluation. Note: If SHAKE is used (see NTC), it is not necessary to calculate forces for the constrained bonds.

- **= 1** complete interaction is calculated (default)

= 2 bond interactions involving H-atoms omitted (use with NTC=2)

= 3 all the bond interactions are omitted (use with NTC=3)

= 4 angle involving H-atoms and all bonds are omitted

= 5 all bond and angle interactions are omitted

= 6 dihedrals involving H-atoms and all bonds and all angle interactions are omitted

= 7 all bond, angle and dihedral interactions are omitted

= 8 all bond, angle, dihedral and non-bonded interactions are omitted

ntb This variable controls whether or not periodic boundaries are imposed on the system during the calculation of non-bonded interactions. Bonds spanning periodic boundaries are not yet supported. There is no longer any need to set this variable, since it can be determined from igb and ntp parameters. The "proper" default for ntb is chosen (ntb=0 when igb > 0, ntb=2 when ntp > 0, and ntb=1 otherwise). This behavior can be overridden by supplying an explicit value, although this is discouraged to prevent errors. The allowed values for NTB are

= 0 no periodicity is applied and PME is off (default when igb > 0)

= 1 constant volume (default when igb and ntp are both 0, which are their defaults)

= 2 constant pressure (default when ntp > 0)

If NTB is nonzero then there must be a periodic boundary in the topology file. Constant pressure is not used in minimization (IMIN=1, above).

For a periodic system, constant pressure is the only way to equilibrate density if the starting state is not correct. For example, the solvent packing scheme used in LEaP can result in a net void when solvent molecules are subtracted which can aggregate into "vacuum bubbles" in a constant volume run. Another potential problem are small gaps at the edges of the box. The upshot is that almost every system needs to be equilibrated at constant pressure (*ntb=2, ntp>0*) to get to a proper density. But be sure to equilibrate first (at constant volume) to something close to the final temperature, before turning on constant pressure.

dielc Dielectric multiplicative constant for the electrostatic interactions. Default is 1.0. Please note this is NOT related to dielectric constants for generalized Born or Poisson-Boltzmann calculations. It should only be used for quasi-vacuum simulations.

cut This is used to specify the nonbonded cutoff, in Angstroms. For PME, the cutoff is used to limit direct space sum, and 8.0 is usually a good value. When *igb>0*, the cutoff is used to truncate nonbonded pairs (on an atom-by-atom basis); here a larger value than the default is generally required. A separate parameter (**RGBMAX**) controls the maximum distance between atom pairs that will be considered in carrying out the pairwise summation involved in calculating the effective Born radii, see the generalized Born section below.
When *igb > 0*, the default is 9999.0 (effectively infinite)
When *igb==0*, the default is 8.0.

fswitch When off, *fswitch<=0* , uses a truncation cutoff. When on *fswitch>0*, sets a force switching region where the force cutoff smoothly approaches 0 between the region of the fswitch value to the cut value. Force values below the fswitch value follow the standard Lennard-Jones force. Default is -1. This option is not supported for use with GB (i.e., only *igb=0* and *ntb>0*), nor is it compatible with the 12-6-4 Lennard-Jones model (*lj1264=1*). Due to performance regressions (about 20%) with running with the force switching on, it is recommended that simulations run with fswitch off unless using a force field that requires or recommends using the force switch.

`nsnb` Determines the frequency of nonbonded list updates when *igb=0* and *nbflag=0*; see the description of *nbflag* for more information. Default is 25.

`igb` Flag for using the generalized Born implicit solvent models. See Chapter 3 for information about using this option. Default is 0.

`irism` Flag for 3D-reference interaction site model (RISM) molecular solvation method. See Section4.3for information about this option. Default is 0.

`lj1264` In general, you should rarely have to set this variable. When the Lennard-Jones C-coefficient is found in your prmtop file, the default value is set to 1 (meaning it is active). When this flag is *not* present in the prmtop file, the default value is set to 0 (meaning the 12-6-4 potential [21] is inactive). Setting this to 0 when the C-coefficient is present will forcibly turn off the 12-6-4 potential. Setting lj1264 to 1 when no C-coefficient is present will result in a fatal error. Therefore, this flag can be used to quickly disable the r^{-4} term. However, the remaining L-J parameters will still be optimized for the 12-6-4 potential, so this should only be done when testing!.

2.8.2 Particle Mesh Ewald

The Particle Mesh Ewald (PME) method is always "on", unless *ntb* = 0. PME is a fast implementation of the Ewald summation method for calculating the full electrostatic energy of a unit cell (periodic box) in a macroscopic lattice of repeating images. The PME method is fast since the reciprocal space Ewald sums are B-spline interpolated on a grid and since the convolutions necessary to evaluate the sums are calculated via fast Fourier transforms (FFTs). Note that the accuracy of the PME method is related to the density of the charge grid (NFFT1, NFFT2, and NFFT3), the spline interpolation order (ORDER), and the direct sum tolerance (DSUM_TOL); see the descriptions below for more information.

The PME method was implemented originally in Amber 3a by Tom Darden and has been developed in subsequent versions by many people, in particular by Tom Darden, Celeste Sagui, Tom Cheatham and Mike Crowley.[22–25] Generalizations of this method to systems with polarizable dipoles and electrostatic multipoles are described in Refs. [26, 27].

The `&ewald` namelist is read immediately after the `&cntrl` namelist. We have tried hard to make the defaults for these parameters appropriate for solvated simulations. *Please take care in changing any values from their defaults.* The `&ewald` namelist has the following variables:

`nfft1, nfft2, nfft3` These give the size of the charge grid (upon which the reciprocal sums are interpolated) in each dimension. Higher values lead to higher accuracy (when the DSUM_TOL is also lowered) but considerably slow the calculation. Generally it has been found that reasonable results are obtained when NFFT1, NFFT2 and NFFT3 are approximately equal to A, B and C, respectively, leading to a grid spacing (A/NFFT1, etc.) of 1.0 Å. Significant performance enhancement in the calculation of the fast Fourier transform is obtained by having each of the integer NFFT1, NFFT2 and NFFT3 values be a *product of powers* of 2, 3, and/or 5. If the values are not given, the program will chose values to meet these criteria.

`order` The order of the B-spline interpolation. The higher the order, the better the accuracy (unless the charge grid is too coarse). The minimum order is 3. An order of 4 (the default) implies a cubic spline approximation which is a good standard value. Note that the cost of the PME goes as roughly the order to the third power.

`verbose` Standard use is to have VERBOSE = 0. Setting VERBOSE to higher values (up to a maximum of 3) leads to voluminous output of information about the PME run.

`dsum_tol` This relates to the width of the direct sum part of the Ewald sum, requiring that the value of the direct sum at the Lennard-Jones cutoff value (specified in CUT as during standard dynamics) be less than DSUM_TOL. In practice it has been found that the relative error in the Ewald forces (RMS) due to cutting off the direct sum at CUT is between 10.0 and 50.0 times DSUM_TOL. Standard values for DSUM_TOL are in the range of 10^{-6} to 10^{-5} , leading to estimated RMS deviation force errors of 0.00001 to 0.0005. Default is 10^{-5}.

`rsum_tol`	This serves as a way to generate the number of reciprocal vectors used in an Ewald sum. Typically the relative RMS reciprocal sum error is about 5-10 times RSUM_TOL. Default is 5 x 10^{-5}.
`mlimit(1,2,3)`	This allows the user to explicitly set the number of reciprocal vectors used in a regular Ewald run. Note that the sum goes from -MLIMIT(2) to MLIMIT(2) and -MLIMIT(3) to MLIMIT(3) with symmetry being used in first dimension. Note also the sum is truncated outside an automatically chosen sphere.
`ew_coeff`	Ewald coefficient, in Å^{-1} . Default is determined by *dsum_tol* and *cutoff*. If it is explicitly inputed then that value is used, and *dsum_tol* is computed from *ew_coeff* and *cutoff*.
`nbflag`	If *nbflag = 0*, construct the direct sum nonbonded list in the "old" way, *i.e.* update the list every *nsnb* steps. If *nbflag = 1* (the default when *imin = 0* or *ntb* > *0*), *nsnb* is ignored, and the list is updated whenever any atom has moved more than 1/2 *skinnb* since the last list update.
`skinnb`	Width of the nonbonded "skin". The direct sum nonbonded list is extended to *cut* + *skinnb*, and the van der Waals and direct electrostatic interactions are truncated at *cut*. Default is 2.0 Å. Use of this parameter is required for energy conservation, and recommended for all PME runs.
`netfrc`	The basic "smooth" PME implementation used here does not necessarily conserve momentum. If *netfrc = 1*, (the default) the total force on the system is artificially removed at every step. This parameter is set to 0 if minimization is requested, which implies that the gradient is an accurate derivative of the energy. You should only change this parameter if you really know what you are doing.
`vdwmeth`	Determines the method used for van der Waals interactions beyond those included in the direct sum. A value of 0 includes no correction; the default value of 1 uses a continuum model correction for energy and pressure.
`eedmeth`	Determines how the switch function for the direct sum Coulomb interaction is evaluated. The default value of 1 uses a cubic spline. A value of 2 implies a linear table lookup. A value of three implies use of an "exact" subroutine call.
`eedtbdns`	Density of spline or linear lookup table, if *eedmeth* is 1 or 2. Default is 500 points per unit.
`column_fft`	1 or 0 flag to turn on or off, respectively, column-mode fft for parallel runs. The default mode is slab mode which is efficient for low processor counts. The column method can be faster for larger processor counts since there can be more columns than slabs and the communications pattern is less congested. This flag has no effect on non-parallel runs. Users should test the efficiency of the method in comparison to the default method before performing long calculations. Default is 0 (off).

2.8.3 Extra point options

Several parameters deal with "extra-points" (sometimes called lone-pairs), which are force centers that are not at atomic positions. These are currently defined as atoms with "EP" in their names. These input variables are really only for the convenience of force-field developers; *do not change the defaults unless you know what you are doing, and have read the code.* These variables are set in the &ewald namelist.

`frameon` If *frameon* is set to 1, (default) the bonds, angles and dihedral interactions involving the lone pairs/extra points are removed except for constraints added during parm. The lone pairs are kept in ideal geometry relative to local atoms, and resulting torques are transferred to these atoms. To treat extra points as regular atoms, set frameon=0.

`chngmask` If *chngmask*=1 (default), new 1-1, 1-2, 1-3 and 1-4 interactions are calculated. An extra point belonging to an atom has a 1-1 interaction with it, and participates in any 1-2, 1-3 or 1-4 interaction that atom has. For example, suppose (excusing the geometry) C1,C2,C3,C4 form a dihedral and each has 1 extra point attached as below

```
C1------C2------C3------C4
 |       |       |       |
Ep1     Ep2     Ep3     Ep4
```

The 1-4 interactions include C1-C4, Ep1-C4, C1-Ep4, and Ep1-Ep4. (To see a printout of all 1-1, 1-2, 1-3 and 1-4 interactions set verbose=1.) These interactions are masked out of nonbonds. Thus the amber mask list is rebuilt from these 1-1, 1-2, 1-3 and 1-4 pairs. A separate list of 1-4 nonbonds is then compiled. This list does not agree in general with the above 1-4, since a 1-4 could also be a 1-3 if its in a ring. See the *ephi()* routine for the precise algorithm involved here. The list of 1-4 nonbonds is printed if *verbose=1.*

2.8.4 Detailed MPI Timings

`profile_mpi` Adjusts whether detailed per thread timings should be written to a file called profile_mpi when running sander in parallel. By default only average timings are printed to the output file. This is done for performance reasons, especially when running *multisander* runs. However for development it is useful to know the individual timings for each mpi thread. When running in serial the value of profile_mpi is ignored.

= 0 No detailed MPI timings will be written (default).

= 1 A detailed breakdown of the timings for each MPI thread will be written to the file: profile_mpi.

2.9 Varying conditions

This section of information is read (*if NMROPT* $>$ *0*) as a series of namelist specifications, with name "&wt". This namelist is read repeatedly until a namelist &wt statement is found with TYPE=END.

`TYPE` Defines quantity being varied; valid options are listed below.

`ISTEP1,ISTEP2` This change is applied over steps/iterations ISTEP1 through ISTEP2. If ISTEP2 = 0, this change will remain in effect from step ISTEP1 to the end of the run at a value of VALUE1 (VALUE2 is ignored in this case). *(default= both 0)*

`VALUE1,VALUE2` Values of the change corresponding to ISTEP1 and ISTEP2, respectively. If ISTEP2=0, the change is fixed at VALUE1 for the remainder of the run, once step ISTEP1 is reached.

`IINC` If IINC $>$ 0, then the change is applied as a step function, with IINC steps/iterations between each change in the target VALUE (ignored if ISTEP2=0). If IINC =0, the change is done continuously. *(default=0)*

`IMULT` If IMULT=0, then the change will be linearly interpolated from VALUE1 to VALUE2 as the step number increases from ISTEP1 to ISTEP2. *(default)* If IMULT=1, then the change will be effected by a series of multiplicative scalings, using a single factor, R, for all scalings. i.e.

VALUE2 = (R**INCREMENTS) * VALUE1.

INCREMENTS is the number of times the target value changes, which is determined by ISTEP1, ISTEP2, and IINC.

The remainder of this section describes the options for the TYPE parameter. For a few types of cards, the meanings of the other variables differ from that described above; such differences are noted below. Valid Options for TYPE (you must use uppercase) are:

`BOND`	Varies the relative weighting of bond energy terms.
`ANGLE`	Varies the relative weighting of valence angle energy terms.
`TORSION`	Varies the relative weighting of torsion (and J-coupling) energy terms. Note that any restraints defined in the input to the PARM program are included in the above. Improper torsions are handled separately (IMPROP).
`IMPROP`	Varies the relative weighting of the "improper" torsional terms. These are not included in TORSION.
`VDW`	Varies the relative weighting of van der Waals energy terms. This is equivalent to changing the well depth (epsilon) by the given factor.
`HB`	Varies the relative weighting of hydrogen-bonding energy terms.
`ELEC`	Varies the relative weighting of electrostatic energy terms.
`NB`	Varies the relative weights of the non-bonded (VDW, HB, and ELEC) terms.
`ATTRACT`	Varies the relative weights of the attractive parts of the van der waals and h-bond terms.
`REPULSE`	Varies the relative weights of the repulsive parts of the van der waals and h-bond terms.
`RSTAR`	Varies the effective van der Waals radii for the van der Waals (VDW) interactions by the given factor. Note that this is done by changing the relative attractive and repulsive coefficients, so ATTRACT/REPULSE should not be used over the same step range as RSTAR.
`INTERN`	Varies the relative weights of the BOND, ANGLE and TORSION terms. "Improper" torsions (IMPROP) must be varied separately.
`ALL`	Varies the relative weights of all the energy terms above (BOND, ANGLE, TORSION, VDW, HB, and ELEC; does not affect RSTAR or IMPROP).
`REST`	Varies the relative weights of *all* the NMR restraint energy terms.
`RESTS`	Varies the weights of the "short-range" NMR restraints. Short- range restraints are defined by the SHORT instruction (see below).
`RESTL`	Varies the weights of any NMR restraints which are not defined as "short range" by the SHORT instruction (see below). When no SHORT instruction is given, RESTL is equivalent to REST.
`NOESY`	Varies the overall weight for NOESY volume restraints. Note that this value multiplies the individual weights read into the "awt" array. (Only if NMROPT=2; see Section 4 below).
`SHIFTS`	Varies the overall weight for chemical shift restraints. Note that this value multiplies the individual weights read into the "wt" array. (Only if NMROPT=2; see section 4 below).

`XRAY` Varies the xray_weight restraint weight.

`SHORT` Defines the short-range restraints. For this instruction, ISTEP1, ISTEP2, VALUE1, and VALUE2 have different meanings. A short-range restraint can be defined in two ways.

(1) If the residues containing each pair of bonded atoms comprising the restraint are close enough in the primary sequence:

$$\text{ISTEP1} \leq \text{ABS(delta\_residue)} \leq \text{ISTEP2},$$

where delta_residue is the difference in the numbers of the residues containing the pair of bonded atoms.

(2) If the distances between each pair of bonded atoms in the restraint fall within a prescribed range:

$$\text{VALUE1} \leq \text{distance} \leq \text{VALUE2}.$$

Only one SHORT command can be issued, and the values of ISTEP1, ISTEP2, VALUE1, and VALUE2 remain fixed throughout the run. However, if IINC>0, then the short-range interaction list will be re-evaluated every IINC steps.

`TGTRMSD` Varies the RMSD target value for targeted MD.

`TEMP0` Varies the target temperature TEMP0.

`TEMP0LES` Varies the LES target temperature TEMP0LES.

`TAUTP` Varies the coupling parameter, TAUTP, used in temperature scaling when temperature coupling options NTT=1 is used.

`CUT` Varies the non-bonded cutoff distance.

`NSTEP0` If present, this instruction will reset the initial value of the step counter (against which ISTEP1/ISTEP2 and NSTEP1/NSTEP2 are compared) to the value ISTEP1. This only affects the way in which NMR weight restraints are calculated. It does not affect the value of NSTEP that is printed as part of the dynamics output. An NSTEP0 instruction only has an effect at the beginning of a run. For this card (only) ISTEP2, VALUE1, VALUE2 and IINC are ignored. If this card is omitted, NSTEP0 = 0. This card can be useful for simulation restarts, where NSTEP0 is set to the final step on the previous run.

`STPMLT` If present, the NMR step counter will be changed in increments of STPMLT for each actual dynamics step. For this card, only VALUE1 is read. ISTEP1, ISTEP2, VALUE2, IINC, and IMULT are ignored. Default = 1.0.

`DISAVE, ANGAVE, TORAVE` If present, then by default time-averaged values (rather than instantaneous values) for the appropriate set of restraints will be used. DISAVE controls distance data, ANGAVE controls angle data, TORAVE controls torsion data. See below for the functional form used in generating time-averaged data.

For these cards: VALUE1 = τ (characteristic time for exponential decay) VALUE2 = POWER (power used in averaging; the nearest integer of value2 is used) Note that the range (ISTEP1→ISTEP2) applies only to TAU; The value of POWER is not changed by subsequent cards with the same ITYPE field, and time-averaging will always be turned on for the entire run if one of these cards appears.

Note also that, due to the way that the time averaged internals are calculated, changing τ at any time after the start of the run will only affect the relative weighting of steps occurring after the change in τ . Separate values for τ and POWER are used for bond, angle, and torsion averaging.

The default value of τ (if it is 0.0 here) is 1.0D+6, which results in no exponential decay weighting. Any value of $\tau \geq$ 1.D+6 will result in no exponential decay.

If DISAVE,ANGAVE, or TORAVE is chosen, one can still force use of an instantaneous value for specific restraints of the particular type (bond, angle, or torsion) by setting the IFNTYP field to "1" when the restraint is defined (IFNTYP is defined in the DISANG file).

If time-averaging for a particular class of restraints is being performed, all restraints of that class that are being averaged (that is, all restraints of that class except those for which IFNTYP=1) *must* have the same values of NSTEP1 and NSTEP2 (NSTEP1 and NSTEP2 are defined below). (For these cards, IINC and IMULT are ignored) See the discussion of time-averaged restraints following the input descriptions.

`DISAVI, ANGAVI, TORAVI` **ISTEP1:** Ignored.

ISTEP2: Sets IDMPAV. If IDMPAV $>$ 0, *and* a dump file has been specified (DUMPAVE is set in the file redirection section below), then the time-averaged values of the restraints will be written every IDMPAV steps. Only one value of IDMPAV can be set (corresponding to the first DISAVI/ANGAVI/TORAVI card with ISTEP2 $>$ 0), and *all* restraints (even those with IFNTYP=1) will be "dumped" to this file every IDMPAV steps. The values reported reflect the current value of τ.

VALUE1: The integral which gives the time-averaged values is undefined for the first step. By default, for each time-averaged internal, the integral is assigned the current value of the internal on the first step. If VALUE1$\neq$0, this initial value of internal r is reset as follows:

```
-1000. < VALUE1 < 1000.: Initial value = r_initial + VALUE
 VALUE1 <= -1000.: Initial value = r_target + 1000.
 1000. <= VALUE1 : Initial value = r_target - 1000.
```

r_target is the target value of the internal, given by R2+R3 (or just R3, if R2 is 0). VALUE1 is in angstroms for bonds, in degrees for angles.

VALUE2: This field can be used to set the value of τ used in calculating the time-averaged values of the internal restraints reported at the end of a simulation (if LISTOUT is specified in the redirection section below). By default, no exponential decay weighting is used in calculating the final reported values, regardless of what value of τ was used during the simulation. If VALUE2>0, then τ = VALUE2 will be used in calculating these final reported averages. Note that the value of VALUE2 = τ specified here only affects the reported averaged values in at the end of a simulation. It does not affect the time-averaged values used during the simulation (those are changed by the VALUE1 field of DISAVE, ANGAVE and TORAVE instructions).

IINC: If IINC = 0, then forces for the class of time-averaged restraints will be calculated exactly as (dE/dr_ave) (dr_ave/dx). If IINC = 1, then then forces for the class of time-averaged restraints will be calculated as (dE/dr_ave) (dr(t)/dx). Note that this latter method results in a non-conservative force, and does not integrate to a standard form. But this latter formulation helps avoid the large forces due to the (1+*i*) term in the exact derivative calculation–and may avert instabilities in the molecular dynamics trajectory for some systems. See the discussion of time-averaged restraints following the input description. Note that the DISAVI, ANGAVI, and TORAVI instructions will have no affect unless the corresponding time average request card (DISAVE, ANGAVE or TORAVE, respectively) is also present.

`DUMPFREQ`	Istep1 is the only parameter read, and it sets the frequency at which the coordinates in the distance or angle restraints are dumped to the file specified by the DUMPAVE command in the I/O redirection section. (For these cards, ISTEP1 and IMULT are ignored).
`END`	END of this section.

NOTES:

1. All weights are relative to a default of 1.0 in the standard force field.
2. Weights are not cumulative.
3. For any range where the weight of a term is not modified by the above, the weight reverts to 1.0. For any range where TEMP0, SOFTR or CUTOFF is not specified, the value of the relevant constant is set to that specified in the input file.
4. If a weight is set to 0.0, it is set internally to 1.0D-7. This can be overridden by setting the weight to a negative number. In this case, a weight of exactly 0.0 will be used. *However,* if any weight is set to exactly 0.0, it cannot be changed again during this run of the program.
5. If two (or more) cards change a particular weight over the same range, the weight given on the last applicable card will be the one used.
6. Once any weight change for which NSTEP2=0 becomes active (i.e. one which will be effective for the remainder of the run), the weight of this term cannot be further modified by other instructions.
7. Changes to RSTAR result in exponential weighting changes to the attractive and repulsive terms (proportional to the scale factor**6 and **12, respectively). For this reason, scaling RSTAR to a very small value (e.g. $\leq$0.1) may result in a zeroing-out of the vdw term.

2.10 File redirection commands

Input/output redirection information can be read as described here. Redirection cards must follow the end of the weight change information. Redirection card input is terminated by the first non-blank line which does not start with a recognized redirection TYPE (e.g. LISTIN, LISTOUT, etc.).

The format of the redirection cards is

TYPE = filename

where TYPE is any valid redirection keyword (see below), and filename is any character string. The equals sign ("=") is required, and TYPE must be given in *uppercase* letters.

Valid redirection keywords are:

`LISTIN`	An output listing of the restraints which have been read, and their deviations from the target distances *before* the simulation has been run. By default, this listing is not printed. cedureIf POUT is used for the filename, these deviations will be printed in the normal output file.
`LISTOUT`	An output listing of the restraints which have been read, and their deviations from the target distances _*after* the simulation has finished. By default, this listing is not printed. If POUT is used for the filename, these deviations will be printed in the normal output file.
`DISANG`	The file from which the distance and angle restraint information described below (Section 9.1) will be read.
`NOESY`	File from which NOESY volume information (Section 9.2) will be read.
`SHIFTS`	File from which chemical shift information (Section 9.3) will be read.

`PCSHIFT`	File from which paramagnetic shift information (Section 9.3) will be read.
`DIPOLE`	File from which residual dipolar couplings (Section 9.5) will be read.
`CSA`	File from which CSA or pseduo-CSA restraints (Section 9.6) will be read.
`DUMPAVE`	File to which the time-averaged values of all restraints will be written. If DISAVI / ANGAVI / TORAVI has been used to set IDMPAV$\neq$0, then averaged values will be output. If the DUMPFREQ command has been used, the instantaneous values will be output.

2.11 Getting debugging information

The debug options in *sander* are there principally to help developers test new options or to test results between two machines or versions of code, but can also be useful to users who want to test the effect of parameters on the accuracy of their ewald or pme calculations. If the debug options are set, *sander* will exit after performing the debug tasks set by the user.

To access the debug options, include a &debugf namelist. Input parameters are:

`do_debugf`	Flag to perform this module. Possible values are zero or one. Default is zero. Set to one to turn on debug options.

One set of options is to test that the atomic forces agree with numerical differentiation of energy.

`atomn`	Array of atom numbers to test atomic forces on. Up to 25 atom numbers can be specified, separated by commas.
`nranatm`	number of random atoms to test atomic forces on. Atom numbers are generated via a random number generator.
`ranseed`	seed of random number generator used in generating atom numbers default is 71277
`neglgdel`	negative log of delta used in numerical differentiating; e.g. 4 means delta is 10^{-4} Angstroms. Default is 5. *Note*: In general it does no good to set nelgdel larger than about 6. This is because the relative force error is at best the square root of the numerical error in the energy, which ranges from 10^{-15} up to 10^{-12} for energies involving a large number of terms.
`chkvir`	Flag to test the atomic and molecular virials numerically. Default is zero. Set to one to test virials.
`dumpfrc`	Flag to dump energies, forces and virials, as well as components of forces (bond, angle forces etc.) to the file "forcedump.dat" This produces an ascii file. Default is zero. Set to one to dump forces.
`rmsfrc`	Flag to compare energies forces and virials as well as components of forces (bond, angle forces etc.) to those in the file "forcedump.dat". Default is zero. Set to one to compare forces.

Several other options are also possible to modify the calculated forces.

`zerochg`	Flag to zero all charges before calculating forces. Default zero. Set to one to remove charges.
`zerovdw`	Flag to remove all van der Waals interactions before calculating forces. Default zero. Set to one to remove van der Waals.
`zerodip`	Flag to remove all atomic dipoles before calculating forces. Only relevant when polarizability is invoked.

`do_dir, do_rec, do_adj, do_self, do_bond, do_cbond, do_angle, do_ephi, do_xconst, do_cap`
: These are flags which turn on or off the subroutines they refer to. The defaults are one. Set to zero to prevent a subroutine from running. For example, set do_dir=0 to turn off the direct sum interactions (van der Waals as well as electrostatic). These options, as well as the zerochg, zerovdw, zerodip flags, can be used to fine tune a test of forces, accuracy, etc.

EXAMPLES:

This input list tests the reciprocal sum forces on atom 14 numerically, using a delta of 10^{-4}.

```
&debugf
neglgdel=4, nranatm = 0, atomn = 14,
do_debugf = 1,do_dir = 0,do_adj = 0,do_rec = 1, do_self = 0,
do_bond = 1,do_angle = 0,do_ephi = 0, zerovdw = 0, zerochg = 0,
chkvir = 0,
dumpfrc = 0,
rmsfrc = 0,
/
```

This input list causes a dump of force components to "forcedump.dat". The bond, angle and dihedral forces are not calculated, and van der Waals interactions are removed, so the total force is the Ewald electrostatic force, and the only nonzero force components calculated are electrostatic.

```
&debugf
neglgdel=4, nranatm = 0, atomn = 0,
do_debugf = 1,do_dir = 1,do_adj = 1,do_rec = 1, do_self = 1,
do_bond = 0,do_angle = 0,do_ephi = 0, zerovdw = 1, zerochg = 0,
chkvir = 0,
dumpfrc = 1,
rmsfrc = 0,
/
```

In this case the same force components as above are calculated, and compared to those in "forcedump.dat". Typically this is used to get an RMS force error for the Ewald method in use. To do this, when doing the force dump use ewald or pme parameters to get high accuracy, and then normal parameters for the force compare:

```
&debugf
neglgdel=4, nranatm = 0, atomn = 0,
do_debugf = 1,do_dir = 1,do_adj = 1,do_rec = 1, do_self = 1,
do_bond = 0,do_angle = 0,do_ephi = 0, zerovdw = 1, zerochg = 0,
chkvir = 0,
dumpfrc = 0,
rmsfrc = 1,
/
```

For example, if you have a 40x40x40 unit cell and want to see the error for default pme options (cubic spline, 40x40x40 grid), run 2 jobs—— (assume box params on last line of inpcrd file)

Sample input for 1st job:

```
&cntrl
dielc =1.0,
cut = 11.0, nsnb = 5, ibelly = 0,
ntx = 5, irest = 1,
ntf = 2, ntc = 2, tol = 0.0000005,
ntb = 1, ntp = 0, temp0 = 300.0, tautp = 1.0,
nstlim = 1, dt = 0.002, maxcyc = 5, imin = 0, ntmin = 2,
ntpr = 1, ntwx = 0, ntt = 0, ntr = 0,
jfastw = 0, nmrmax=0, ntave = 25,
/
&debugf
do_debugf = 1,do_dir = 1,do_adj = 1,do_rec = 1, do_self = 1,
do_bond = 0,do_angle = 0,do_ephi = 0, zerovdw = 1, zerochg = 0,
chkvir = 0,
dumpfrc = 1,
rmsfrc = 0,
/
&ewald
nfft1=60,nfft2=60,nfft3=60,order=6, ew_coeff=0.35,
/
```

Sample input for 2nd job:

```
&cntrl
dielc =1.0,
cut = 8.0, nsnb = 5, ibelly = 0,
ntx = 5, irest = 1,
ntf = 2, ntc = 2, tol = 0.0000005,
ntb = 1, ntp = 0, temp0 = 300.0, tautp = 1.0,
nstlim = 1, dt = 0.002, maxcyc = 5, imin = 0, ntmin = 2,
ntpr = 1, ntwx = 0, ntt = 0, ntr = 0,
jfastw = 0, nmrmax=0, ntave = 25,
/
&debugf
do_debugf = 1,do_dir = 1,do_adj = 1,do_rec = 1, do_self = 1,
do_bond = 0,do_angle = 0,do_ephi = 0, zerovdw = 1, zerochg = 0,
chkvir = 0,
dumpfrc = 0,
rmsfrc = 1,
/
&ewald
ew_coeff=0.35,
/
```

Note that an Ewald coefficient of 0.35 is close to the default error for an 8 Angstrom cutoff. However, the first job used an 11 Angstrom cutoff. The direct sum forces calculated in the 2nd job are compared to these, giving the RMS error due to an 8 Angstrom cutoff, with this value of ew_coeff. The reciprocal sum error calculated in the 2nd job is with respect to the pme reciprocal forces in the 1st job considered as "exact".

Note further that if in these two jobs you had not specified "ew_coeff" *sander* would have calculated ew_coeff according to the cutoff and the direct sum tolerance, defaulted to 10^{-5}. This would give two different ewald coefficients. Under these circumstances the direct, reciprocal and adjust energies and forces would not agree well between the two jobs. However the total energy and forces should agree reaso`nrespa`nably, (forces to within about 5x10^{-4} relative RMS force error) Since the totals are invariant to the coefficient.

Finally, note that if other force components are calculated, such as van der Waals, bond, angle, etc., then the total force will include these, and the relative RMS force errors will be with respect to this total force in the denominator.

2.12 multisander

The *multisander* functionality is available in the parallel versions of the programs (i.e., *sander.MPI*). This mode allows multiple independent simulations, or replicas, to be run in the same program instance. It is particularly useful for computer clusters in which priority is given to large CPU-count jobs. In this case, the command-line usage of sander and pmemd is slightly altered, as shown below:

```
mpirun -np <#proc> sander.MPI -ng <#groups> -groupfile groupfile
```

In this case, `#proc` processors will be evenly divided among `#groups` individual simulations (`#proc` must be a multiple of `#group`!). The groupfile consists of a number of lines which is the command-line for each of the `#groups` simulations you wish to run. Comment lines (i.e., those with # in the first column) are ignored, after which the first `#groups` lines are read as the command-line flags of the N^{th} simulation.

The multisander and multipmemd mechanisms are also utilized for methods requiring multiple simulations to communicate with one another, such as thermodynamic integration in sander and replica exchange molecular dynamics (both described later). An example groupfile and program call are shown below.

Groupfile:

```
# Comment lines must start with a pound sign
# and there can be as many comment lines as you
# want, wherever you want them.
-O -p prmtop1 -c inpcrd1 -i replica1.mdin -suffix replica1
-O -p prmtop2 -c inpcrd2 -i replica2.mdin -suffix replica2
-O -p prmtop3 -c inpcrd3 -i replica3.mdin -suffix replica3
-O -p prmtop4 -c inpcrd4 -i replica4.mdin -suffix replica4
```

The `-suffix` flag behaves slightly differently than it does for classical use. In standard simulations (*i.e.*, without *multisander* or *multipmemd*), the provided suffix will be applied only to output files that are printed but were not given names on the command-line. With multisander, however, each thread has to produce different output files so that different replicas do not try to write to the same file. As a result, a default suffix of 000, 001, 002, etc. is given to the replicas and is added to every unspecified output file. If a `-suffix` is specified in the groupfile, as shown above, every output file—including those given an explicit name for that replica—are given the additional suffix.

The four simulations shown in the groupfile above can be run on 8 processors each with the following command (note, running *sander.MPI* may differ on your system).

```
mpirun -np 32 sander.MPI -ng 4 -groupfile groupfile
```

The *multisander* and *multipmemd* concepts are implemented via the use of MPI communicators. Each replica is assigned a replica-wide communicator along which all communications required for standard MD simulations are performed (called `commsander` and `pmemd_comm` in *sander* and *pmemd*, respectively). Each replica communicator has a master thread (rank 0 in that communicator), and the master thread of each replica are joined in another MPI communicator of replica masters (called `commmaster` and `pmemd_master_comm` in *sander* and *pmemd*, respectively). All inter-replica communication is performed via `commmaster` or `pmemd_master_comm`.

By default, all N threads are allocated to each of the M groups by dividing the threads sequentially. That is, the first N/M threads are assigned to replica 0, the second group of N/M threads are assigned to replica 1, etc. The `-ng-nonsequential` flag will stripe the thread assignments. Replica 0 will receive threads 0, $N-1$, $2N-1$, etc., while replica 1 receives threads 1, N, $2N$, etc.

3 The Generalized Born/Surface Area Model

Implicit solvent methods can speed up atomistic simulations by approximating the discrete solvent as a continuum, thus drastically reducing the number of particles in the system. An additional effective speedup often comes from much faster sampling of the conformational space afforded by these methods.[15, 28–31] The generalized Born (GB) solvation model is the most commonly used implicit solvent model for atomistic MD simulation; it has been most widely tested on ff99SB and ff14SBonlysc, but in principle could be used with other non-polarizable force fields, such as ff03. A recent (2019) review gives a good overview.[32] To estimate the total solvation free energy of a molecule, ΔG_{solv}, one typically assumes that it can be decomposed into the "electrostatic" and "non-electrostatic" parts:

$$\Delta G_{solv} = \Delta G_{el} + \Delta G_{nonel} \tag{3.1}$$

where ΔG_{nonel} is the free energy of solvating a molecule from which all charges have been removed (i.e. partial charges of every atom are set to zero), and ΔG_{el} is the free energy of first removing all charges in the vacuum, and then adding them back in the presence of a continuum solvent environment. Generally speaking, ΔG_{nonel} comes from the combined effect of two types of interaction: the favorable van der Waals attraction between the solute and solvent molecules, and the unfavorable cost of breaking the structure of the solvent (water) around the solute. In the current Amber codes, this is taken to be proportional to the total solvent accessible surface area (SA) of the molecule, with a proportionality constant derived from experimental solvation energies of small non-polar molecules, and uses a fast LCPO algorithm [33] to compute an analytical approximation to the solvent accessible area of the molecule.

The Poisson-Boltzmann approach described in the next section has traditionally been used in calculating ΔG_{el}. However, in molecular dynamics applications, the associated computational costs are often very high, as the Poisson-Boltzmann equation needs to be solved every time the conformation of the molecule changes. Amber developers have pursued an alternative approach, the analytic generalized Born (GB) method, to obtain a reasonable, computationally efficient estimate to be used in molecular dynamics simulations. The methodology has become popular,[34–41] especially in molecular dynamics applications,[42–45] due to its relative simplicity and computational efficiency, compared to the more standard numerical solution of the Poisson-Boltzmann equation. Within Amber GB models, each atom in a molecule is represented as a sphere of radius R_i with a charge q_i at its center; the interior of the atom is assumed to be filled uniformly with a material of dielectric constant 1. The molecule is surrounded by a solvent of a high dielectric ε (80 for water at 300 K). The GB model approximates ΔG_{el} by an analytical formula,[34, 46]

$$\Delta G_{el} \approx -\frac{1}{2}\sum_{ij}\frac{q_i q_j}{f_{GB}(r_{ij}, R_i, R_j)}\left(1 - \frac{\exp[-\kappa f_{GB}]}{\varepsilon}\right) \tag{3.2}$$

where r_{ij} is the distance between atoms i and j, the R_i are the so-called *effective Born radii*, and $f_{GB}()$ is a certain smooth function of its arguments. The electrostatic screening effects of (monovalent) salt are incorporated [46] via the Debye-Huckel screening parameter κ.

A common choice [34] of f_{GB} is

$$f_{GB} = \left[r_{ij}^2 + R_i R_j \exp(-r_{ij}^2/4R_i R_j)\right]^{1/2} \tag{3.3}$$

although other expressions have been tried.[37, 47] The effective Born radius of an atom reflects the degree of its burial inside the molecule: for an isolated ion, it is equal to its van der Waals (VDW) radius ρ_i. Then one obtains the particularly simple form:

$$\Delta G_{el} = -\frac{q_i^2}{2\rho_i}\left(1 - \frac{1}{\epsilon}\right) \tag{3.4}$$

where we assumed $\kappa = 0$ (pure water). This is the famous expression due to Born for the solvation energy of a single ion. The function $f_{GB}()$ is designed to interpolate, in a clever manner, between the limit $r_{ij} \to 0$, when atomic spheres merge into one, and the opposite extreme $r_{ij} \to \infty$, when the ions can be treated as point charges obeying the Coulomb's law.[40] For deeply buried atoms, the effective radii are large, $R_i \gg \rho_i$, and for such atoms one can use a rough estimate $R_i \approx L_i$, where L_i is the distance from the atom to the molecular surface. Closer to the surface, the effective radii become smaller, and for a completely solvent exposed side-chain one can expect R_i to approach ρ_i.

The effective radii depend on the molecule's conformation, and so have to be re-computed every time the conformation changes. This makes the computational efficiency a critical issue, and various approximations are normally made that facilitate an effective estimate of R_i. With the exception of GBNSR6 (see Section **??**), the so-called *Coulomb field approximation*, or *CFA*, is used for Amber GB models, which replaces the true electric displacement around the atom by the Coulomb field. Within this assumption, the following expression can be derived:[40]

$$R_i^{-1} = \rho_i^{-1} - \frac{1}{4\pi}\int \theta(|\mathbf{r}| - \rho_i) r^{-4} d^3\mathbf{r} \tag{3.5}$$

where the integral is over the solute volume surrounding atom i. For a realistic molecule, the solute boundary (molecular surface) is anything but trivial, and so further approximations are made to obtain a closed-form analytical expression for the above equation, *e.g.* the so-called pairwise de-screening approach of Hawkins, Cramer and Truhlar,[48] which leads to a GB model implemented in Amber with *igb=1*. The 3D integral used in the estimation of the effective radii is performed over the van der Waals (VDW) spheres of solute atoms, which implies a definition of the solute volume in terms of a set of spheres, rather than the complex molecular surface,[49] commonly used in the PB calculations. For macromolecules, this approach tends to underestimate the effective radii for buried atoms,[40] arguably because the standard integration procedure treats the small vacuum–filled crevices between the van der Waals (VDW) spheres of protein atoms as being filled with water, even for structures with large interior.[47] This error is expected to be greatest for deeply buried atoms characterized by large effective radii, while for the surface atoms it is largely canceled by the opposing error arising from the Coulomb approximation, which tends [35, 39, 50] to overestimate R_i.

The deficiency of the model described above can, to some extent, be corrected by noticing that even the optimal packing of hard spheres, which is a reasonable assumption for biomolecules, still occupies only about three quarters of the space, and so "scaling-up" of the integral by a factor of four thirds should effectively increase the underestimated radii by about the right amount, without any loss of computational efficiency. This idea was developed and applied in the context of pH titration,[40] where it was shown to improve the performance of the GB approximation in calculating pKa values of protein sidechains. However, the one-parameter correction introduced in Ref. [40] was not optimal in keeping the model's established performance on small molecules. It was therefore proposed [45] to re-scale the effective radii with the re-scaling parameters being proportional to the degree of the atom's burial, as quantified by the value I_i of the 3D integral. The latter is large for the deeply buried atoms and small for exposed ones. Consequently, one seeks a well-behaved re-scaling function, such that $R_i \approx (\rho_i^{-1} - I_i)^{-1}$ for small I_i, and $R_i > (\rho_i^{-1} - I_i)^{-1}$ when I_i becomes large. The following simple, infinitely differentiable re-scaling function was chosen to replace the model's original expression for the effective radii:

$$R_i^{-1} = \tilde{\rho}_i^{-1} - \rho_i^{-1}\tanh(\alpha\Psi - \beta\Psi^2 + \gamma\Psi^3) \tag{3.6}$$

where $\Psi = I_i\tilde{\rho}_i$, and α, β, γ are treated as adjustable dimensionless parameters which were optimized using the guidelines mentioned earlier (primarily agreement with the PB). Currently, Amber supports two GB models (termed OBC) based on this idea. These differ by the values of α, β, γ, and are invoked by setting igb to either igb=2 or igb=5. The details of the optimization procedure and the performance of the OBC model relative to the PB treatment and in MD simulations on proteins is described in Ref. [45]; an independent comparison to the PB in calculating the electrostatic part of solvation free energy on a large data set of proteins can be found in Ref. [51].

Our experience with generalized Born simulations is mainly with *ff99SB*, *ff14SBonlysc* 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 [52, 53] 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 α-helix present.[54, 55] The combination of the *ff14SBonlysc* force field with igb=8 gives the best results for proteins [56][57], nucleic acids and protein-nucleic acid complexes. [58]

Despite these limitations, implicit treatment of solvent is widely used in molecular simulations for two main reasons: algorithmic/computational speed and conformational sampling. [15, 59] Implicit solvent methods can be algorithmically/computationally faster, as measured by simulation time steps per processor (CPU) time, because the vast number of individual interactions between the atoms of individual solvent molecules do not need to be explicitly computed. Implicit-solvent simulations can also sample conformational space faster in the low viscosity regime afforded by the implicit solvent model.[15, 28–31] To some extent, the interest in implicit-solvent-based simulations is motivated by the need to sample very large conformational spaces for problems such as protein folding, binding-affinity calculations, or large-scale fluctuations of nucleosomal DNA fragments. The speedup of conformational change can vary considerably, depending on the details of the transition, and can range from no speedup at all to almost a 100-fold speedup. [15] In general, the larger the conformational change, the higher the speedup one may expect, but this tendency is not universal or uniform. These speedup values are also expected to vary by the specific flavour of GB model used, a detailed analysis for igb5 can be found in Ref. [15].

The generalized Born models used here are based on the "pairwise" model introduced by Hawkins, Cramer and Truhlar,[48, 60] which in turn is based on earlier ideas by Still and others.[34, 39, 50, 61] The so-called overlap parameters for most models are taken from the Tinker molecular modeling package (http://tinker.wustl.edu). The effects of added monovalent salt are included at a level that approximates the solutions of the linearized Poisson-Boltzmann equation.[46] The original implementation was by David Case, who thanks Charlie Brooks for inspiration. Details of our implementation of generalized Born models can be found in Refs. [62, 63].

3.1 GB/SA input parameters

As outlined above, there are several "flavors" of GB available, depending upon the value of *igb*. The version that has been most extensively tested corresponds to *igb*=1; the "OBC" models (*igb*=2 and 5) are newer, but appear to give significant improvements and are recommended for most projects (certainly for peptides or proteins). The newest, most advanced, and least extensively tested model, *GBn* (*igb*=7), yields results in considerably better agreement with molecular surface Poisson-Boltzmann and explicit solvent results than the "OBC" models under many circumstances.[55] The *GBn* model was parameterized for peptide and protein systems and is not recommended for use with nucleic acids. A modification on the GBn model (igb=8) further improves agreement between Poisson-Boltzmann and explicit solvent data compared to the original formulation (igb=7).[56] Users should understand that all (current) GB models have limitations and should proceed with caution. Generalized Born simulations can only be run for non-periodic systems, *i.e.* where *ntb*=0. Unlike its use in explicit solvent PME simulations, short nonbonded cutoff values have much stronger impact on accuracy of the GB calculations. Essentially, any cutoff values other than *cut* > *structure size* can lead to artifacts. Current GPU implementation of the GB can not use cutoffs. An alternative that retains most of the speed of the GB with a cutoff, but without most of its artifacts, is GB-HCP described in Section 14.6. If the nonbonded cutoff is used in GB calculations, it should be greater than that for PME calculations, perhaps cut=16. The slowly-varying

1	2	5	7	8
mbondi	*mbondi2*	*mbondi2*	*bondi*	*mbondi3*

Table 3.1: *Recommended radii sets for various GB models. For values of igb given in the top row, the string in the second row should be entered in LEaP as "set default PBRadii xxx".*

forces generally do not have to be evaluated at every step for GB, either nrespa=2 or 4, although that option may lead to some artifacts as well.

igb

= 0 No generalized Born term is used. (Default)

= 1 The Hawkins, Cramer, Truhlar[48, 60] pairwise generalized Born model is used, with parameters described by Tsui and Case.[62] This model uses the default radii set up by LEaP. It is slightly different from the GB model that was included in Amber6. If you want to compare to Amber 6, or need to continue an ongoing simulation, you should use the command "set default PBradii amber6" in LEaP, and set *igb*=1 in *sander*. For reference, the Amber6 values are those used by an earlier Tsui and Case paper.[43] Note that most nucleic acid simulations have used this model, so you take care when using other values. Also note that Tsui and Case used an offset (see below) of 0.13 Å, which is different from its default value.

= 2 Use a modified GB model developed by A. Onufriev, D. Bashford and D.A. Case; the main idea was published earlier,[40] but the actual implementation here[45] is an elaboration of this initial idea. Within this model, the effective Born radii are re-scaled to account for the interstitial spaces between atom spheres missed by the GB^{HCT} approximation. In that sense, GB^{OBC} is intended to be a closer approximation to true molecular volume, albeit in an average sense. With *igb*=2, the inverse of the effective Born radius is given

by:cedure

$$R_i^{-1} = \bar{\rho}_i^{-1} - \tanh\left(\alpha\Psi - \beta\Psi^2 + \gamma\Psi^3\right)/\rho_i$$

where $\bar{\rho}_i = \rho_i - offset$, and $\Psi = I\rho_i$, with I given in our earlier paper. The parameters α, β, and γ were determined by empirical fits, and have the values 0.8, 0.0, and 2.909125. This corresponds to model I in Ref [45]. With this option, you should use the LEaP command "set default PBradii mbondi2" to prepare the *prmtop* file.

= 3 or 4 These values are unused; they were used in Amber 7 for parameter sets that are no longer supported.

= 5 Same as *igb*=2, except that now α, β, γ are 1.0, 0.8, and 4.85. This corresponds to model II in Ref [45]. With this option, you should use the command "set default PBradii mbondi2" in setting up the *prmtop* file, although "set default PBradii bondi" is also OK. When tested in MD simulations of several proteins,[45] both of the above parameterizations of the "OBC" model showed equal performance, although further tests [51] on an extensive set of protein structures revealed that the *igb=5* variant agrees better with the Poisson-Boltzmann treatment in calculating the electrostatic part of the solvation free energy.

= 6 With this option, there is no continuum solvent model used at all; this corresponds to a non-periodic, "vacuum", model where the non-bonded interactions are just Lennard-Jones and Coulomb interactions.

= 7 The *GBn* model described by Mongan, Simmerling, McCammon, Case and Onufriev[64] is employed. This model uses a pairwise correction term to GB^{HCT} to approximate a molecular surface dielectric boundary; that is to eliminate interstitial regions of high dielectric smaller than a solvent molecule. This correction affects all atoms and is geometry-specific, going beyond the geometry-free, "average" re-scaling approach of GB^{OBC}, which mostly affects buried atoms. With this method, you should use the bondi radii set. The overlap or screening parameters in the *prmtop* file are ignored, and the model-specific *GBn* optimized values are substituted. The model carries little additional computational overhead relative to the other GB models described above.[64] This method is not recommended fcedureor systems involving nucleic acids.

= 8 Same GB functional form as the GBn model (igb=7), but with different parameters. The offset, overlap screening parameters, and gbneckscale are changed. In addition, individual α , β , and γ parameters can be specified for each of the elements H, C, N, O, S, P. Parameters for other elements have not been optimized, and the default values used are the ones from igb=5, which were not element-dependent. Default values were optimized for H, C, N, O and S atoms in protein systems.[56] Although the parameters for P in proteins can be specified, the default values were not optimized and are the igb=5 values. Nucleic acids have separate parameters from those used for proteins, and default values were optimized for H, C, N, O and P atoms in nucleic acid systems.[58]

The following are the default parameters sander uses with *igb=8*:

```
Sh=1.425952, Sc=1.058554, Sn=0.733599,
So=1.061039, Ss=-0.703469, Sp=0.5,
offset=0.195141, gbneckscale=0.826836,
gbalphaH=0.788440, gbbetaH=0.798699, gbgammaH=0.437334,
gbalphaC=0.733756, gbbetaC=0.506378, gbgammaC=0.205844,
gbalphaN=0.503364, gbbetaN=0.316828, gbgammaN=0.192915,
gbalphaOS=0.867814, gbbetaOS=0.876635, gbgammaOS=0.387882,
gbalphaP=1.0, gbbetaP=0.8, gbgammaP=4.85
screen_hnu=1.69654, screen_cnu=1.26890,
screen_nnu=1.425974, screen_onu=0.18401, screen_pnu=1.54506,
gb_alpha_hnu=0.53705, gb_beta_hnu=0.36286, gb_gamma_hnu=0.11670,
gb_alpha_cnu=0.33167, gb_beta_cnu=0.19684, gb_gamma_cnu=0.09342,
gb_alpha_nnu=0.68631, gb_beta_nnu=0.46319, gb_gamma_nnu=0.13872,
gb_alpha_onu=0.60634, gb_beta_onu=0.46301, gb_gamma_onu=0.14226,
gb_alpha_pnu=0.41836, gb_beta_pnu=0.29005, gb_gamma_pnu=0.10642
```

Parameters for proteins and for nucleic acids were optimized separately and can be independently specified. Protein parameters: Sh, Sc, Sn, So, Ss and Sp are scaling parameters, gbalphaX, gbbetaX, gbgammaX are the α , β , γ set for element X. gbalphaOS, gbbetaOS, gbgammaOS is the α , β , γ set applied to both O and S. The phosphorus parameters (in proteins) were not optimized and are simply taken as the parameters used in the OBC-2 model (igb=5). Nucleic acid parameters (end with "nu"): screen_Xnu (X=h, c, n, o, p) are scaling parameters, gb_alpha_Xnu (X=h, c, n, o, p) are the α , β , γ set for element X.

Since parameters are assigned for each atom based on its residue name (hard-coded in "sander/egb.F90" (subroutine isnucat)), users need to update the residue table in the sander source code if nucleic acids with different names are simulated using this GB model.

The default values for offset=0.195141, gbneckscale=0.826836 are recommended for both proteins and nucleic acids.

mbondi3 radii are recommended with igb=8 and can be employed with the LEaP command "set default PBradii mbondi3". The mbondi3 radii were adjusted based on protein simulations, and optimization of these radii for nucleic acids is currently underway.

=10 Calculate the reaction field and nonbonded interactions using a numerical Poisson-Boltzmann solver. This option is described in the Chapter **??**. Note that this is *not* a generalized Born simulation, in spite of its use of *igb*; it is rather an alternative continuum solvent model.

`intdiel` Sets the interior dielectric constant of the molecule of interest. Default is 1.0. Other values have not been extensively tested.

`extdiel` Sets the exterior or solvent dielectric constant. Default is 78.5.

saltcon Sets the concentration (M) of 1-1 mobile counterions in solution, using a modified generalized Born theory based on the Debye-Hückel limiting law for ion screening of interactions.[46] Default is 0.0 M (*i.e.* no Debye-Hückel screening.) Setting *saltcon* to a nonzero value does result in some increase in computation time.

rgbmax This parameter controls the maximum distance between atom pairs that will be considered in carrying out the pairwise summation involved in calculating the effective Born radii. Atoms whose associated spheres are farther way than *rgbmax* from given atom will not contribute to that atom's effective Born radius. This is implemented in a "smooth" fashion (thanks mainly to W.A. Svrcek-Seiler), so that when part of an atom's atomic sphere lies inside *rgbmax* cutoff, that part contributes to the low-dielectric region that determines the effective Born radius. The default is 25 Å, which is usually plenty for single-domain proteins of a few hundred residues. Even smaller values (of 10-15 Å) are reasonable, changing the functional form of the generalized Born theory a little bit, in exchange for a considerable speed-up in efficiency, and without introducing the usual cut-off artifacts such as drifts in the total energy.
The *rgbmax* parameter affects only the effective Born radii (and the derivatives of these values with respect to atomic coordinates). The *cut* parameter, on the other hand, determines the maximum distance for the electrostatic, van der Waals and "off-diagonal" terms of the generalized Born interaction. The value of *rgbmax* might be either greater or smaller than that of *cut*: these two parameters are independent of each other. However, values of *cut* that are too small are more likely to lead to artifacts than are small values of *rgbmax*; therefore one typically sets *rgbmax* <= *cut*.

`rbornstat` If *rbornstat = 1*, the statistics of the effective Born radii for each atom of the molecule throughout the molecular dynamics simulation are reported in the output file. Default is 0.

`offset` The dielectric radii for generalized Born calculations are decreased by a uniform value "offset" to give the "intrinsic radii" used to obtain effective Born radii. Default is 0.09 Å.

gbsa Option to carry out GB/SA (generalized Born/surface area) simulations. For the default value of 0, surface area will not be computed and will not be included in the solvation term. If *gbsa = 1*, surface area will be computed using the LCPO model.[33] If *gbsa = 2*, surface area will be computed by recursively approximating a sphere around an atom, starting from an icosahedra. Note that no forces are generated in this case, hence, *gbsa = 2* only works for a single point energy calculation and is mainly intended for energy decomposition in the realm of MM-GBSA. If *gbsa = 3*, surface area will be computed using a fast pairwise approximation [65] suitable for GPU computing in pmemd.cuda program; the acceleration in pmemd.cuda compared with *gbsa = 2* is ~30 times faster [65]. Note that *gbsa = 3* is currently not supported in sander, MM-GBSA, QM/MM or libsff. Although *gbsa = 3* is supported in pmemd, the general usage is not recommended as the speed gain is trivial, given that the algorithm was particularly designed for fast approximation of surface area in GPU-accelerated GB simulations. Therefore, we recommend users to use gbsa=3 with pmemd.cuda.

surften Surface tension used to calculate the nonpolar contribution to the free energy of solvation (when *gbsa = 1*), as Enp = surften*SA. The default is 0.005 kcal/mol/A^2.[66] For *gbsa = 3*, suften works comparably with *gbsa = 1* given the same value. [65]

rdt This parameter is only used for GB simulations with LES (Locally Enhanced Sampling). In GB+LES simulations, non-LES atoms require multiple effective Born radii due to alternate descreening effects of different LES copies. When the multiple radii for a non-LES atom differ by less than RDT, only a single radius will be used for that atom. See Chapter **??** for more details. Default is 0.0 Å.

4 Reference Interaction Site Model

In addition to explicit and continuum implicit solvation models, Amber also has a third type of solvation model for molecular mechanics simulations, the reference interaction site model (RISM) of molecular solvation[67–80]. In AmberTools, 1D-RISM is available as rism1d. 3D-RISM is available as an option in NAB, MMPBSA.py and sander. rism3d.snglpnt is a simplified, standalone interface, ideal for calculating solvation thermodynamics on individual structures and trajectories. Details specific to using sander and sander.MPI can be found in Chapter 2.

4.1 Introduction

RISM is an inherently microscopic approach, calculating the equilibrium distribution of the solvent, from which all thermodynamic properties are then determined. Specifically, RISM is an approximate solution to the Ornstein-Zernike (OZ) equation[68, 77, 78, 81, 82]

$$h(r_{12},\Omega_1,\Omega_2) = c(r_{12},\Omega_1,\Omega_2) + \rho \int d\mathbf{r}_3\, d\Omega_3\, c(r_{13},\Omega_1,\Omega_3)\, h(r_{32},\Omega_3,\Omega_2), \tag{4.1}$$

where r_{12} is the separation between particles 1 and 2 while Ω_1 and Ω_2 are their orientations relative to the vector $\mathbf{r}_{12}$. The two functions in this relation are h, the total correlation function, and c, the direct correlation function. The total correlation function is defined as

$$h_{ab}(r_{ab},\Omega_a,\Omega_b) \equiv g_{ab}(r_{ab},\Omega_a,\Omega_b) - 1,$$

where g_{ab} is the pair-distribution function, which gives the conditional density distribution of species b about a. In cases where only radial separation is considered, for example by orientational averaging over site α of species a and site γ of species b, gives the familiar one dimensional site-site radial distribution function, $g_{\alpha\gamma}(r_{\alpha\gamma})$.

For real mixtures, it is often convenient to speak in terms of a solvent, V, of high concentration and a solute, U, of low concentration. A generic case of solvation is infinite dilution of the solute, i.e., $\rho^{\mathrm{U}} \to 0$. We can rewrite Equation (4.1), in the limit of infinite dilution, as a set of three equations:

$$h^{\mathrm{VV}}(r_{12},\Omega_1,\Omega_2) = c^{\mathrm{VV}}(r_{12},\Omega_1,\Omega_2) + \rho^{\mathrm{V}} \int d\mathbf{r}_3\, d\Omega_3\, c^{\mathrm{VV}}(r_{13},\Omega_1,\Omega_3)\, h^{\mathrm{VV}}(r_{32},\Omega_3,\Omega_2), \tag{4.2}$$

$$h^{\mathrm{UV}}(r_{12},\Omega_1,\Omega_2) = c^{\mathrm{UV}}(r_{12},\Omega_1,\Omega_2) + \rho^{\mathrm{V}} \int d\mathbf{r}_3\, d\Omega_3\, c^{\mathrm{UV}}(r_{13},\Omega_1,\Omega_3)\, h^{\mathrm{VV}}(r_{32},\Omega_3,\Omega_2), \tag{4.3}$$

$$h^{\mathrm{UU}}(r_{12},\Omega_1,\Omega_2) = c^{\mathrm{UU}}(r_{12},\Omega_1,\Omega_2) + \rho^{\mathrm{V}} \int d\mathbf{r}_3\, d\Omega_3\, c^{\mathrm{UV}}(r_{13},\Omega_1,\Omega_3)\, h^{\mathrm{VU}}(r_{32},\Omega_3,\Omega_2). \tag{4.4}$$

Equation (4.3) 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 (4.3) can be obtained using 3D-RISM. However, a solution to Equation (4.2) 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[81]

$$g(r_{12},\Omega_1,\Omega_2) = \exp\left[-\beta u(r_{12},\Omega_1,\Omega_2) + h(r_{12},\Omega_1,\Omega_2) - c(r_{12},\Omega_1,\Omega_2) + b(r_{12},\Omega_1,\Omega_2)\right] \tag{4.5}$$

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 [81]). It should be noted that u is the only point at which the interaction potential enters the equations. Depending on the method used to solve the OZ equations, u is generally an explicit potential. In principle, it should now be possible to solve our two equations. For example, we may wish to use SPC/E as a water model. Inputting the relevant aspects of the SPC/E model into u, 1D-RISM can be used to calculate the equilibrium properties of the SPC/E model. A different explicit water model will yield different properties.

A fundamental problem for all OZ-like integral equation theories is the bridge function, which contains multiple integrals that are readily solved only in special circumstances. In practice, an approximate closure relation must be used. While many closures have been developed, at this time only three are implemented in 3D-RISM: hypernetted-chain approximation (HNC), Kovalenko-Hirata (KH) and the partial series expansion of order-n (PSE-n).

For HNC, we set $b = 0$, giving[81]

$$\begin{aligned} g^{\mathrm{HNC}}(r_{12},\Omega_1,\Omega_2) &= \exp\left(-\beta u(r_{12},\Omega_1,\Omega_2) + h(r_{12},\Omega_1,\Omega_2) - c(r_{12},\Omega_1,\Omega_2)\right) \\ &= \exp\left(t^*(r_{12},\Omega_1,\Omega_2)\right) \end{aligned} \tag{4.6}$$

where t^* is the renormalize-indirect correlation function. HNC works well in many situations, including charged particles, but has difficulties when the size ratios of particles in the system are highly varied and may not always converge on a solution when one should exist. Also, as the bridge term is generally repulsive, HNC allows particles to approach too closely, overestimating non-Coulombic interactions[78].

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[77, 78]

$$g^{\mathrm{KH}}(r_{12},\Omega_1,\Omega_2) = \begin{cases} \exp\Big(t^*(r_{12},\Omega_1,\Omega_2)\Big) & \text{for } g(r_{12},\Omega_1,\Omega_2) \le 1 \\ 1 + t^*(r_{12},\Omega_1,\Omega_2) & \text{for } g(r_{12},\Omega_1,\Omega_2) > 1 \end{cases}. \tag{4.7}$$

Like HNC, KH handles Coulombic systems well but overestimates non-Coulombic interactions. Unlike HNC, it does not have difficulties with highly asymmetric particle sizes and readily converges to stable solutions for almost all systems of practical interest. The reliability of the KH closure makes it particularly suitable for molecular mechanics calculations.

PSE-n offers the ability to interpolate between KH and HNC. Here, the exponential regions of solvent density enrichment are treated as a Taylor expansion,

$$g^{\mathrm{PSE\text{-}n}}(r_{12},\Omega_1,\Omega_2) = \begin{cases} \exp\Big(t^*(r_{12},\Omega_1,\Omega_2)\Big) & \text{for } g(r_{12},\Omega_1,\Omega_2) \le 1 \\ \sum_{i=0}^{n} {(t^*(r_{12},\Omega_1,\Omega_2))^i}/{i!} & \text{for } g(r_{12},\Omega_1,\Omega_2) > 1 \end{cases}. \tag{4.8}$$

In the case of $n = 1$, the KH closure is obtained, while in the limit of $n \to \infty$ HNC is recovered. This allows a balance between the numerical stability of KH and the often better accuracy of HNC.

4.1.1 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 (4.3) becomes[76, 77]

$$h_\gamma^{\mathrm{UV}}(\mathbf{r}) = \sum_\alpha \int d\mathbf{r}' c_\alpha^{\mathrm{UV}}(\mathbf{r}-\mathbf{r}')\,\chi_{\alpha\gamma}^{\mathrm{VV}}(r'), \tag{4.9}$$

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

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

3D-RISM supports HNC, KH and PSE-n closures (see Sections **??**, 14.2 and **??**). As with the 1D-RISM closures, these are constructed by analogy from Eqs. 4.6-4.8. For example, HNC becomes

$$g_{\gamma}^{\mathrm{HNC,UV}}(\mathbf{r}) = \exp\left(-\beta u_{\gamma}^{\mathrm{UV}}(\mathbf{r}) + h_{\gamma}^{\mathrm{UV}}(\mathbf{r}) - c_{\gamma}^{\mathrm{UV}}(\mathbf{r})\right). \tag{4.10}$$

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 $g_{\mathrm{H}}^{\mathrm{UV}}(\mathbf{r})$ and $g_{\mathrm{O}}^{\mathrm{UV}}(\mathbf{r})$ grids. Each point on the $g_{\mathrm{H}}^{\mathrm{UV}}(\mathbf{r})$ will give the fractional density of water hydrogen a 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 [78, 79, 83]. 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 [83]. The convolution of the short-range terms in the integral equation (4.9) is calculated using 3D fast Fourier transform [84, 85]. Accordingly, the electrostatic asymptotics terms in the thermodynamics integral (4.12) below are handled analytically and reduced to one-dimensional integrals easy to compute [83].

With a converged 3D-RISM solution for h^{UV} and c^{UV}, it is straightforward to calculate solvation thermodynamics. From the perspective of molecular simulations, the most important thermodynamic values are the excess chemical potential of solvation (solvation free energy), μ^{ex} and the mean solvation force, $\mathbf{f}_i^{\mathrm{UV}}(\mathbf{R}_i)$, on each solute atom, i. μ^{ex} can be obtained through analytical thermodynamic integration for HNC,

$$\mu^{\mathrm{ex,HNC}} = k_{\mathrm{B}}T\sum_{\alpha}\rho_{\alpha}^{\mathrm{V}}\int d\mathbf{r}\left[\frac{1}{2}\left(h_{\alpha}^{\mathrm{UV}}(\mathbf{r})\right)^2 - c_{\alpha}^{\mathrm{UV}}(\mathbf{r}) - \frac{1}{2}h_{\alpha}^{\mathrm{UV}}(\mathbf{r})c_{\alpha}^{\mathrm{UV}}(\mathbf{r})\right], \tag{4.11}$$

KH ,

$$\mu^{\mathrm{ex,KH}} = k_{\mathrm{B}}T\sum_{\alpha}\rho_{\alpha}^{\mathrm{V}}\int d\mathbf{r}\left[\frac{1}{2}\left(h_{\alpha}^{\mathrm{UV}}(\mathbf{r})\right)^2\Theta\left(-h_{\alpha}^{\mathrm{UV}}(\mathbf{r})\right) - c_{\alpha}^{\mathrm{UV}}(\mathbf{r}) - \frac{1}{2}h_{\alpha}^{\mathrm{UV}}(\mathbf{r})c_{\alpha}^{\mathrm{UV}}(\mathbf{r})\right], \tag{4.12}$$

and PSE-n,

$$\begin{aligned}\mu^{\mathrm{ex,PSE}\text{-}n} = k_{\mathrm{B}}T\sum_{\alpha}\rho_{\alpha}^{\mathrm{V}}\int d\mathbf{r}\Bigg[&\frac{1}{2}\left(h_{\alpha}^{\mathrm{UV}}(\mathbf{r})\right)^2 - c_{\alpha}^{\mathrm{UV}}(\mathbf{r}) - \frac{1}{2}h_{\alpha}^{\mathrm{UV}}(\mathbf{r})c_{\alpha}^{\mathrm{UV}}(\mathbf{r}) \\ &- \frac{\left(t^*(\mathbf{r})\right)^{n+1}}{(n+1)!}\Theta\left(h_{\alpha}^{\mathrm{UV}}(\mathbf{r})\right)\Bigg],\end{aligned} \tag{4.13}$$

where Θ is the Heaviside function.

Analogous versions of Eqns. 4.6, 4.12 and 4.13 are used in 1D-RISM. While these are used for DRISM they are have been derived for XRISM. Furthermore, these equations have been derived a number of different ways with slightly different functional forms of the $-\frac{1}{2}hc$ term [77, 86–89]. These different functional forms are equivalent in XRISM but not in DRISM. The form introduced by Pettitt and Rossky [87] is the most popular in the literature and the default selection in `rism1d`. It is possible to have `rism1d` evaluate and output all three functional forms (see **??**) but, for DRISM, none of these expressions are strictly correct.

The force equation

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

is valid for all closures with a path independent expression for the excess chemical potential, such as

HNC, KH and PSE-n closures implemented in 3D-RISM [67, 90–92].

In addition to closure specific expressions for the solvation free energy, other approximations also exist. The Gaussian fluctuation (GF) approximation[93, 94] is given as

$$\mu^{\mathrm{ex,GF}} = k_{\mathrm{B}}T\sum_{\alpha}\rho_{\alpha}^{\mathrm{V}}\int d\mathbf{r}\left[-c_{\alpha}^{\mathrm{UV}}(\mathbf{r}) - \frac{1}{2}h_{\alpha}^{\mathrm{UV}}(\mathbf{r})\,c_{\alpha}^{\mathrm{UV}}(\mathbf{r})\right] \tag{4.14}$$

and has been shown to yield improved absolute solvation free energies for both polar and non-polar solutes[94, 95] but not necessarily for relative free energies[96]. It is not associated with a particular closure but is typically used in place of the expression for a given closure.

Eqs. (4.11)-(4.13) give the total solvation free energy, ΔG_{sol}, but it is often useful to decompose this into electrostatic (solvent polarization), ΔG_{pol}, and non-electrostatic (dispersion and cavity formation), $(\Delta G_{\mathrm{dis}} + \Delta G_{\mathrm{cav}})$, terms. Conceptually, we can divide the path of the thermodynamic integration into two steps: first the solute without partial charges is inserted into the solvent (dispersion and cavity formation) and then partial charges are introduced, which polarize the solvent,

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

ΔG_{sol} is produced by a 3D-RISM calculation on the charged solute. ΔG_{pol} is then the difference of the two calculations. As a point of reference, generalized-Born and Poisson-Boltzmann methods calculate only ΔG_{pol} and, typically, use a calculation involving solvent accessible surface area to predict $\Delta G_{\mathrm{dis}} + \Delta G_{\mathrm{cav}}$.

4.2 Practical Considerations

4.2.1 Computational Requirements and Parallel Scaling

Calculating a 3D-RISM solution for a single solute conformation typically requires about 100 times more computer time than the same calculation with explicit solvent or PB. While there are other factors to consider, such as sampling confined solvent or overall efficiency of sampling in the whole statistical ensemble at once, this can be prohibitive for many applications. Memory is also an issue as the 3D correlation grids require anywhere from a few megabytes for the smallest solutes to gigabytes for large complexes. A lower bound and very good estimate for the total memory required is

$$\text{Total memory} \geq 8\,\text{bytes}\times\left[N_{\mathrm{box}}\,N^{\mathrm{V}}\left(\underbrace{2N_{\mathrm{MDIIS}}}_{c\text{,residual}} + \underbrace{1}_{u} + \underbrace{N_{\mathrm{decomp}}}_{\text{polar decomp}}\;\underbrace{N_{\mathrm{propagate}}}_{\text{past solutions}}\right)\right.$$
$$\left.\left(N_{\mathrm{box}} + 2N_yN_z\right)\left\{\underbrace{4}_{\text{asymptotics}} + \underbrace{1}_{\text{FFT scratch}} + \underbrace{2}_{g,h}N^{\mathrm{V}}\right\}\right]$$

where $N_{\mathrm{box}} = N_x \times N_y \times N_z$ is the total number of grid points, N^{V} is the number of solvent atom species and N_{MDIIS} is the number of MDIIS vectors used to accelerate convergence. u^{UV}, c^{UV} and the residual of c^{UV} are stored in real-space only and require a full grid for each solvent. c^{UV} and its residual also require N_{MDIIS} grids for the MDIIS routine (see the mdiis_nvec keyword) and $N_{\mathrm{propagate}}$ grids to make use of solutions from previous solute configurations to improve the initial guess (see the npropagate keyword). If a polar/non-polar decomposition is requested (see the polardecomp keyword) an additional set of grids for past solutions with no solute charges is kept ($N_{\mathrm{decomp}} = 2$); by default this is turned off ($N_{\mathrm{decomp}} = 1$). The full real space grid plus an additional $2N_yN_x$ grid points are needed (due to the FFT) for g and h for each solvent species and for the four grids required to compute the long range asymptotics. Memory, therefore, scales linearly with N_{box} while computation time scales as $O(N_{\mathrm{box}}\log(N_{\mathrm{box}}))$ due to the requirements of calculating the 3D fast Fourier transform (3D-FFT). To overcome these requirements, two options are available beyond optimizations already in place, multiple time steps and parallelization. Multiple time step methods are available only in sander (Chapter 2) and are applicable to molecular dynamics calculations only. Parallelization is available for all calculations but is limited by system size and computational resources.

Both `sander` and `NAB` have MPI implementations of 3D-RISM (see Section **??** for NAB compiling instructions) that distribute both memory requirements and computational load. As memory is distributed, the aggregate memory of many computers can be used to perform calculations on very large systems. Memory distribution is handled by the FFTW 3.3 library so decomposition is done along the z-axis. If a variable solvation box size is used, the only consideration is to avoid specifying a large, prime number of processes (≥ 7). For fixed box sizes, the number of grids points in each dimension must be divisible by two (a general requirement) and the number of grid points in the z-axis must be divisible by the number of processes. `sander.MPI` also has the additional consideration that the number of processes cannot be larger than the number of solute residues; `NAB` does not suffer from this limitation.

4.2.2 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. As all file formats save only one density per file (see https://ambermd.org/FileFo
there is one file for each solvent atom type for each requested frame. For the default MRC format, each file is $(256 + N_{\mathrm{box}} \times 4)$ bytes, which can quickly fill disk space. Note that these file format use single precision floating point numbers.

4.2.3 Numerical Accuracy

Numerical accuracy depends on the residual tolerance specified for the numerical solution at runtime and the solvation box physical size and grid spacing. In most cases, you will need to test these parameters to ensure you have the accuracy required. As a rough guide, the numerical error in the solvation free energy is related to the tolerance by

$$\epsilon_{\Delta G_{\mathrm{solv}}} \approx 10 \times \text{tolerance.} \tag{4.15}$$

Molecular dynamics [67], minimization and trajectory post-processing [96] 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 is typically limited to **`drms`** $\geq 10^{-4}$. Trajectory post-processing for MM/RISM should use enough digits to obtain the necessary accuracy when differences in solvation free energy are computed. For example, if a error $< 0.2\,\mathrm{kcal/mol}$ is required for $\Delta\Delta G_{\mathrm{solv}}$, then ΔG_{solv} should be computed with an absolute error of $0.1\,\mathrm{kcal/mol}$. The relative error required to achieve this depends on the magnitude of ΔG_{solv}.

Almost all applications should use a grid spacing of 0.3 to 0.5 Å or smaller. 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).

4.2.4 Solution Convergence

The default parameters for 3D-RISM are selected to provide the best performance for the majority of systems. In cases where a convergence is not achieved, the strategies below may be useful.

4.2.4.1 Closure Bootstrap

When a PSE-*n* or HNC closure is desired, the most effective method to overcome convergence issues is to use a low order closure solution as a starting guess. The KH closure should be the starting point as it is numerically robust and, typically, converges easily in the vast majority of case. After this, higher orders of PSE-*n* can be used until the desired closure is reached. The procedure for 1D-RISM and 3D-RISM differs slightly in practice.

3D-RISM All 3D-RISM interfaces have closure bootstrapping builtin via the *closure* and *tolerance* keywords. Closures should be specified as an ordered list with last closure being the highest order closure. The solutions of the intermediate closures can have a high tolerance. The default tolerance for intermediate closures is 1 and there is no observed benefit to tolerances less than 1e-2.

4.2.4.2 MDIIS Settings

MDIIS default setting are appropriate for most cases. Should your residual diverge or the solver get stuck on a particular value, you can try modest adjustments.

Decrease *mdiis_del* *mdiis_del* controls the step size of MDIIS. A smaller step size can help convergence but if this is set too small it can cause convergence problems. For `rism1d`, this should be no lower than 0.1 or 0.2. For 3D-RISM, it should be 0.5 at the lowest.

Increase *mdiis_nvec* This is the number of trial solutions that are saved for predicting a new solution. The optimal number for rapid convergence is typically 10 for 3D-RISM and 20 for 1D-RISM. However, for 3D-RISM, the default choice of 5 requires much less memory and is computationally faster even though more iterations are required. Increasing the *mdiis_nvec* may help for 3D-RISM but is unlikely to help for 1D-RISM.

Increase *mdiis_restart* Occasionally, the MDIIS routine goes in the wrong direction and the residual increases significantly. If it increases more than *mdiis_restart* then the MDIIS routine selects the solution with the lowest residual and purges the other trial solutions. The default value of 10 can be too aggressive and cause the solver to cycle. Increasing the value to 100 or 1000 sometimes allows the solver to recover from a misstep.

4.2.4.3 Parameter Annealing

Chargeless, hot gases are the easiest systems to converge. For 1D-RISM, this can be used to bootstrap a solution in a similar manner to closure bootstrapping. By slowly turning on charges, lowering the temperature or increasing the density, a converged solution may be reached. This only works for 1D-RISM because it requires restarting from a previous solution. As with closure bootstrapping, files should be carefully renamed during the procedure. There is no general protocol but the parameter increment should be reduced as the target value is approached. E.g., turning on charges in a linear fashion usually isn't helpful.

4.2.4.4 Forcefield selection

The forcefield may affect convergence due to the number of solvent sites involved or the particular parameters of the forcefield.

Number of Sites Molecules with more sites are more difficult to converge. Six or more sites is already difficult to converge and more that 10 may not be possible under any circumstances. One solution is to use a united atom or coarse grained forcefields to reduce the number of sites.

Alternate Parameterization Some parameter sets simply yield a stiffer set of equations to solve. Choosing an alternate parameter set may allow convergence with only small differences in the numerical results. For example, the cSPC/E water model with SPC/E Joung/Cheatham ions is easier to converge at higher ion concentrations in 1D-RISM than cTIP3P water with TIP3P Joung/Cheatham ions. Both models give nearly identical results in RISM at lower concentrations but NaCl in cTIP3P water will not converge above 0.5 M for the PSE-3 closure despite using all of the above methods.

4.3 3D-RISM in sander

3D-RISM functionality is available in *sander* and is built as part of the standard install procedure. MPI functionality for 3D-RISM in sander requires some additional information at compile time, described in Section **??**. Some features specific to *sander* are discussed here.

4.3.1 3D-RISM in `sander`

Full 3D-RISM functionality is available in sander as part of the standard install procedure. However, some methods available in `sander` are not compatible with 3D-RISM, such as QM/MM simulations. At this time, only standard molecular dynamics, minimization and trajectory post-processing with non-polarizable force fields are supported. With the exception of multiple time step features, 3D-RISM keywords in sander are identical to those in NAB, rism3d.snglpnt and MMPBSA.py.

3D-RISM specific command line options for sander are

```
sander [standard options] -xvv xvvfile -guv guvroot -huv huvroot
    -cuv cuvroot -uuv uuvroot -asymp asympfile
    -quv quvroot -chgdist chgdistroot
    -exchem exchemroot -solvene solveneroot -entropy entropyroot -potUV potUVroot
```

xvvfile *input* description of bulk solvent properties, required for 3D-RISM calculations. Produced by `rism1d`.

guvroot *output* root name for solute-solvent 3D pair distribution function, $G^{\mathrm{UV}}(\mathbf{R})$. This will produce one file for each solvent atom type for each frame requested.

huvroot *output* root name for solute-solvent 3D total correlation function, $H^{\mathrm{UV}}(\mathbf{R})$. This will produce one file for each solvent atom type for each frame requested.

cuvroot *output* root name for solute-solvent 3D total correlation function, $C^{\mathrm{UV}}(\mathbf{R})$. This will produce one file for each solvent atom type for each frame requested.

uuvroot *output* root name for solute-solvent 3D potential energy function, $U^{\mathrm{UV}}(\mathbf{R})$, in units of kT. This will produce one file for each solvent atom type for each frame requested.

asympfile *output* root name for solute-solvent 3D long-range real-space asymptotics for C and H. This will produce one file for each of C and H for each frame requested and does not include the solvent site charge. Multiply the distribution by the solvent site charge to obtain the long-range asymptotics for that site.

quvroot *output* root name for solute-solvent 3D charge density distribution [e/Å]. This will produce one file that combines contributions from all solvent atom types for each frame requested.

chgdistroot *output* root name for solute-solvent 3D charge distribution [e]. This will produce one file that combines contributions from all solvent atom types for each frame requested.

exchemroot *output* root name for 3D excess chemical potential distribution files.

solveneroot *output* root name for 3D solvation energy distribution files.

entropyroot *output* root name for 3D solvation entropy distribution files.

potUVroot *output* root name for 3D solute-solvent potential energy distribution files.

Generated output files can be 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 `-guv guv` on the command line will produce two files on time step ten: `guv.O.10.mrc` and `guv.H1.10.mrc`.

4.3.1.1 Keywords

With the exception of irism, which is found in the &cntrl name list, all 3D-RISM options are specified in the &rism name list.

`irism` [0] Use 3D-RISM. Found in `&cntrl` name list.

- **= 0** Off.
- **= 1** On.

Closure Approximation

`closure` [KH] Comma separate list of closure approximations. If more than one closure is provided, the 3D-RISM solver will use the closures in order to obtain a solution for the last closure in the list when no previous solutions are available. The solution for the last closure in the list is used for all output.

- **= KH** Kovalenko-Hirata (KH).
- **= HNC** Hyper-netted chain equation (HNC).
- **=PSE*n*** Partial series expansion of order-*n* (PSE-*n*), where "*n*" is a positive integer.

Solvation Free Energy Corrections

`gfCorrection` [0] Compute the Gaussian fluctuation excess chemical potential functional (see §4.1.1).

- **= 0** Off.
- **= 1** On.

`pcpluscorrection` [0] Compute the PC+/3D-RISM excess chemical potential functional (see §**??**).

- **= 0** Off.
- **= 1** On.

`uccoeff` [0,0,0,0] Compute the UC excess chemical potential functional with the provided coefficients (see §**??**). *a* and *b* are the coefficients for the original UC functional, though using the closure excess chemical potential functional. *a1* and *b1* are optional and provide temperature dependence to the correction (UCT in [97]).

Fixed Box Size

`ng3` [] Sets the number of grid points for a fixed size solvation box. This is only used if buffer< 0.

`nx,ny,nz` Points for x, y and z dimensions.

`solvbox` [] Sets the size in Å of the fixed size solvation box. This is only used if buffer< 0. See §4.2.3 for details on how this affects numerical accuracy and how this interacts with `ljTolerance`, and `tolerance`.

`lx,ly,lz` Box length in x, y and z dimensions.

Solution Convergence

`tolerance` [1e-5] A list of maximum residual values for solution convergence. When used in combination with a list of closures it is possible to define different tolerances for each of the closures. This can be useful for difficult to converge. For the sake of efficiency, it is best to use as high a tolerance as possible for all but the last closure. For minimization a tolerance of 1e-11 or lower is recommended. See §4.2.3 for details on how this affects numerical accuracy and how this interacts with `ljTolerance`, `buffer`, and `solvbox`. Three formats of list are possible.

`one tolerance` All closures but the last use a tolerance of 1. The last tolerance in the list is used by the last closure. In practice this, is the most efficient.

`two tolerances` All closures but the last use the first tolerance in the list. The last tolerance in the list is used by the last closure.

n `tolerances` Tolerances from the list are assigned to the closure list in order.

`ljTolerance` [-1] Determines the Lennard-Jones cutoff distance based on the desired accuracy of the calculation. See §4.2.3 for details on how this affects numerical accuracy and how this interacts with `tolerance`, `buffer`, and `solvbox`.

`asympKSpaceTolerance` [-1] Determines the reciprocal space long range asymptotics cutoff distance based on the desired accuracy of the calculation. See §4.2.3 for details on how this affects numerical accuracy. Possible values are

< 0	`asympKSpaceTolerance=tolerance/10`,
`0`	no cutoff, and
> 0	given value determines the maximum error in the reciprocal-space long range asymptotics calculations.

`mdiis_del` [0.7] "Step size" in MDIIS.

`mdiis_nvec` [5] Number of vectors used by the MDIIS method. Higher values for this parameter can greatly increase memory requirements but may also accelerate convergence.

`mdiis_restart` [10] If the current residual is `mdiis_restart` times larger than the smallest residual in memory, then the MDIIS procedure is restarted using the lowest residual solution stored in memory. Increasing this number can sometimes help convergence.

`mdiis_method` [2] Specify implementation of the MDIIS routine.

= 0 Original. For small systems (e.g. $< 64^3$ grid points) this implementation may be faster than the BLAS optimized version.

= 1 BLAS optimized.

= 2 BLAS and memory optimized.

`maxstep` [10000] Maximum number of iterations allowed to converge on a solution.`nrespa`

`npropagate` [5] Number of previous solutions propagated forward to create an initial guess for this solute atom configuration.

- **= 0** Do not use any previous solutions
- **= 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).

Minimization and Molecular Dynamics

`zerofrc` [1] Redistribute solvent forces across the solute such that the net solvation force on the solute is zero.

- **= 0** Unmodified forces.
- **= 1** Zero net force.

Trajectory Post-Processing

`apply_rism_force` [1] Calculate and use solvation forces from 3D-RISM. Not calculating these forces can save computation time and is useful for trajectory post-processing.

- **= 0** Do not calculate forces.
- **= 1** Calculate forces.

Output

`ntwrism` [0] Indicates that solvent density grid should be written to file every `ntwrism` iterations.

- **= 0** No files written.
- **>= 1** Output every `ntwrism` time steps.

`molReconstruction` [0] For any thermodynamic distributions requested, also out the molecular reconstruction (see section **??**).

`volfmt` ['mrc'] Format of volumetric data files. May be mrc, ccp4, dx or xyzv (see section **??**).

`verbose` [0] Indicates level of diagnostic detail about the calculation written to the log file.

- **= 0** No output.
- **= 1** Print the number of iterations used to converge.
- **= 2** Print details for each iteration and information about what FCE is doing every `progress` iterations.

`write_thermo` [1] Print solvation thermodynamics in addition to standard sander output. The format is the same as that found in NAB and rism3d.snglpnt.

`polarDecomp` [0] Decomposes solvation free energy into polar and non-polar components. Note that this typically requires 80% more computation time.

- **= 0** No polar/non-polar decomposition.
- **= 1** Polar/non-polar decomposition.

`progress` [1] Display progress of the 3D-RISM solution every `kshow` iterations. 0 indicates this information will not be displayed. Must be used with `verbose` > 1.

4.3.1.2 Example

Molecular Dynamics (`imin=0`)

```
molecular dynamics with 3D-RISM and impulse MTS
&cntrl
    ntx=1, ntpr=100, ntwx=1000,ntwr=10000,
    nstlim=10000,dt=0.001,              !No shake or r-RESPA
    ntt=3, temp0=300, gamma_ln=20,      !Langevin dynamics
    ntb=0,                              !Non-periodic
    cut=999.,                           !Calculate all
                                        !solute-solute
                                        !interactions
    irism=1,
/
&rism
    rismnrespa=5,                       !r-RESPA MTS
    fcenbasis=10,fcestride=2,fcecrd=2   !FCE MTS
/
```

Minimization (`imin=1`)

```
Default XMIN minimization with 3D-RISM
&cntrl
    imin=1, maxcyc=200,
    drms=1e-3,          !RMS force. Can be as low as 1e-4
    ntmin=3,            !XMIN
    ntpr=5,
    ntb=0,              !Non-periodic
    cut=999.,           !Calculate all
                        !solute-solute interactions
    irism=1
/
&rism
    tolerance=1e-11,    !Low tolerance
    solvcut=9999,       !No cut-off for
                        !solute-solvent interactions
    centering=2         !Solvation box centering
                        !using center-of-geometry
/
```

Trajectory Post-Processing (`imin=5`)

```
Trajectory post-processing with 3D-RISM
&cntrl
    ntx=1, ntpr=1, ntwx=1,
    imin=5,maxcyc=1,       !Single-point energy calculation
                           !on each frame
    ntb=0,                 !Non-periodic
    cut=9999.,             !Calculate all
                           !solute-solute interactions
    irism=1
/
&rism
```

```
        tolerance=1e-4,         !Saves some time compared to 1e-5
        apply_rism_force=0,     !Saves some time. Forces are not used.
        npropagate=1            !Saves some time and 4*8*Nbox bytes
                                !of memory compared to npropagate=5.
    /
```

4.4 MoFT: analysis of volumetric data

MoFT is a series of computational programs and libraries for analysis of volumetric data generated by theoretical models (MD, MC simulations, 3D-RISM, NLPB) or derived from experimental measurement (e.g X-ray crystallography, cryo-EM). `metatwist` is an application that provides a low level access to most of the functionalities available in MoFT and is supported by metaFFT, a templated interface to FFTW library v3 (http://www.fftw.org/), that supports discrete Fourier transforms, correlations, convolutions on 1 or 3-dimensional data of float, double of complex datatypes.

Examples of MoFT usage and how to cite. The development of the functionalities available in MoFT has been driven by applied work which has been reported in the references bellow. Consider including these publications in your reference list when using MoFT:

1. "Ion counting from explicit-solvent simulations and 3D-RISM" GM Giambaşu, T Luchko, D Herschlag, DM York, DA Case **Biophysical Journal** 106 (4), 883-894 doi:10.1016/j.bpj.2014.01.021

2. "Competitive interaction of monovalent cations with DNA from 3D-RISM" GM Giambaşu, MK Gebala, MT Panteva, T Luchko, DA Case, DM York **Nucleic Acids Research** 43 (17), 8405-8415 doi:10.1093/nar/gkv830

3. "Predicting site-binding modes of ions and water to nucleic acids using molecular solvation theory" GM Giambaşu, DA Case, DM York **Journal of the American Chemical Society** doi:10.1021/jacs.8b11474

4.4.1 Usage

Most of the functionalities available in MoFT are exposed through the `metatwist` application, and include:

1. Reading, converting and writing `.dx` (OpenDX [1]), `.mrc` [2], `.ccp4` [3] volumetric data formats.

2. Dimensionality reduction of volumetric data:
   a) radial distribution functions using cylindrical and spherical frames of references (3D -> 1D).
   b) projection of 3D-data on x,y or z coordinates (3D -> 1D).
   c) worm plots (3D -> 1D), useful to characterize how density changes along curvilinear paths (such as channels) which are represented as B-splines and whose pivot points are provided by the user. The abscissa is the result of integrating the 3D density within a tube of specified radius around the curvilinear path. See [98] for examples of how worm plots can be used to analyze water and ion distribution in ion channels and G-quadruplexes.
   d) twisted, untwisted maps (3D -> 2D), meant to map the density of ions and water in an average plane of nucleic acid basepairs that are part of helical regions. Twisted maps are simply average densities in a plane perpendicular to the helical axis. Untwisted maps deconvolute this information with a mobile frame of reference that moves against the natural twist of the helical motif. See [99, 100] for examples of untwisted maps usage.

[1] antiquated format that is still widely used by most molecular graphics programs, see https://en.wikipedia.org/wiki/IBM_OpenDX.

[2] a common binary format in use by X-ray crystallography and cryo-EM, seehttps://www.ccpem.ac.uk/mrc_format/mrc2014.php

[3] another common, but older, binary format in use by X-ray crystallography and cryo-EM, see http://www.ccp4.ac.uk/html/maplib.html

3. Convolutions of volumetric data with several kernels, including Gaussian, sinc, box, Laplacian of a Gaussian, Butterworth filter for reduction of resolution range in the reciprocal space, crystallographic atomic form factors and densities to obtain to corresponding electron densities.
4. Transformations, including numerical derivatives (finite difference Laplacian), logarithm operators to compute potentials of mean force from equilibrium distributions.
5. Water and ion placement using Laplacian mapping.

`metatwist` has the following command line options:

`--help`	Produces help message.
`--dx`	Input density file(s): *.dx(gz,bz2)|*.mrc|*.ccp4.
`--ldx`	Input Laplacian file (*.dx|*.ccp4) for use with "-- map blobs(per)".
`--odx`	Output density file. File type is determined by extension: *.dx, *.mrc or *.ccp4.
`--map`	Mapping type: ~ cylindrical (1D): cylindrical RDF along z-axis. ~ twist (2D): twisted helical map along z-axis. ~ untwist (2D): untwisted helical map along z-axis. ~ spherical (1D): spherical RDF. ~ projxyz: (1D) project 3D-map on x,y,z axes. ~ excess: excess number of particles. ~ blobs: Laplacian blob analysis. ~ blobsper: Laplacian blob analysis on a periodic 3D-map. ~ rhoel (3D) : Electron density using atomic form factors. ~ rhoelreal (3D): Electron density using atomic densities. ~ cutresol (3D): Cut 3D-map resolution range.
`--bin`	Bin size for re-sampling (Å) .
`--(x|y|z|r)max`	Extent in the x,y, z or r directions (Å).
`--utrate`	Untwisting rate for use with "--map twist". Untwisting rate: 0.18587 rad/Å - BDNA 0.16870 rad/Å - TDNA 0.25590 rad/Å - ARNA (rad/Å).
`--com`	COM coordinates .
`--resolution`	Min and max resolution thresholds (in Å) for use with " -- map cutresol" (default "1.0 10.0", Å).
`--bulkdens`	Bulk density (M, mol/L, molar).
`--species`	Chemical species: atom, e.g. "N", or atom & residue, e.g. "O WAT", useful for water and ions placement as well as for computing electron densities.
`--sigma`	Convolution kernel width, sigma (in Å).
`--threshold`	Laplacian threshold. Sometimes not all the locally concentrated regions might be interesting. The threshold limits the region of interest to min(L[rho]) to threshold*min(L[rho]).
`--convolve`	Convolution type: (1) Gaussian, (2) box, (3) sinc, (4) Laplacian of Gaussian.
`--nlog`	Take the negative natural logarithm of the input density.
`--laplacian`	Compute Laplacian of the input density using finite difference.
`--average`	Average volumetric data when multiple datasets have been loaded. Otherwise, data will be accumulated.

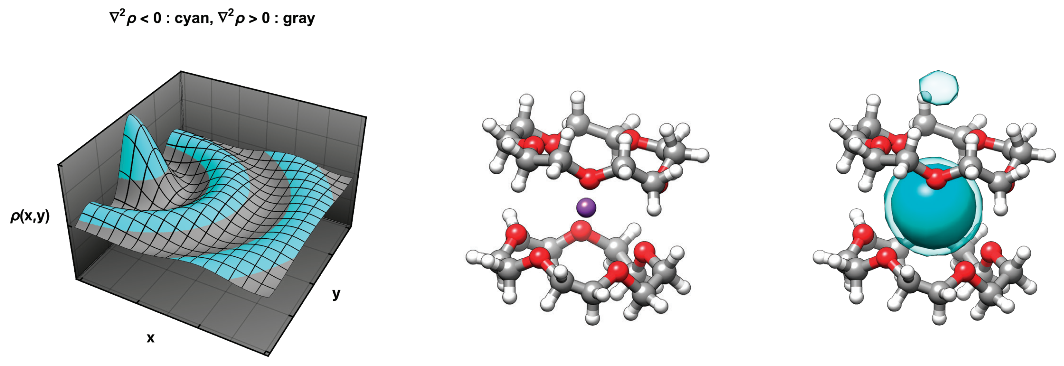


Figure 4.1: *(Left) The negative Laplacian (cyan) can be used to map locally concentrated regions of a density distribution, ρ. (Middle) A bis-crown ether (shown as CPK) binding mode to K+ (violet sphere) determined using density distributions obtained using RISM and located using Laplacian mapping in MoFT. (Right) Two level sets of the Laplacian of the K+ distribution, one using a threshold (see documentation) of 0.1 (solid cyan) and the other using a threshold of 0.01 (semi-transparent cyan).*

4.4.2 Examples

All files relevant for these examples are available in $MSANDERHOME/src/moft/examples/.

Water and ion placement using Laplacian mapping

We will use MoFT to locate and map tightly bound solution particles to a solute molecule of interest using molecular distribution functions obtained from 3D-RISM. Specifically, we will try to locate K+ binding mode(s) to a small molecule ionophore - a crown ether.

Generally, molecular density distributions of ions and water have alternating regions where they are highly concentrated and others where they are locally depleted. While these complex topologies are a benefit of models that include particle-particle correlations (such as explicit solvent MD, RISM) they make determination of boundaries of "binding modes" a complex task. Our solution is to demarcate these binding modes using the Laplacian of the solvent distributions. When applied to 3D distributions, the Laplacian measures the difference between the local particle density and the average of the density in a small neighborhood of that point. Hence, where the Laplacian is positive the local particle density is *locally depleted*, while for the regions with negative values of the Laplacian the particle density is *locally concentrated*. Experience shows that a pre-conditioning using kernels that smooth out small local variations in the density can help eliminate false positives. Here we will apply the Laplacian mapping on the density convolution with a 3D Gaussian which can be carried out in a single step by a convolution with a Laplacian of a Gaussian kernel.

(1) Generate density distributions. Solvent density distributions can be determined by several means, but here RISM is used (See 4 for how to run RISM). When running RISM, make sure to specify the "--guv" keyword to have the solution components density distributions outputted:

```
rism3d.snglpnt --prmtop bc5-k.parm7 --xvv KCl-aq-0.2M-pse3.xvv \
               --closure pse1,pse2,pse3 --tolerance 1e-03,1e-06 \
               --ng 192,192,192 --solvbox 96,96,96 --buffer -1 \
               --mdiis_del 0.5 --mdiis_nvec 10 \
               --verbose 2 --npropagate 0 --guv g > rism.out
```

(2) Compute the Laplacian map. First, one has to take the Laplacian of the distribution, using the "convolve" option:

```
metatwist --dx g.K+.1.dx.bz2 --odx lp-K+.dx --species K+ K+ --bulkdens 0.2\
          --convolve 4 --sigma 1.0
```

Here, --dx specifies the input density, --odx the root of the output file containing the Laplacian density. Option "--convolve 4" specifies the type of convolution that leads to the Laplacian; here we have chosen to obtain the Laplacian using a convolution with the Laplacian of a Gaussian, in this case of width 1.0, specified using "–sigma 1.0". This step produces a "convolution-lp-K+.dx" file that can be visualized in your molecular graphics application and will be used in the next step.

(3) Solvent Placement. Second, using the determined Laplacian, we can proceed to the actual analysis:

```
metatwist --dx g.K+.1.dx.bz2 --ldx convolution-lp-K+.dx --species K+ K+ \
          --bulkdens 0.2 --map blobs --thresh 0.1
```

Here, --ldx specifies the input Laplacian, "--map blobs" asks for solvent placement analysis to be carried out (you can think about solvent binding modes as blobs) using a Laplacian threshold of 0.1. While all the regions of space having a negative Laplacian can be considered as "locally concentrated", often a tighter (more negative) threshold can simplify the analysis. Lastly, --bulkdens specifies the concentration of the solution particle; in this case K+ has a bulk concentration of 0.2M. With these settings, a pdb file named "g.K+.1-convolution-lp-K+-blobs-centroid.pdb" is produced that contains the coordinates of the centroid of each solvation binding mode (in this case only one mode has been found), its occupancy and temperature factor.

```
ATOM      1  K+   K+ C   1      10.926  12.084   4.026  0.10 92.73          K+
```

Converting particle density distributions to electron densities

It is often necessary to convert particle density distributions to electron densities to directly compare against experimentally derived data, such as that obtained from X-ray crystallography. To illustrate this functionality, we will use the aforementioned RISM calculation on the crown ether immersed in a KCl aqueous solution which produced density distributions for K+, Cl-, water H and water O. In the first step, each of the particle densities is converted to their corresponding electron densities using model atomic factors used in crystallography. In a second stage, all the electron densities are accumulated into a resulting total electron density. Note the use of " --species" option to guide the choice of model density based on the ionization or oxidation number of each atom as well as the " --map rhoel" option to ask for computation of the electron density map. A similar option "--map rhoelreal" could be used which instead of atomic factors will use reference atomic densities to compute the overall electron density.

```
# (1) convert each particle density to electron densities :
metatwist --dx g.K+.1.dx.bz2  --species K+  --odx rho.K+.1.dx  --map rhoel --bulkden
metatwist --dx g.Cl-.1.dx.bz2 --species Cl- --odx rho.Cl-.1.dx --map rhoel --bulkden
metatwist --dx g.O.1.dx.bz2   --species O2- --odx rho.O.1.dx   --map rhoel --bulkden
# (2) assembly of all densities into rho.dx :
metatwist --dx rho.Cl-.1.dx   rho.K+.1.dx    rho.O.1.dx --odx rho.dx --species none
```

5 sqm: Semi-empirical quantum chemistry

AmberTools contains its own quantum chemistry program, called *sqm*. This is code extracted from the QM/MM portions of *sander*, but is limited to "pure QM" calculations. A principal current use is as a replacement for MOPAC for deriving AM1-bcc charges, but the code is much more general than that. Presently, it is limited to single point calculations and energy minimizations (geometry optimizations) for closed-shell systems. It supports a wide variety of semi-empirical Hamiltonians, including many recent ones. An external electric field generated by a set of point charges can be included for single point calculations. Our plan is to add capabilities to subsequent versions. The major contributors are as follows:

- The original semi-empirical support was written by Ross Walker, Mike Crowley, and Dave Case,[101] based on public-domain MOPAC codes of Various SCF convergence schemes were added by Tim Giese and Darrin York.
- The PM6 Hamiltonian was added by Andreas Goetz and dispersion and hydrogen bond corrections were added by Andreas Goetz and Kyoyeon Park.
- The extension for MNDO type Hamiltonians to support d orbitals was written by Tai-Sung Lee, Darrin York and Andreas Goetz.
- The charge-dependent exchange-dispersion corrections of vdW interactions[102] was contributed by Tai-Sung Lee, Tim Giese, and Darrin York.
- Support for reading user-defined parameters for NDDO methods was added by Tai-Sung Lee and Darrin York.

5.1 Available Hamiltonians

Available MNDO-type semi-empirical Hamiltonians are PM3,[103] AM1,[104] RM1,[105] MNDO,[106] PDDG/PM3,[107] PDDG/MNDO,[107] PM3CARB1,[108], PM3-MAIS[109, 110], MNDO/d[111–113], AM1/d (Mg from AM1/d[114] and H, O, and P from AM1/d-PhoT[115]) and PM6[116].

Also available is the density functional theory-based tight-binding (DFTB) Hamiltonian[117–119] and its self-consistent-charge version with Taylor expansion up to second order (SCC-DFTB or DFTB2)[120] and third-order (DFTB3)[121]. If you use the mio-1-1 parameters for DFTB2, you can add an empirical correction for dispersion effects[122] and calculate CM3 charges[123] (both only for elements H, C, N, O, S, P). Diagonal third-order corrections are available for DFTB2[124] with mio-1-1 parameters but it is recommended to perform full DFTB3 simulations instead. Neither dispersion corrections nor halogen corrections are implemented for DFTB3.

The elements supported by each QM method are:

- MNDO: H, Li, Be, B, C, N, O, F, Al, Si, P, S, Cl, Zn, Ge, Br, Cd, Sn, I, Hg, Pb
- MNDO/d: H, Li, Be, B, C, N, O, F, Na, Mg, Al, Si, P, S, Cl, Zn, Ge, Br, Sn, I, Hg, Pb
- AM1: H, C, N, O, F, Al, Si, P, S, Cl, Zn, Ge, Br, I, Hg
- AM1/d: H, C, N, O, F, Mg, Al, Si, P, S, Cl, Zn, Ge, Br, I, Hg
- PM3: H, Be, C, N, O, F, Mg, Al, Si, P, S, Cl, Zn, Ga, Ge, As, Se, Br, Cd, In, Sn, Sb, Te, I, Hg, Tl, Pb, Bi

- PDDG/PM3: H, C, N, O, F, Si, P, S, Cl, Br, I
- PDDG/MNDO: H, C, N, O, F, Cl, Br, I
- RM1: H, C, N, O, P, S, F, Cl, Br, I
- PM3CARB1: H, C, O
- PM3-MAIS: H, O, Cl
- PM6: H, He, Li, Be, B, C, N, O, F, Ne, Na, Mg, Al, Si, P, S, Cl, Ar, K, Ca, Sc, Ti, V, Cr, Mn, Fe, Co, Ni, Cu, Zn, Ga, Ge, As, Se, Br, Kr, Rb, Sr, Y, Zr, Nb, Mo, Tc, Ru, Rh, Pd, Ag, Cd, In, Sn, Sb, Te, I, Xe, Cs, Ba, La, Lu, Hf, Ta, W, Re, Os, Ir, Pt, Au, Hg, Tl, Pb, Bi

The PM6 implementation has not been extensively tested for all available elements. Please check your results carefully, possibly by comparison to other codes that implement PM6, if transition metal elements are present. SCF convergence may be more difficult to achieve for transition metal elements with partially filled valence shells.

If the PM6 Hamiltonian is used in a QM/MM simulation with *sander* using electrostatic embedding (see Section **??**) or if an electric field of external point charges is used, then the electrostatic interactions between QM and MM atoms are modeled using the MNDO type core repulsion function for interactions between QM and MM atoms. Parameters for the exponents α of the QM atoms are taken from PM3 (a default value of five is used for the exponents α of the MM atoms as is the case for MNDO, AM1 and PM3). Since PM3 does not have parameters for all elements that are supported by PM6, the missing exponents were defined in an ad hoc manner (see the source code in $AMBERHOME/AmberToosl/src/sqm/qm2_parameters.F90, variable alp_pm6). The magnitude of the coefficients α is probably not critical for the accuracy of QM/MM calculations but this should be tested on a case by case basis. This does not affect QM calculations with *sqm*.

5.2 Dispersion and hydrogen bond correction

An empirical dispersion and hydrogen bonding correction is implemented for the MNDO type Hamiltonians AM1 and PM6[125]. The empirical dispersion correction follows the formalism for DFT-D[126] and consists of a physically sound r^{-6} term that is damped at short distances to avoid the short-range repulsion which can be written as

$$E_{\mathrm{dis}} = -s_6 \sum_{ij} f_{\mathrm{d}amp}(r_{ij}, R^0_{ij}) C_{6,ij} r_{ij}^{-6}, \tag{5.1}$$

where r_{ij} is the distance between two atoms i and j, R^0_{ij} is the equilibrium van der Waals (vdW) separation derived from the atomic vdW radii, $C_{6,ij}$ the dispersion coefficient, and s_6 a general scaling factor. The damping function is given as

$$f_{\mathrm{d}amp}(r_{ij}, R^0_{ij}) = \left[1 + \exp\left(-\alpha \frac{r_{ij}}{s_R R^0_{ij}} - 1\right)\right]^{-1}. \tag{5.2}$$

Bondi vdW radii[127] are used and for a pair of unlike atoms we have

$$R^0_{ij} = \frac{{R^0_{ii}}^3 + {R^0_{jj}}^3}{{R^0_{ii}}^2 + {R^0_{jj}}^2}. \tag{5.3}$$

For the C_6 coefficients the following equation is used,

$$C_{6,ij} = 2\frac{(C_{6,ii}^2 C_{6,jj}^2 N_{eff,i} N_{eff,j})^{1/3}}{(C_{6,ii} N_{eff,j}^2)^{1/3} + (C_{6,jj} N_{eff,i}^2)^{1/3}}, \tag{5.4}$$

where the Slater-Kirkwood effective number of electrons $N_{eff,i}$ and the C_6 coefficients can easily be found in the literature[126].

An empirical hydrogen bonding correction[125] that is transferable among different semiempirical Hamiltonians and has been parametrized for use with the dispersion correction described above is also available. This correction does not make the assumption of a specific acceptor/hydrogen/donor binding situation. Instead it considers the hydrogen bond as a charge-independent atom- atom term between two atoms capable of serving as an acceptor or donor (for example, O, N) and weights this by a function that accounts for the steric arrangement of the two atoms and the favorable positioning of a hydrogen atom inbetween. A damping function corrects for long- and short-range behavior,

$$E_{\mathrm{H}-bond} = \frac{C_{AB}}{r_{AB}^2} f_{geom} f_{\mathrm{d}amp}, \tag{5.5}$$

$$f_{geom} = \cos(\theta_A)^2 \cos(\phi_A)^2 \cos(\psi_A)^2 \cos(\phi_B)^2 \cos(\phi_B)^2 \cos(\psi_B)^2 f_{\mathrm{bond}}, \tag{5.6}$$

$$f_{\mathrm{bond}} = 1 - \frac{1}{1+\exp[-60(r_{XH}/1.2-1)]}, \tag{5.7}$$

$$f_{\mathrm{damp}} = \left(\frac{1}{1+\exp[-100(r_{AB}/2.4-1)]}\right)\left(1 - \frac{1}{1+\exp[-10(r_{AB}/7.0-1)]}\right), \tag{5.8}$$

$$C_{AB} = \frac{C_A + C_B}{2}. \tag{5.9}$$

Here, C_A and C_B are the atomic hydrogen bonding correction parameters and the (torsion) angles in the function f_{geom} are defined similarly to an earlier hydrogen bond correction[128].

The hydrogen bond correction can be used both for single point energy calculations or geometry optimizations with SQM and for molecular dynamics simulations with SANDER. However, we do not recommend the use for molecular dynamics at present since cutoffs needed to be implemented for the calculation of f_{geom} of equation (5.6). This and some other conditional evaluations give rise to discontinuities in the potential energy surface and thus make this method unattractive for MD simulations.

5.3 Usage

The *sqm* program uses the following simple command line:

```
sqm [-O] -i <input-file> -o <output-file>
```

mdin is the default input-file name, and *mdout* is the default output-file name.As in other Amber programs, the "-O" flag allows the program to over-write the output file.

An example input file for running a simple minimization is shown here:

```
Run semi-empirical minimization
 &qmmm
  qm_theory='AM1',   qmcharge=0,
 /
    6    CG       -1.9590        0.1020        0.7950
    6   CD1       -1.2490        0.6020       -0.3030
    6   CD2       -2.0710        0.8650        1.9630
    6   CE1       -0.6460        1.8630       -0.2340
    6    C6       -1.4720        2.1290        2.0310
```

```
   6    CZ       -0.7590        2.6270        0.9340
   1   HE2       -1.5580        2.7190        2.9310
  16   S15       -2.7820        0.3650        3.0600
   1   H19       -3.5410        0.9790        3.2740
   1   H29       -0.7870       -0.0430       -0.9380
   1   H30        0.3730        2.0450       -0.7840
   1   H31       -0.0920        3.5780        0.7810
   1   H32       -2.3790       -0.9160        0.9010
```

The *&qmmm* namelist contains variables that allow you to control the options used. Following that is one line per atom, giving the atomic number, atom name, and Cartesian coordinates (free format). The variables in the *&qmmm* namelist are these:

`qm_theory` Level of theory to use for the QM region of the simulation (Hamiltonian). Default is to use the semi-empirical Hamiltonian PM3. Options are AM1, RM1, MNDO, PM3-PDDG, MNDO-PDDG, PM3-CARB1, MNDO/d (same as MNDOD), AM1/d (same as AM1D), PM6, DFTB2 (same as DFTB), and DFTB3. The dispersion correction can be switched on for AM1 and PM6 by choosing AM1-D* and PM6-D, respectively. The dispersion and hydrogen bond correction will be applied for AM1-DH+ and PM6-DH+.

`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 use analytical or numerical derivatives of the semiempirical electron repulsion integrals. 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 ntpr steps.

= 1 Print SCF energy at every step to many more significant figures than usual. Also print the number of SCF cycles needed on each step.

= 2 As 1 and also print info about memory reallocations, number of pairs per QM atom, QM core - QM core energy, QM core - MM atom energy, and total energy.

= 3 As 2 and also print SCF convergence information at every step.

= 4 As 3 and also print forces on the QM atoms due to the SCF calculation and the coordinates of the link atoms at every step.

= 5 As 4 and 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(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 will be used during the SCF procedure. This is an advanced option to fine-tune performance which has negligible effect on energies (and generally little effect on geometries in the case of SQM energy minimizations). The speed of each diagonalizer is a function of the number and type of QM atoms as well as the LAPACK library that the program 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 the program will test each diagonalizer in turn, including the pseudo diagonalizer, and select the one that gives optimum performance. As of AmberTools 15 diag_routine = 0 is the default for both SQM and QMMM in Sander. Not available for DFTB/SCC-DFTB.

= 0 Automatically select the fastest routine (default).

= 1 Use internal diagonalization routine.

= 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.

`print_eigenvalues` Controls printing of MO eigenvalues.

= 0 Do not print MO eigenvalues

= 1 Print MO eigenvalues at the end of a single point calculation or geometry optimization (default)

= 2 Print MO eigenvalues at the end of every SCF cycle (only NDDO methods, not DFTB)

= 3 Print MO eigenvalues during each step of the SCF cycle (only NDDO methods, not DFTB)

`qxd` Flag to turn on (=.true.) or off (=.false., default) the charge-dependent exchange-dispersion corrections of vdW interactions[102].

`parameter_file` **=** 'PARAM.FILE' Read user-defined parameters from the file 'PARAM.FILE'. The first three space-separated entries (case insensitive) of each line will be interpreted as a user-modified parameter in the sequence of *parameter name*, *element name*, and *value*. For example, a line contains "USS Cl -111.6139480D0 " will cause the USS parameter of the Cl element changed to -111.6139480. A line beginning with "END" will stop the reading. This function currently only works for MNDO, AM1, PM3, MNDO/d, and AM1/d. Also, when new nuclear core-core parameters (FN, in PM3, AM1, and AM1/d) are re-defined, the number of FNN parameter sets (NUM_FN) also needs to be defined. For example, if FN*n*3 (*n* = 1, 2, or 3) is defined, then NUM_FN needs to be set to 3 or 4.

`peptide_corr` **= 0** Don't apply MM correction to peptide linkages. (default)

= 1 Apply a MM correction to peptide linkages. This correction is of the form $E_{scf} = E_{scf} + h_{type}(i_{type}) \sin^2 \phi$, where ϕ is the dihedral angle of the H-N-C-O linkage and h_{type} is a constant dependent on the Hamiltonian used. (Recommended, except for DFTB/SCC-DFTB.)

`itrmax` Integer specifying the maximum number of SCF iterations to perform before assuming that convergence has failed. Default is 1000. Typically higher values will not do much good since if the SCF hasn't converged after 1000 steps it is unlikely to. If the convergence criteria have not been met after itrmax steps the SCF will stop and the minimisation will proceed with the gradient at itrmax. Hence if you have a system which does not converge well you can set itrmax smaller so less time is wasted before assuming the system won't converge. In this way you may be able to get out of a bad geometry quite quickly. Once in a better geometry SCF convergence should improve.

`maxcyc` Maximum number of minimization cycles to allow, using the *xmin* minimizer (see Section 14.5) with the TNCG method. Default is 9999. Single point calculations can be done with *maxcyc* = 0.

`ntpr` Print the progress of the minimization every *ntpr* steps; default is 10.

`grms_tol` Terminate minimization when the gradient falls below this value; default is 0.02

`ndiis_attempts` Controls the number of iterations that DIIS (direct inversion of the iterative subspace) extrapolations will be attempted. Not available for DFTB/SCC-DFTB. The SCF does not even begin to exhaust its attempts at using DIIS extrapolations until the end of iteration 100. Therefore, for example, if ndiis_attempts=50, then DIIS extrapolations would be performed at end of iterations 100 to 150. The purpose of not performing DIIS extrapolations before iteration 100 is because the existing code base performs quite well for most molecules; however, if convergence is not met after 100 iterations, then it is presumed that further iterations will not yield SCF convergence without doing something different, i.e., DIIS. Thus, the implementation of DIIS in SQM is a mechanism to try and force SCF convergence for molecules that are otherwise difficult to converge. Default 0. Maximum 1000. Minimum 0. Note that DIIS will automatically turn itself on for 100 attempts at the end of iteration 800 even if you did not explicitly set ndiis_attempts to a nonzero value. This is done as a final effort to achieve convergence.

`ndiis_matrices` Controls the number of matrices used in the DIIS extrapolation. Including only one matrix is the same as not performing an extrapolation. Including an excessive number of matrices may require a large amount of memory. Not available for DFTB/SCC-DFTB. Default 6. Minimum 1. Maximum 20.

`vshift` Controls level shifting (only NDDO methods, not DFTB). Virtual orbitals can be shifted up by vshift (in eV) to improve SCF convergence in cases with small HOMO/LUMO gap. Default 0.0 (no level shift).

`errconv` SCF tolerance on the maximum absolute value of the error matrix, i.e., the commutator of the Fock matrix with the density matrix. The value has units of hartree. The default value of errconv is sufficiently large to effectively remove this tolerance from the SCF convergence criteria. Not available for DFTB/SCC-DFTB. Default 1.d-1. Minimum 1.d-16. Maximum 1.d0.

`qmmm_int` When running QM calculations in the sqm program, an electric field of external point charges can be added. In this way, the electrostatic effect outside of the QM region can be modeled, making the calculation a simplified QM/MM calculation without QM/MM vdW's contribution. Like QM/MM calculations (see Section **??**), the method to couple QM and MM electrostatic interactions for external charges and semiempirical Hamiltonians can be specified via the *qmmm_int* namelist variable.

The current implementation limits use of external charges to only single point energy calculations. To run such a calculation, an additional field, which begins with #EXCHARGES and ends with #END, is required to specify the external point charges in the input. Each external point charge must include atomic number, atom name, X, Y, Z coordinates and the charge in units of the electron charge. An example input looks like:

```
single point energy calculation (adenine), with external charges (thymine)
&qmmm
 qm_theory = 'PM3',
 qmcharge = 0,
 maxcyc = 0,
 qmmm_int = 1,
/
7  N   1.0716177  -0.0765366   1.9391390
1  H   0.0586915  -0.0423765   2.0039181
1  H   1.6443796  -0.0347395   2.7619159
6  C   1.6739638  -0.0357766   0.7424316
7  N   0.9350155  -0.0279801  -0.3788916
6  C   1.5490760   0.0012569  -1.5808009
1  H   0.8794435   0.0050260  -2.4315709
7  N   2.8531510   0.0258031  -1.8409596
6  C   3.5646109   0.0195446  -0.7059872
6  C   3.0747955  -0.0094480   0.5994562
7  N   4.0885824  -0.0054429   1.5289786
6  C   5.1829921   0.0253971   0.7872176
1  H   6.1882591   0.0375542   1.1738824
7  N   4.9294871   0.0412404  -0.5567274
1  H   5.6035368   0.0648755  -1.3036811
#EXCHARGESwill be
6  C  -4.7106131   0.0413373   2.1738637  -0.03140
1  H  -4.4267056   0.9186178   2.7530256   0.06002
1  H  -4.4439282  -0.8302573   2.7695655   0.05964
```

```
1  H  -5.7883971   0.0505530   2.0247280   0.03694
6  C  -3.9917387   0.0219348   0.8663338  -0.25383
6  C  -4.6136833   0.0169051  -0.3336520   0.03789
1  H  -5.6909220   0.0269347  -0.4227183   0.16330
7  N  -3.9211729  -0.0009646  -1.5163659  -0.47122
1  H  -4.4017172  -0.0036078  -2.4004924   0.35466
6  C  -2.5395897  -0.0149474  -1.5962357   0.80253
8  O  -1.9416783  -0.0291878  -2.6573783  -0.63850
7  N  -1.9256484  -0.0110593  -0.3638948  -0.58423
1  H  -0.8838255  -0.0216168  -0.3784269   0.35404
6  C  -2.5361367   0.0074651   0.8766724   0.71625
8  O  -1.8674730   0.0112093   1.9120833  -0.60609
#END
```

6 Atom and Residue Selections

There are three ways to select atoms and residues in AMBER-related routines: the **AMBER "mask"** notation, used by most programs, the **NAB "atom expressions"**, which work only with NAB-compiled applications, and an older "GROUP" specification used in *sander* and *pmemd*. Information about these is collected in this chapter.

6.1 Amber Masks

A "mask" is a notation which selects atoms or residues for special treatment. A frequent usage is fixing or tethering selected atoms or residues during minimization or molecular dynamics.

The following lines are partially copied from the original AMBER documentation. For more details, refer to the entire section of that documentation describing the *ambmask* utility.

The "mask" selection expression is composed of "elementary selections". These start with "**:**" to select by residues, or "@" to select by atoms. Residues can be selected by numbers (given as numbers separated by commas, or as ranges separated by a dash) or by names (given as a list of residue names separated by commas). The same holds true for atom selections by atom numbers or atom names. In addition, atoms can be selected by AMBER atom type, in which case "@" must be immediately followed by "%". The notation ":*" means all residues and "@*" means all atoms. The following examples show the usage of this syntax.

Residue Number List Examples

```
:1-10      = "residues 1 to 10"
:1,3,5     = "residues 1, 3, and 5"
:1-3,5,7-9 = "residues 1 to 3 and residue 5 and residues 7 to 9"
```

Residue Name List Examples

```
:LYS          = "all lysine residues"
:ARG,ALA,GLY  = "all arginine and alanine and glycine residues"
```

Atom Number List Examples

Note that these masks use the **actual sequential numbers of atoms** in the file. This is tricky and a serious source of error. You must know these numbers correctly. Using the atom numbers of a PDB file written out by an AMBER tool is an appropriate way to avoid pitfalls. **Do not use the original atom numbers from the raw PDB file you started with.**

```
@12,17       = "atoms 12 and 17"
@54-85       = "all atoms from 54 to 85"
@12,54-85,90 = "atom 12 and all atoms from 54 to 85 and atom 90"
```

Atom Name List Examples

```
@CA         = all atoms with the name CA (i.e., all C-alpha atoms)
@CA,C,O,N,H = all atoms with names CA or C or O or N or H
                    (i.e., the entire protein backbone)
```

Atom Type List Examples

This last mask type is only used by specialists and mentioned here for completeness. It allows the selection of AMBER atom types and requires detailed knowledge of AMBER force fields.

```
@%CT          = all atoms with the force field type CT
                  (the standard sp3 aliphatic carbon)
@%N*,N3       = all atoms with the force field type N* or N3
                  (N* is a special sp2 nitrogen, N3 is an sp3 nitrogen)
```

Note that in the above example, `N*` is actually an atom type. The * is **not** a wild card meaning "all N-something types"!

Logical Combinations

The selections above can be combined by various logical operators, including selections like "all atoms within a certain distance from...". The use of such combinations goes beyond this introductory script. Interested users should refer to the next section for details.

6.1.1 ambmask

NAME

ambmask - test group input FIND mask (or mask string given in the &cntrl section) and dump the resulting atom selection in a given format

SYNOPSIS

```
    ambmask -p prmtop -c inpcrd -prnlev [0-3] -out [short| pdb| amber] -find [maskstr]
```

DESCRIPTION

ambmask acts as a filter that inputs an Amber topology file and an Amber coordinate file and applies the "maskstr" selection string to select specific atoms or residues. (The "maskstr" selection string is similar syntactically to UCSF Chimera/Midas.) Residues can be selected by their numbers or names. Atoms can be selected by numbers, names, or Amber (forcefield) type. Selections are case insensitive. The selected atoms are printed to **stdout** (by default, in Amber-style PDB format). Atom and residue names and numbers are taken from the Amber topology. Beware that the selection string works on those names and not the ones from the original PDB file. If you are not sure how atoms or residues are named or numbered in the Amber topology, use **ambmask** with a selection string ":*" (which is the default) to dump the whole PDB file with corresponding Amber atom/residue names and numbers.

The "maskstr" selection expression is composed of "elementary selections". These start with ":" to select by residues, or "@" to select by atoms. Residues can be selected by numbers (given as numbers separated by commas, or as ranges separated by a dash) or by names (given as a list of residue names separated by commas). The same holds true for atom selections by atom numbers or atom names. In addition, atoms can be selected by Amber atom type, in which case "@" must be immediately followed by "%". ":*" means all residues and "@*" means all atoms. The following examples show the usage of this syntax. Square brackets should not be used in actual expressions, they are only used below to denote individual selection string examples:

```
:{residue numlist} [:1-10] [:1,3,5] [:1-3,5,7-9]
:{residue namelist} [:LYS] [:ARG,ALA,GLY]
@{atom numlist} [@12,17] [@54-85] [@12,54-85,90]
@{atom namelist} [@CA] [@CA,C,O,N,H]
@%{atom typelist} [@%CT] [@%N*,N3]
```

These "elementary selections" can be combined into more complex selections using binary operators "&" (and) and "|" (or), unary operator "!" (negation), distance binary operators "<:", ">:", "<@", ">@", and parentheses. Spaces around operators are irrelevant. Parentheses have the highest priority, followed by distance operators ("<:", ">:", "<@", ">@"), "!" (negation), "&" (and) and "|" (or) in order of descending priority. A wildcard "=" in an atom or residue name matches any name starting with a given character (or characters). For example, [:AS=] would match all aspartic acid residues (ASP), and asparagines (ASN); [@H=] would match all atom names starting with H (which are effectively all hydrogens). It cannot be used to match the end part of names (such as [:=A]). Some examples of more complex selections follow:

```
[@C= & !@CA,C]
```

.. all carbons except backbone alpha and carbonyl carbon

```
[(:1-3@CA | :5-7@CB)]
```

.. alpha carbons in residues 1-3 and beta carbons in residues 5-7

```
[:CYS,ARG & !(:1-10 | @CA,CB)]
```

.. all CYS and ARG atoms except those which are in residues 1-10 and which are CA or CB

```
[:* & !@H=] or [!@H=]
```

.. all heavy atoms (i.e. except hydrogens)

```
[:5 <@4.5]
```

.. all atoms within 4.5A from residue 5

```
[(:1-55 <:3.0) & :WAT]
```

.. all water molecules within 3A from residues 1-55

Compound expressions of the following type are also allowed:

```
:{residue numlist|namelist}@{atom numlist|namelist|typelist}
[:1-10@CA] is equivalent to [:1-10 & @CA]
[:LYS@H=] is equivalent to [:LYS & @H=]
```

OPTIONS

The program needs an Amber topology file and coordinates (restrt format). The filename specified with the *-p* option is Amber topology, while the filename given with the *-c* option is a coordinate file. If *-p* or *-c* options are not given, the program expects that files "prmtop" and/or "inpcrd" exist in the current directory, which will be taken as topology and coordinate files correspondingly. If no command line options are given, the program prints the usage statement.

The option *-prnlev* specifies how much (debugging) information is printed to **stdout**. If it is 0, only selected atoms are printed. More verbose output (which might be useful for debugging purposes) is achieved with higher values: 1 prints original "maskstr" in its tokenized (with operands enclosed in square brackets) and postfix (or Reverse Polish Notation) forms; number of atoms and residues in the topology file and number of selected atoms are also printed to **stdout**. 2 prints the resulting mask array, which is an array of integer values, with '1' representing a selected atom, and '0' an unselected one. Value of 3, in addition, prints mask arrays as they are pushed or popped from the stack (this is really only useful for tracing the problems occurring during stack operations). The *-prnlev* values of 0 or 1 should suffice for most uses.

The option *-out* specifies the format of printed atoms. "short" means a condensed output using residue (:) and atom (@) designators followed by residue ranges and atom names. "pdb" (default) prints atoms in Amber-style PDB format with the original "maskstr" printed as a REMARK at the top of the PDB file, and "amber" prints atom/residue ranges in the format suitable for copying into group input section of Amber input file.

The option *-find* is followed by "maskstr" expression. This is a string where some characters have a special meaning and thus express what parts (atoms/residues) of the molecule will get selected. The syntax of this string is explained in the section above (DESCRIPTION). If this option is left out, it defaults to ":*", which selects all atoms in the given topology file. The length of "maskstr" is limited to 80 characters. If the "maskstr" contains spaces or special characters (which would be expanded by the shell), it should be protected by single or double quotes (depending on the shell). In addition, for C-shells even a quoted exclamation character may be expanded for history substitution. Thus, it is recommended that the operand of the negation operator always be enclosed in parentheses so that "!" is always followed by a "(" to produce "!(" which disables the special history interpretation. For example, [@C= & !(@CA,C)] selects all carbons except backbone alpha and carbonyl carbon; the parentheses are redundant but shell safe. The man page indicates further ways to disable history substitution.

FILES

Assumes that *prmtop* and *inpcrd* files exists in the current directory if they are not specified with *-p* and *-c* options. Resulting (i.e. selected) atoms are written to **stdout**.

BUGS

Because all atom names are left justified in Amber topology and the selections are case insensitive, there is no way to distinguish some atom names: alpha carbon CA and a calcium ion Ca are a notorious example of that.

6.2 GROUP Specification

This section describes the format used to define groups of atoms in various Amber programs. In *sander*, a group can be specified as a movable "belly" while the other atoms are fixed absolutely in space (aside from scaling caused by constant pressure simulation), and/or a group of movable atoms can independently restrained (held by a potential) at their positions. In *anal*, groups can be defined for energy analysis. In sander and pmemd, GROUP input comes at the end of the mdin input file, as discussed in Section 2.6.

Except in the analysis module where different groups of atoms are considered with different group numbers for energy decomposition, in all other places the groups of atoms defined are considered as marked atoms to be included for certain types of calculations. In the case of constrained minimization or dynamics, the atoms to be constrained are read as groups with a different weight for each group.

Reading of groups is performed by the routine RGROUP, and you are advised to consult it if there is still some ambiguity in the documentation.

Input description:

```
- 1 - Title format(20a4)
ITITL Group title for identification.
Setting ITITL = 'END' ends group input.
-----------------------------------------------------------------
- 1A - Weight format(f)
This line is only provided/read when using GROUP input to
define restrained atoms.
WT The harmonic force constants in kcal/mol-A**2 for the group
of atoms for restraining to a reference position.
```

```
------------------------------------------------------------------------
- 1B - Control to define the group
KTYPG , (IGRP(I) , JGRP(I) , I = 1,7) format(a,14i)
KTYPG Type of atom selection performed. A molecule can be
defined by using only 'ATOM' or 'RES', or part of the
molecule can be defined by 'ATOM' and part by 'RES'.
'ATOM' The group is defined in terms of atom numbers. The atom
number list is given in igrp and jgrp.
'RES' The group is defined in terms of residue numbers. The
residue number list is given in igrp and jgrp.
'FIND' This control is used to make additional conditions
(apart from the 'ATOM' and 'RES' controls) which a given
atom must satisfy to be included in the current group.
The conditions are read in the next section (1C) and are
terminated by a SEARCH card.
Note that the conditions defined by FIND filter any set(s) of atoms
defined by the following ATOM/RES instructions. For example,
-- group input: select main chain atoms --
FIND
* * M *
SEARCH
RES 1 999
END
END
'END' End input for the current group. Followed by either another
group definition (starting again with line 1 above), or by a second
'END' "card", which terminates all group input.
IGRP(I) , JGRP(I)
The atom or residue pointers. If ktypg .eq. 'ATOM' all
atoms numbered from igrp(i) to jgrp(i) will be put into
the current group. If ktypg .eq. 'RES' all atoms in the
residues numbered from igrp(i) to jgrp(i) will be put
into the current group. If igrp(i) = 0 the next control
card is read.
It is not necessary to fill groups according to the
numerical order of the residues. In other words, Group 1
could contain residues 40-95 of a protein, Group 2 could
contain residues 1-40 and Group 3 could contain residues
96-105.
If ktypg .eq. 'RES', then associating a minus sign with
igrp(i) will cause all residues igrp(i) through jgrp(i)
to be placed in separate groups.
In the analysis modules, all atoms not explicitly defined
as members of a group will be combined as a unit in the
(n + 1) group, where the (n) group in the last defined
group.
------------------------------------------------------------------------
- 1C - Section to read atom characteristics
***** Read only if KTYPG = 'FIND' *****
JGRAPH(I) , JSYMBL(I) , JTREE(I) , JRESNM(I) format(4a)
A series of filter specifications are read. Each filter consists
of four fields (JGRAPH,JSYMBL,JTREE,JRESNM), and each filter is placed
on a separate line. Filter specification is terminated by a line with
JGRAPH = 'SEARCH'. A maximum of 10 filters may be specified for a
single 'FIND' command.
The union of the filter specifications is applied to the atoms defined
by the following ATOM/RES cards. I.e. if an atom satisfies any of the
```

```
filters, it will be included in the current group. Otherwise, it is not
included. For example, to select all non main chain atoms from residues
1 through 999:
-- group input: select non main chain atoms --
FIND
* * S *
* * B *
* * 3 *
* * E *
SEARCH
RES 1 999
END
END
'END' End input for the current group. Followed by either another
The four fields for each filter line are:
JGRAPH(I) The atom name of atom to be included. If this and the
following three characteristics are satisfied the atom is
included in the group. The wild card '*' may be used to
to indicate that any atom name will satisfy the search.
JSYMBL(I) Amber atom type of atom to be included. The wild card
'*' may be used to indicate that any atom type will
satisfy the search.
JTREE(I) The tree name (M, S, B, 3, E) of the atom to be included.
The wild card '*' may be used to indicate that any tree
name will satisfy the search.
JRESNM(I) The residue name to which the atom has to belong to be
included in the group. The wild card '*' may be used to
indicate that any residue name will satisfy the search.
-----------------------------------------------------------------------
```

Examples:

The molecule 18-crown-6 will be used to illustrate the group options. This molecule is composed of six repeating (-CH2-O-CH2-) units. Let us suppose that one created three residues in the PREP unit: CRA, CRB, CRC. Each of these is a (-CH2-O-CH2-) moiety and they differ by their dihedral angles. In order to construct 18-crown-6, the residues CRA, CRB, CRC, CRB, CRC, CRB are linked together during the LINK module with the ring closure being between CRA(residue 1) and CRB(residue 6).

Input 1:

```
Title one
RES 1 5
END
Title two
RES 6
END
END
```

Output 1: Group 1 will contain residues 1 through 5 (CRA, CRB, CRC, CRB, CRC) and Group 2 will contain residue 6 (CRB).

Input 2:

```
Title one
RES 1 5
END
Title two
ATOM 36 42
END
END
```

Output 2: Group 1 will contain residues 1 through 5 (CRA, CRB, CRC, CRB, CRC) and Group 2 will contain atoms 36 through 42. Coincidentally, atoms 36 through 42 are also all the atoms in residue 6.

Input 3:

```
Title one
RES -1 6
END
END
```

Output 3: Six groups will be created; Group 1: CRA, Group 2: CRB,..., Group 6: CRB.

Input 4:

```
Title one
FIND
O2 OS M CRA
SEARCH
RES 1 6
END
END
```

Output 4: Group 1 will contain those atoms with the atom name ′O2′, atom type ′OS′, tree name ′M′ and residue name ′CRA′.

Input 5:

```
Title one
FIND
O2 OS * *
SEARCH
RES 1 6
END
END
```

Output 5: Group 1 will contain those atoms with the atom name ′O2′, atom type ′OS′, any tree name and any residue name.

Input 6:

```
Title one
RES 1 3 6 6
END
END
```

Output 6: One group is created containing residues 1 to 3 and 6. Up to seven ranges of contiguous residues can be specified per group. (In this case there are two ranges).

Input 7:

```
First restraint group
10.0
FIND
CA * * *
SEARCH
RES 1 17   25 36
END
Second restraint group, with a different restraint weight
1.0
FIND
CA * * *
```

```
SEARCH
RES 61 127
END
END
```

Output 7: CA atoms in residues 1-17 and 25-36 will be restrained to their initial positions with a strong weight of $10.0\text{kcal} \cdot \text{mol}^{-1} \cdot \text{Å}^{-2}$; CA atoms in residues 61 to 127 will have a weaker restraint force constant.

7 Sampling configuration space

The "middle" scheme [Section **??**] offers an efficient approach to accurately sample configuration space in standard molecular dynamics simulations. There are many instances when standard molecular dynamics simulations get "stuck" near the starting configuration, and fail to adequately sample the available low-energy configurational space. This chapter describes a variety of techniques that can partially overcome such problems. The following chapter (on Free Energies) continues many of these ideas, adapting them to the calculation of alchemical or configurational free energy differences. There is no good distinction between these two chapters, because good sampling of the canonical distribution and estimation of free energies go hand-in-hand. But the present chapter covers methods that are *primarily* devoted to enhanced or accelerated sampling, whereas the following chapter considers methods that explicitly estimate free energy differences.

7.1 Self-Guided Langevin dynamics

Self-guided Langevin dynamics (SGLD) is designed to enhance conformational search efficiency in either a molecular dynamics (MD) simulation (when *gamma_ln=0*) or a Langevin dynamics (LD) simulation (when *gamma_ln>0*). This method accelerates low frequency motion to enhance conformational sampling. [129–133]

Overview: The input parameter, *tsgavg*, defines the lower limit period of the low frequency motion. Typically, *tsgavg*=0.2 ps is sutiable to define bond stretching and bending motions as high frequency motion and is recommanded to enhance motions like phase separation, secondary structure folding, and ligand docking, while *tsgavg*=1.0 ps is suitable to include more local motions such as bond rotation and solvent relaxation and is recommended for protein domain motion, and protein-protein docking. The input parameter, *sgft* or *sgff*, defines the strength of the guiding effect. *sgft* between -1 and +1 sets the momentum guiding effect, with 0 for regular LD or MD simulations. A larger *sgft* will enhance low frequency motion to accelerate conformational search. *sgff* between -0.32 and +0.32 defines a force guiding factor to target energy barriers. A negative value will flatten energy barriers. Normally, *sgft* and *sgff* have opposite effects on conformational distribution and there exist balanced values to conserve the canonical ensemble. SGLD accelerates slow events to an affordable time scale while minimizing the perturbation to the conformational distribution. The guiding force can be applied to a part of a simulation system between atom *isgsta* and atom *isgend*.

The conformational distribution of SGLD can be reweighted to produce canonical ensemble averages[130, 133, 134]. The current implementation is mainly based on the most recent reformulation named the generallized self-guided molecular simulation method (SGMDg or SGLDg)[133]. The previous method[134, 135], SGLDfp is no longer available. As an alternative to Langevin dynamics (LD), self-guided Langevin dynamics via generalized Langevin equation (SGLD-GLE) [132] can exactly preserve canonical ensemble while avoiding the slow down by friction forces. SGLD-GLE method can be turned on by setting *sgfg* between -1 and +1. SGLD-GLE can be used with SGLDg by setting *sgft*, *sgff*, and *sgfg*.

`isgld`	SGLD algorithm index. Default *isgld*= 0, SGLD is diabled; *isgld*>0 will turn on SGLDg/SGMDg and/or SGLD-GLE method.
`tsgavg`	Local averaging time (*psec*) for the guiding force calculation. Default 0.2 *psec*. A larger value defines slower motion to be enhanced.
`sgft`	Momentum guiding factor. Defines the strength of the guiding effect. Default 0.0. Suggested value is 1.0 when *sgff=0.* Its value range is -1~1 with larger value leads to stronger

low frequency motion. When *sgft* is set <-1 or >1, it is reset to the balanced value of *sgff* to preserve canonical ensemble.

`sgff`	Force guiding factor. *sgff* is used to scale down low frequency energy surface by a factor, (1+*sgff*). *sgff* is suggested to take values between -0.32 and 0.32, with default value of 0. Suggested value is -0.3 when *sgft=0.* When sgff is set <-1 or >1, it is reset to the balanced value of *sgft* to preserv canonical ensemble.
`sgfg`	momentum guiding factor for SGLD-GLE. Degault is 0. Its value range is -1~1 with larger value resulting in faster low frequency motion. Suggested value is 1 when *sgft=0 and sgff=0.*
`isgsta`	The first atom index of SGLD region. Default is 1.
`isgend`	The last atom index of SGLD region. Default is *natom*.
`nsgsize`	size of the bonded substructure for each atom used to calculate average guiding forces. 1 for no bonded structure is considered; 2 for all atoms connected through bonds or angles ; 3 for all atoms connected through bonds, angles, and dihedral angles. Default is 1.

The output of SGMD/SGLD simulations contains the following properties related to the enhancement in conformational search and reweighting of conformational distribution:
METHOD: GAMM TEMPLF TEMPHF EPOTLF EPOTHF EPOTLLF SGWT

These quantities are instantaneous values defined as below:
METHOD: SGLD when *gamma_ln>0* or SGMD when *gamma_ln=0*
GAMM: Average atomic friction constant based on atom interactions.
TEMPLF: low frequency temperature
TEMPHF: high frequency temperature. Apparent temperature=TEMPLF+TEMPHF
EPOTLF: low frequency potential energy
EPOTHF: high frequency potential energy, EPOT=EPOTLF+EPOTHF
EPOTLLF: Average of low frequency potential energy. It is needed for reweighting.
SGWT: Weighting number. exp(SGWT) is the weighting factor of current frame.

The weight of a conformation is calculated by

```
Weight=exp(SGWT) =exp(((p(sgft)-sgff)*(EPOTLF-EPOTLLF)/(KBOLTZ*Temp))
where: sgft=(1+p(sgft))^2-1/(1+p(sgft))
```

or:

$$w_i = \exp\left(\frac{\lambda_{FP} - \lambda_F}{kT}(E_{LF} - E_{LLF})\right)$$

where p(sgft) or λ_{FP} is called the balanced force guiding factor of λ_P. The relation between sgft, λ_P, and p(sgft), λ_{FP}, is: $\lambda_P = (1+\lambda_{FP})^2 - \frac{1}{1+\lambda_{FP}}$. Similarly, λ_{PF} is called the balanced momentum guiding factor of λ_F, where $\lambda_{PF} = (1+\lambda_F)^2 - \frac{1}{1+\lambda_F}$. When both λ_P and λ_F are used in a SGMDg/SGLDg simulation, it is recommanded that $-1 \leq \lambda_p - \lambda_{PF} \leq 1$.
For convenience, two scripts, *sgldinfo.sh* and *sgldwt.sh*, are provided in the AMBERHOME/bin directory to extract SGLD properties and weighting factors from sander output files. For example, one can run:

```
sgldinfo.sh mdout
```

to examine the SGLD properties, and run:

```
sgldwt.sh mdout
```

to print weighting factors at each print time frame, averages and reweighted averages of the SGMD/SGLD properties. Ensemble average properties can be calculated through reweighting:

$$< P >= \frac{\sum_{i=N_0}^{N} w_i P_i}{\sum_{i=N_0}^{N} w_i}$$

Below are example input files to run SGMD/SGLD simulations.
1. a SGLD simulation with sgft only:

```
Sample SGLD simulation for enhanced conformational search
&cntrl
nstlim=1000, cut=99.0, igb=1, saltcon=0.1,
ntpr=100, ntwr=100000, ntt=3, gamma_ln=10.0,
ntx=5, irest=1,
ntc=2, ntf=2, tol=0.000001,
dt=0.002, ntb=0, tempi=300., temp0=300.,
isgld=1,tsgavg=0.2,sgft=1.0,
/
```

2. a SGLD simulation using bonded substructure(nsgsize=2) to estimate low frequency motion, which reduces noises due to bond stretching and bending.

```
Sample SGLD simulation using bonded substructures
&cntrl
nstlim=1000, cut=99.0, igb=1, saltcon=0.1,
ntpr=100, ntwr=100000, ntt=3, gamma_ln=10.0,
ntx=5, irest=1,
ntc=2, ntf=2, tol=0.000001,
dt=0.002, ntb=0, tempi=300., temp0=300.,
isgld=1,tsgavg=0.2,sgft=1.0, nsgsize=2,
/
```

3. a SGLDg simulation using both *sgft* and *sgff* factors. When both guiding factors are used, it is recommand that $-1 \leq \lambda_p - (1+\lambda_F)^2 + \frac{1}{1+\lambda_F} \leq 1$.

```
Sample SGLDg simulation using both sgft and sgff
&cntrl
nstlim=1000, cut=99.0, igb=1, saltcon=0.1,
ntpr=100, ntwr=100000, ntt=3, gamma_ln=10.0,
ntx=5, irest=1,
ntc=2, ntf=2, tol=0.000001,
dt=0.002, ntb=0, tempi=300., temp0=300.,
isgld=1,tsgavg=0.2,sgft=0.5,sgff=-0.1,
/
```

4. a SGMDg simulation when gamma_ln=0:

```
Sample SGMDg simulation using both sgft and sgff
&cntrl
nstlim=1000, cut=99.0, igb=1, saltcon=0.1,
ntpr=100, ntwr=100000, ntt=1, tautp=1.0,
ntx=5, irest=1,
ntc=2, ntf=2, tol=0.000001,
dt=0.002, ntb=0, tempi=300., temp0=300.,
isgld=1,tsgavg=0.2,sgft=0.5,sgff=-0.1,
/
```

5. a SGLD-GLE simulation using sgfg to achieve enhanced conformational search while conserve canonical ensemble exactly:

```
Sample SGLD-GLE simulation to conserve canonical ensemble
&cntrl
nstlim=1000, cut=99.0, igb=1, saltcon=0.1,
ntpr=100, ntwr=100000, ntt=3, gamma_ln=10.0,
ntx=5, irest=1,
ntc=2, ntf=2, tol=0.000001,
dt=0.002, ntb=0, tempi=300., temp0=300.,
isgld=1,tsgavg=0.2,sgfg=1.0,
/
```

7.2 Targeted MD

The targeted MD option adds an additional term to the energy function based on the mass-weighted root mean square deviation of a set of atoms in the current structure compared to a reference structure. The reference structure is specified using the *-ref* flag in the same manner as is used for Cartesian coordinate restraints (NTR=1). Targeted MD can be used with or without positional restraints. If positional restraints are not applied (ntr=0), *sander* performs a best-fit of the reference structure to the simulation structure based on selection in *tgtfitmask* and calculates the RMSD for the atoms selected by *tgtrmsmask*. The two masks can be identical or different. This way, fitting to one part of the structure but calculating the RMSD (and thus restraint force) for another part of the structure is possible. If targeted MD is used in conjunction with positional restraints (ntr=1), only *tgtrmsmask* should be given in the control input because the molecule is 'fitted' implicitly by applying positional restraints to atoms specified in *restraintmask*.
The energy term has the form:

```
E = 0.5 * TGTMDFRC * NATTGTRMS * (RMSD-TGTRMSD)**2
```

The energy will be added to the RESTRAINT term. Note that the energy is weighted by the number of atoms that were specified in the *tgtrmsmask* (NATTGTRMS). The RMSD is the root mean square deviation and is mass weighted. The force constant is defined using the *tgtmdfrc* variable (see below). This option can be used with molecular dynamics or minimization. When targeted MD is used, *sander* will print the current values for the actual and target RMSD to the energy summary in the output file.

`itgtmd`	**= 0** no targeted MD (default) **= 1** use targeted MD **= 2** use targeted MD to multiple targets (Multiply-targeted MD, or MTMD, see next section below)
`tgtrmsd`	Value of the target RMSD. The default value is 0. This value can be changed during the simulation by using the weight change option.
`tgtmdfrc`	This is the force constant for targeted MD. The default value is 0, which will result in no penalty for structure deviations regardless of the RMSD value. Note that this value can be negative, which would force the coordinates AWAY from the reference structure.
`tgtfitmask`	Define the atoms that will be used for the rms superposition between the current structure and the reference structure. Syntax is in Chapter 6.1.1.
`tgtrmsmask`	Define the atoms that will be used for the rms difference calculation (and hence the restraint force), as outlined above. Syntax is in Chapter 6.1.1.

One can imagine many uses for this option, but a few things should be kept in mind. In this implementation of targeted MD, there is currently only one reference coordinate set, so there is no way to force the coordinates to any specific structure other than the one reference. To move a structure toward a reference coordinate set, one might use an initial *tgtrmsd* value corresponding to the actual RMSD between the input and reference (*inpcrd* and *refc*). Then the weight change option could be used to decrease this value to 0 during the simulation. To move a structure away from the reference, one can increase *tgtrmsd* to values larger than zero. The minimum for this energy term will then be at structures with an RMSD value that matches *tgtrmsd*. Keep in mind that many different structures may have similar RMSD values to the reference, and therefore one cannot be sure that increasing *tgtrmsd* to a given value will result in a particular structure that has that RMSD value. In this case it is probably wiser to use the final structure, rather than the initial structure, as the reference coordinate set, and decrease *tgtrmsd* during the simulation. To address this, multiply-targeted MD is now available in Amber (*sander only*), and is described in the next section. As an additional note, a negative force constant *tgtmdfrc* can be used, but this can cause problems since the energy will continue to decrease as the RMSD to the reference increases.

Also keep in mind that phase space for molecular systems can be quite complex, and this method does not guarantee that a low energy path between initial and target structures will be followed. It is possible for the simulation to become unstable if the restraint energies become too large if a low-energy path between a simulated structure and the reference is not accessible.

Note also that the input and reference coordinates are expected to match the *prmtop* file and have atoms in the same sequence. No provision is made for symmetry; rotation of a methyl group by 120° would result in a nonzero RMSD value.

7.3 Multiply-Targeted MD (MTMD)

In Amber (*sander only*), the user may perform targeted MD calculations using multiple reference structures. Each reference may have its own associated target RMSD value and force constant, each of which can evolve independently in time. Additionally, the masks for each defined target may differ, and targeting to any given reference structure can be activated for some or part of the simulation. The energy term for MTMD is simply the sum of the energies that would be calculated for the molecule calculated relative to each target given the target RMSD and force constant for that target. The energy will then be added to the RESTRAINT term.

To use MTMD, the MTMD input file is specified using the *-mtmd* flag in the command line arguments for *sander*. The MTMD input file will contain one instance of the tgt namelist ("&tgt") for each reference structure used. The user may specify any number of reference structures.

7.3.1 Variables in the &tgt namelist:

`refin` The file name of the reference structure used. The input and reference coordinates are expected to match the *prmtop* file and have atoms in the same sequence. *Default for refin is ", no reference structure given.*

`mtmdform` If MTMDFORM > 0, then the reference coordinate file is formatted. Otherwise, the reference coordinate file is an unformatted (binary) file. *Default for MTMDFORM is the value assigned to MTMDFORM in the most recent namelist where MTMDFORM was specified. If MTMDFORM has not been specified in any namelist, it defaults to 1.*

`mtmdstep1, mtmdstep2` Targeted MD for this structure is run for steps/iterations MTMDSTEP1 through MTMDSTEP2. If MTMDSTEP2 = 0, then TMD will be run through the end of the run, and the values of the target RMSD and the force constant will not change with time. Note that the first step/iteration is considered step 0. *Defaults for MTMDSTEP1 and MTMDSTEP2 are the values assigned to them in the most recent namelist where MTMDSTEP1 and MTMDSTEP2 were specified. If MTMDSTEP1 and MTMDSTEP2 have not been specified in any namelist, they default to 0.*

`mtmdvari` If MTMDVARI > 0, then the force constant and target RMSD will vary with step number. Otherwise, they are constant throughout the run. If MTMDVARI >0, then the values MTMDSTEP2, MTMDRMSD2, and MTMDFORCE2 must be specified (see below). *Default for* MTMDVARI *is the value assigned to* MTMDVARI *in the most recent namelist where* MTMDVARI *was specified. If* MTMDVARI *has not been specified in any namelist, it defaults to 0.*

`mtmdrmsd, mtmdrmsd2` The target RMSD for this reference. If MTMDVARI >0, then the value of MTMDRMSD will vary between MTMDSTEP1 and MTMDSTEP2, so that, e.g. MTMDRMSD(MTMDSTEP1) = MTMDRMSD and MTMDRMSD(MTMDSTEP2) = MTMDRMSD2. *Defaults for MTMDRMSD and MTMDRMSD2 are the values assigned to them in the most recent namelist where MTMDRMSD and MTMDRMSD2 were specified. If MTMDRMSD and MTMDRMSD2 have not been specified in any namelist, they default to 0.0.*

`mtmdforce, mtmdforce2` The force constant for this reference. If MTMDVARI >0, then the value of MTMDFORCE will vary between MTMDSTEP1 and MTMDSTEP2, so that, e.g. MTMDFORCE(MTMDSTEP1) = MTMDFORCE and MTMDFORCE(MTMDSTEP2) = MTMDFORCE2. *Defaults for* MTMDFORCE *and* MTMDFORCE2 *are the values assigned to them in the most recent namelist where* MTMDFORCE *and* MTMDFORCE2 *were specified. If* MTMDFORCE *and* MTMDFORCE2 *have not been specified in any namelist, they default to 0.0.*

`mtmdninc` If MTMDVARI >0 and MTMDNINC > 0, then the changes in the values of of MTMDRMSD and MTMDFORCE are applied as a step function, with NINC steps/iterations between each change in the target values. If MTMDNINC = 0, the change is effected continuously (at every step). *Default for MTMDNINC is the value assigned to MTMDNINC in the most recent namelist where MTMDNINC was specified. If MTMDNINC has not been specified in any namelist, it defaults to 0.*

`mtmdmult` If MTMDMULT=0, and the values of MTMDFORCE changes with step number, then the changes in the force constant will be linearly interpolated from MTMDFORCE$\rightarrow$MTMDFORCE2 as the step number changes. If MTMDMULT=1 and the force constant is changing with step number, then the changes in the force constant will be effected by a series of multiplicative scalings, using a single factor, R, for all scalings. *i.e.*

```
MTMDFORCE2 = R**INCREMENTS * MTMDFORCE
```

INCREMENTS is the number of times the target value changes, which is determined by MTMDSTEP1, MTMDSTEP2, and MTMDNINC. *Default for MTMDMULT is the value assigned to MTMDMULT in the most recent namelist where MTMDMULT was specified. If MTMDMULT has not been specified in any namelist, it defaults to 0.*

`mtmdmask` Define the atoms that will be used for both the rms superposition between the current structure and the reference structure and the rms difference calculation (and hence the restraint force), as outlined above. Syntax is in Chapter 6.1.1.*Default for MTMDMASK is the value assigned to MTMDMASK in the most recent namelist where MTMDMASK was specified. If MTMDMASK has not been specified in any namelist, it defaults to '*', use all atoms in the fit and force calculations.* \

Namelist &tgt is read for each reference structure. Input ends when a namelist statement with refin = '' (or refin not specified) is found. Note that comments can precede or follow any namelist statement, allowing comments and reference definitions to be freely mixed.

7.4 Low-MODe (LMOD) methods

István Kolossváry's LMOD methods for minimization, conformational searching, and flexible docking[136–139] are fully implemented in Amber. 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.

In the Amber 2020 release, the LMOD optimization code has been updated with major improvements and new features including more accurate flexible docking, the option to visualize normal modes, utilization of random mixtures of low-frequency modes, and the option to work with a range of modes anywhere in the spectrum and not just the lowest frequency modes. The latter is particularly useful for docking where the modes relevant to binding a ligand molecule are usually not the lowest frequency modes. The interface of the new LMOD has not changed, everything works exactly the same way as in Amber18 and earlier versions, a few parameters simply have additional options as documented below. The new features are demonstrated with production quality examples.

Details of the LMOD procedure, and hints on getting good performance, are given Section 14.5, which should be consulted before trying the procedures in *sander*. The only difference between the *sander* and *NAB* implementations is the input specification; the same LMOD code is linked into both. The sections below give input details for *sander*.

There are **four "real-life" examples** of performing LMOD searches and in *lmod_vib_anim* **three examples** of updates in Amber20 including generating LMOD-vibration visualization. These are available at *https://ambermd.org/Manuals.php*. Each directory in the tar file has a README file with more information.

7.4.1 XMIN

The XMIN methods for minimization are traditional and manifold in the field of unconstrained optimization: PRCG is a Polak-Ribiere nonlinear Conjugate Gradient algorithm,[140] LBFGS is a Limited-memory Broyden-Fletcher-Goldfarb-Shanno quasi-Newton algorithm,[141] and TNCG is a Truncated Newton linear Conjugate Gradient method with optional LBFGS preconditioning.[142]

Some of the &cntrl namelist variables that control Amber's other minimization facilities also control XMIN. Consequently, non-experts can employ the default XMIN method merely by specifying ntmin = 3.

`maxcyc` The maximum number of cycles of minimization. Default is 1 to be consistent with Amber's other minimization facilities although it may be unrealistically short.

`ntmin` The flag for the method of minimization.

- **= 3** The XMIN method is used.
- **= 4** The LMOD method is used. The LMOD procedure employs XMIN for energy relaxation and minimization.

`drms` The convergence criterion for the energy gradient: minimization will halt when the root-mean-square of the Cartesian elements of the gradient is less than this. Default is $10^{-4}\text{kcal} \cdot \text{mol}^{-1} \cdot \text{Å}^{-1}$. This is consistent with Amber's other minimization facilities. In Amber18 and earlier this default may have been unrealistically strict. In Amber20 this criterion refers to the minimization of the input structure for which the normal modes are computed, and to avoid unnatural vibrational modes it should be set to even stricter values, e.g., 10^{-8}. Compare with input parameter lmod_minimize_grms below.

Other options that control XMIN are in the scope of the &lmod namelist. These parameters enable expert control of XMIN.

`lbfgs_memory_depth` The depth of the LBFGS memory for LBFGS minimization, or LBFGS preconditioning in TNCG minimization. Default is 3. Suggested alternate value is 5. The value 0 turns off LBFGS preconditioning in TNCG minimization.

`matrix_vector_product_method` The finite difference Hv matrix-vector product method: "forward" = forward difference, "central" = central difference. Default is forward difference.

`xmin_method` The minimization method: "PRCG" = Polak-Ribiere Conjugate Gradient, "LBFGS" = Limited-memory Broyden-Fletcher-Goldfarb-Shanno, and "TNCG" = Optionally LBFGS-preconditioned Truncated Newton Conjugate Gradient. Default is LBFGS.

`xmin_verbosity` The verbosity of the internal status output from the XMIN package: 0 = none, 1 = minimization details, and 2 = minimization and line search details plus CG details in TNCG. Currently, the XMIN status output may be disordered with respect to Amber's output. Default is 0, no output of the XMIN package internal status. Note that XMIN is also available in AmberTools, in the NAB package. An annotated example output corresponding to XMIN_VERBOSITY=2 can be found in the NAB documentation.

7.4.2 LMOD

Some of the options that control LMOD have the same names as Amber's other minimization facilities. See the XMIN section immediately above. Other options that control LMOD are in the scope of the &lmod namelist. These parameters enable expert control of LMOD.

`arnoldi_dimension` The dimension of the ARPACK Arnoldi factorization. Zero specifies the whole space, that is, three times the number of atoms. Default is 0, the whole space. 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 total_low_modes + 1 <= arnoldi_dimension <= three times the number of atoms. 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 arnoldi_dimension 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 an arnoldi_dimension such that all the ARPACK storage fits in memory. For proteins, arnoldi_dimension=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.

`conflib_filename` The user-given filename of the LMOD conformational library. The file format is Amber standard formatted trajectory output regardless of the value of &cntrl namelist variable ioutfm. The conformations are stored in energetic order (global minimum energy structure first), the number of conformations<= conflib_size. The default filename is *conflib*.

`conflib_size` The number of conformations to store in conflib. Default is 3.

`energy_window` The energy window for conformation storage; the energy of a stored structure will be in the interval [global_min, global_min + energy_window]. Default is 0, only storage of the global minimum structure.

`explored_low_modes` The number of low frequency vibrational modes used per LMOD iteration. Default is 3.

`frequency_eigenvector_recalc` The frequency, measured in LMOD iterations, of the recalculation of eigenvectors. Default is 3.

`frequency_ligand_rotrans` The frequency, measured in LMOD iterations, of the application of rigid-body rotational and translational motions to the ligand(s). At each frequency_ligand_rotrans-th LMOD iteration number_ligand_rotrans rotations and translations are applied to the ligand(s). Default is 1, ligand(s) are rotated and translated at every LMOD iteration.

`lmod_job_title` The user-given title for the job that goes in the first line of the conflib and lmod_trajectory files. The default job title is "job_title_goes_here".

`lmod_minimize_grms` In Amber18 and earlier the gradient root-mean-square convergence criterion of structure minimization. In Amber 20 this was specified to be the criterion to minimize low-energy conformations; such conformations do not require as strict a convergence criterion as does the first minimization whose convergence is now controlled with input parameter drms, see above. Default is 0.1.

`lmod_relax_grms` The gradient RMS convergence criterion of structure relaxation. Default is 1.0.

`lmod_restart_frequency` The frequency, in LMOD iterations, of conflib updating and LMOD restarting with a randomly chosen structure from the pool. Default is 5.

`lmod_step_size_max` The maximum length of a single LMOD ZIG move. Default is 5.0 Å.

`lmod_step_size_min` The minimum length of a single LMOD ZIG move. Default is 2.0 Å.

`lmod_trajectory_filename` The filename of the LMOD pseudo trajectory. The file format is standard Amber trajectory file. The conformations in this file show the progress of the LMOD search. The number of conformations = number_lmod_iterations + 1. The default filename is *lmod_trajectory*.

`lmod_verbosity` The verbosity of the internal status output from the LMOD package: 0 = none, 1 = some details, 2 = more details, 3 = everything including ARPACK information, 4 = ARPACK only, 5 = visualize normal modes. Currently, the LMOD status output may be disordered with respect to Amber's output. Default is 0, no output of the LMOD package internal status. Note that LMOD is also available in AmberTools, in the NAB package. An annotated example output corresponding to LMOD_VERBOSITY=2 can be found in the NAB documentation.

`monte_carlo_method` The Monte Carlo method: "Metropolis" = Metropolis Monte Carlo, "Total_Quench" = the LMOD trajectory always proceeds towards the lowest lying neighbor of a particular energy well found after exhaustive search along all of the low modes, and "Quick_Quench" = 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. Default is Metropolis Monte Carlo.

`number_free_rotrans_modes` In Amber18 and earlier this was solely the number of rotational and translational degrees of freedom (dof) which 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. In Amber20 the input domain was extended to any non-negative integer, and it represents the number of modes for LMOD to skip. In this way LMOD can now explore a range of modes instead of simply modes starting with the lowest frequency. Note that it is recommended to set this to 0 once in order to examine the ro-translational modes. Default is 6.

`number_ligand_rotrans` The number of rigid-body rotational and translational motions applied to the ligand(s). Such applications occur at each frequency_ligand_rotrans-th LMOD iteration. Default is 0, no rigid-body motions applied to the ligand(s).

`number_ligands` The number of ligands for flexible docking. Default is 0, no ligand(s).

`number_lmod_iterations` The number of LMOD iterations. Default is 10. Note that setting number_lmod_iterations = 0 will result in a single energy minimization.

`number_lmod_moves` The number of LMOD ZIG-ZAG moves. 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 the "LMOD Procedure" background section. number_lmod_moves > 0 means that multiple barriers may be crossed and LMOD can carry the molecule to a large distance on the potential energy surface without severely distorting the geometry. Default is 0, LMOD will determine automatically where to stop the ZIG-ZAG sequence.

`random_seed` The seed of the random number generator. Default is 314159.

`restart_pool_size` The size of the pool of lowest-energy structures to be used for restarting. Default is 3.

`rtemperature` The value of RT in Amber energy units. This is utilized in the Metropolis criterion. Default is 1.5.

`total_low_modes` The total number of low frequency vibrational modes to be used. Default is the minimum of 10 and three times the number of atoms minus the number of rotational and translational degrees of freedom (number_free_rotrans_modes).

The following commands are part of the &lmod namelist. These commands control the way LMOD applies explicit translations and rotations to one or more ligands and take effect only if number_ligands >= 1. All commands are lists in square brackets, separated by commas such as [1, 33, 198], however, the list is read by Sander as a string and, therefore, it should be enclosed in single quotes.

`ligstart_list, ligend_list` The serial number(s) of the first/last atom(s) of the ligand(s). Type integer. The number(s) should correspond to the numbering in the Amber input files prmtop and inpcrd/restart. For example, if there is only one ligand and it starts at atom 193, the command should be ligstart_list = '[193]'. If there are three ligands, the command should be, e.g., '[193, 244, 1435]'. The same format holds for all of the following commands. Note that the ligand(s) can be anywhere in the atom list, however, a single ligand must have continuous numbering between the corresponding ligstart_list and ligend_list values. For example, ligstar_list = '[193, 244, 1435]' and ligend_list = '[217, 302, 1473]'.

`ligcent_list` The serial number(s) of the atom(s) of the ligand(s), which serves as the center of rotation. Type integer. The value zero means that the center of rotation will be the geometric center of gravity of the ligand.

`rotmin_list, rotmax_list` The range of random rotation of a particular ligand about the origin defined by the corresponding ligcent_list value is specified by the commands rotmin_list and rotmax_list. The angle is given in +/- degrees. Type float. For example, in case of a single ligand and ligcent_list = '[0]', rotmin_list = '[30.0]' and rotmax_list = '[180.0]' means that random rotations by an angle +/- 30-180 degrees about the center of gravity of the ligand, will be applied. Similarly, with number_ligands = 2, ligcent_list= 120.0]' means that the first ligand will be rotated like in the single ligand example in this paragraph, but a second ligand will be rotated about its atom number 201, by an angle +/- 60-120 degrees.

`trmin_list, trmax_list` The range of random translation(s) of ligand(s) is defined by the same way as rotation. For example, with number_ligand = 1, trmin_list = '[0.1]' and trmax_list = '[1.0]' means that a single ligand is translated in a random direction by a random distance between 0.1 and 1.0 Angstroms.

8 Free energies

8.1 Thermodynamic integration

In a free energy calculation, the system evolves according to a mixed potential (such as in Eqs. 8.3 or 8.4, below). The essence of free energy calculations is to record and analyze the fluctuations in the values of V_0 and V_1 (that is, what the energies *would have been* with the endpoint potentials) as the simulation progresses. For thermodynamic integration (which is a very straightforward form of analysis) the required averages can be computed "on-the-fly" (as the simulation progresses), and printed at the end of a run. For more complex analyses (such as the Bennett acceptance ratio scheme), one needs to write the history of the values of V_0 and V_1 to a file, and later post-process this file to obtain the final free energy estimates.

There is not room here to discuss the theory of free energy simulations, and there are many excellent discussions elsewhere.[143–145] There are also plenty of recent examples to consult.[146, 147] Such calculations are demanding, both in terms of computer time, and in a level of sophistication to avoid pitfalls that can lead to poor convergence. Since there is no one "best way" to estimate free energies, *sander* and *pmemd* primarily provide the tools to collect the statistics that are needed. Assembling these into a final answer, and assessing the accuracy and significance of the results, generally requires some calculations outside of what Amber provides, *per se*. The discussion here will assume a certain level of familiarity with the basis of free energy calculations.

Both *sander* and *pmemd* have the capability of doing simple thermodynamic free energy calculations, using either PME or generalized Born potentials. When *icfe* is set to 1, information useful for doing thermodynamic integration estimates of free energy changes will be computed. The implementation is different between *sander* and *pmemd*. For *sander*, you must use the *multisander* capability to create two groups, one corresponding to the starting state, and a second corresponding to the ending state (see Section 2.12 for information); you will need a *prmtop* file for each of these two endpoints. For *pmemd*, you use a single *prmtop* file which contains both the starting and ending states. For both *sander* and *pmemd* a mixing parameter λ is used to interpolate between the "unperturbed" and "perturbed" potential functions.

8.1.1 Thermodynamic integration using Sander

There are now two different ways to prepare a thermodynamic integration free energy calculation in Sander. The first is unchanged from previous versions of Amber: Here, the two prmtop files that you create must have the same number of atoms, and the atoms must appear *in the same order* in the two files. This is because there is only one set of coordinates that are propagated in the molecular dynamics algorithm. If there are more atoms in the initial state than in the final, "dummy" atoms must be introduced into the final state to make up the difference. Although there is quite a bit of flexibility in choosing the initial and final states, it is important in general that the system be able to morph "smoothly" from the initial to the final state. Alternatively, you can set up your system to use the softcore potential algorithm described below. This will remove the requirement to prepare "dummy" atoms and allows the two *prmtop* files to have different numbers of atoms.

The basics of the *multisander* functionality are given in Section 2.12, but the mechanics are really quite simple. You start a free energy calculation as follows:

```
mpirun -np 4 sander.MPI -ng 2 -groupfile <filename>
```

Since there are 4 total cpu's in this example, each of the two groups will run in parallel with 2 cpu's each. The number of processors must be a multiple of two. The *groups* file might look like this:

```
-O -i mdin -p prmtop.0 -c eq1.x -o md1.o -r md1.x -inf mdinfo
-O -i mdin -p prmtop.1 -c eq1.x -o md1b.o -r md1b.x -inf mdinfob
```

The input (*mdin*) and starting coordinate files must be the same for the two groups. Furthermore, the two *prmtop* files must have the same number number of atoms, in the same order (since one common set of coordinates will be used for both.) The simulation will use the masses found in the first *prmtop* file; in classical statistical mechanics, the Boltzmann distribution in coordinates is independent of the masses so this should not represent any real restriction.

On output, the two restart files should be identical, and the two output files should differ only in trivial ways such as timings; there should be no differences in any energy-related quantities, except if energy decomposition is turned on (*idecomp* > 0); then only the output file of the first group contains the per residue contributions to $\langle \partial V/\partial\lambda \rangle$. For our example, this means that one could delete the *md1b.o* and *md1b.x* files, since the information they contain is also in *md1.o* and *md1.x*. (It is a good practice, however, to check these file identities, to make sure that nothing has gone wrong.)

8.1.2 Basic inputs for thermodynamic integration

icfe The basic flag for free energy calculations. The default value of 0 skips such calculations. Setting this flag to 1 turns them on, using the mixing rules in Eq. 8.3, below.

clambda The value of λ for this run, as in Eqs. 8.3 and 8.4, below. Zero corresponds to the unperturbed Hamiltonian V_0. λ=1 corresponds to the perturbed Hamiltonian V_1.

`klambda` The exponent in Eq. 8.4, below.

`tishake` Flag that determines how SHAKE is handled:

= 0 Coordinates are synchronized after SHAKE, no constraints removed (default).

= 1 SHAKE is removed between bonds containing one common and one unique atom. This was the default in previous versions of *sander*. Note that disabling SHAKE requires the use of a 1 fs timestep.

8.1.2.1 Input flags specific to Sander

`idecomp` Flag that turns on/off decomposition of $\langle \partial V/\partial\lambda \rangle$ on a per-residue level. The default value of 0 turns off energy decomposition. A value of 1 turns the decomposition on, and 1-4 nonbonded energies are added to internal energies (bond, angle, torsional). A value of 2 turns the decomposition on, and 1-4 nonbonded energies are added to EEL and VDW energies, respectively. The frequency by which values of $\langle \partial V/\partial\lambda \rangle$ are included into the decomposition is determined by the NTPR flag. This ensures that the sum of all contributions equals the average of all total $\langle \partial V/\partial\lambda \rangle$ values output every NTPR steps. All residues, including solvent molecules, have to be chosen by the RRES card to be considered for decomposition. The RES card determines which residue information is finally output. The output comes at the end of the *mdout* file. For each residue contributions of internal -, VdW-, and electrostatic energies to $\langle \partial V/\partial\lambda \rangle$ are given as an average over all (NSTLIM/NTPR) steps. In a first section total per residue values are output followed below by further decomposed values from backbone and sidechain atoms.

8.1.3 Background theory of thermodynamic integration

The *sander and pmemd* programs do not compute free energies; it is up to the user to combine the output of several runs (at different values of λ) and to numerically estimate the integral:

$$\Delta A = A(\lambda=1) - A(\lambda=0) = \int_0^1 \langle \partial V/\partial\lambda \rangle_\lambda \, d\lambda \tag{8.1}$$

If you understand how free energies work, this should not be at all difficult. However, since the actual values of λthat are needed, and the exact method of numerical integration, depend upon the problem and upon the precision desired, we have not tried to pre-code these into the program.

The simplest numerical integration is to evaluate the integrand at the midpoint:

$$\Delta A \simeq \langle \partial V / \partial \lambda \rangle_{1/2}$$

This might be a good first thing to do to get some picture of what is going on, but is only expected to be accurate for very smooth or small changes, such as changing just the charges on some atoms. Gaussian quadrature formulas of higher order are generally more useful:

$$\Delta A = \sum_i w_i \langle \partial V / \partial \lambda \rangle_i \tag{8.2}$$

Some weights and quadrature points are given in the accompanying table; other formulas are possible,[148] but the Gaussian ones listed there are probably the most useful. The formulas are always symmetrical about λ= 0.5, so that λ and $(1 - \lambda)$ both have the same weight. For example, if you wanted to use 5-point quadrature, you would need to run five jobs, setting $\lambda$to 0.04691, 0.23076, 0.5, 0.76923, and 0.95308 in turn. (Each value of λ should have an equilibration period as well as a sampling period; this can be achieved by setting the *ntave* parameter.) You would then multiply the values of $\langle \partial V / \partial \lambda \rangle_i$ by the weights listed in the Table, and compute the sum.

When *icfe=1* and *klambda* has its default value of 1, the simulation uses the mixed potential function:

$$V(\lambda) = (1 - \lambda)V_0 + \lambda V_1 \tag{8.3}$$

where V_0 is the potential with the original Hamiltonian, and V_1 is the potential with the perturbed Hamiltonian. The program also computes and prints $\langle \partial V / \partial \lambda \rangle$ and its averages; note that in this case, $\langle \partial V / \partial \lambda \rangle = V_1 - V_0$. This is referred to as linear mixing, and is often what you want unless you are making atoms appear or disappear. If some of the perturbed atoms are "dummy" atoms (with no van der Waals terms, so that you are making these atoms "disappear" in the perturbed state), the integrand in Eq. 8.1 diverges at λ= 1; this is a mild enough divergence that the overall integral remains finite, but it still requires special numerical integration techniques to obtain a good estimate of the integral.[144] *Sander and pmemd* implement one simple way of handling this problem: if you set *klambda* > 1, the mixing rules are

$$V(\lambda) = (1 - \lambda)^k V_0 + [1 - (1 - \lambda)^k] V_1 \tag{8.4}$$

where k is given by *klambda*. Note that this reduces to Eq. 8.3 when k = 1, which is the default. If $k \geq 4$, the integrand remains finite as $\lambda \rightarrow 1$.[144] We have found that setting k= 6 with disappearing groups as large as tryptophan works, but using the softcore option (*ifsc*>0) instead is generally preferred.[149] Note that the behavior of $\langle \partial V / \partial \lambda \rangle$ as a function of λ is not monotonic when *klambda* > 1. You may need a fairly fine quadrature to get converged results for the integral, and you may want to sample more carefully in regions where $\langle \partial V / \partial \lambda \rangle$ is changing rapidly.

Notes:

1. This is implemented in *sander* by calling the force() routine independently for each *multisander* group and then combining the forces on each step. For a fixed number of processors this increases the cost of the calculation compared with the *pmemd* code, which only calculates the differences between V_0 and V_1.

2. It is rather easy to make mistakes when running TI calculations. It is generally good to carry out a short run (say 50 steps) setting ntpr=1. Then check the following; if either test fails, be sure to fix the problem before proceeding.

   a) The restart files from V_0 and V_1 should be identical for *sander* (for *pmemd* there will only be a single restart file).

b) If you diff the output files for *sander*, there should only be simple differences (for *pmemd* there will only be a single combined output file). All energies, temperatures, pressures, etc. should be the same in the two files. Simulations with *sander* using the QM/MM facility may show differences in the SCF energies, but be sure that the total energies, and all the MM components, are the same.

3. Eq. 8.4 is designed for having dummy atoms in the perturbed Hamiltonian, and "real" atoms in the regular Hamiltonian. You must ensure that this is the case when you set up the system in LEaP. (See the softcore section, below, for a more general way to handle disappearing atoms, which does not require dummy atoms at all.)

4. One common application of this model is to pKa calculations, where the charges are mutated from the protonated to the deprotonated form. Since H atoms bonded to oxygen already have zero van der Waals radii (in the Amber force fields and in TIP3P water), once their charge is removed (in the deprotonated form) they are really then like dummy atoms. For this special situation, there is no need to use *klambda* > 1: since the van der Waals terms are missing from both the perturbed and unperturbed states, the proton's position can never lead to the large contributions to $\langle \partial V / \partial \lambda \rangle$ that can occur when one is changing from a zero van der Waals term to a finite one.

5. The implementation requires that the masses of all atoms be the same on all threads. To enforce this, the masses found for V_0 are used for V_1 as well. In classical statistical mechanics, the canonical distribution of configurations (and hence of potential energies) is unaffected by changes in the masses, so this should not pose a limitation. Since the masses for V_1 are ignored, they do not have to match those found for V_0.

6. Special care needs to be taken when using SHAKE for atoms whose force field parameters differ in the two end points. The same bonds must be SHAKEN in both cases, and the equilibrium bond lengths must also be the same. By default, the coordinates from V_0 are synchronized with those from V_1 after SHAKE. This will work for small perturbations, but if there is a significant change in bond length, it may be necessary to use the *noshakemask* input to remove SHAKE from the regions that are being perturbed. If this is done, be sure to set *tishake*=1 and to use a 1 fs timestep. Special care needs to be taken when water molecules are part of the region that is changing. You need to make sure that the “number of 3-point waters” is the same in both V_0 and V_1. This may require setting *jfastw* and/or building the structure so that *sander or pmemd* do not think that the water molecules involved are actually rigid waters. Also, just setting *noshakemask* might not be enough, since this flag does not affect the *settle* routine that handles rigid waters.

n	λ_i	$1-\lambda_i$	w_i
1	0.5		1.0
2	0.21132	0.78867	0.5
3	0.1127	0.88729	0.27777
	0.5		0.44444
5	0.04691	0.95308	0.11846
	0.23076	0.76923	0.23931
	0.5		0.28444
7	0.02544	0.97455	0.06474
	0.12923	0.87076	0.13985
	0.29707	0.70292	0.19091
	0.5		0.20897
9	0.01592	0.98408	0.04064
	0.08198	0.91802	0.09032
	0.19331	0.80669	0.13031
	0.33787	0.66213	0.15617
	0.5		0.16512
12	0.00922	0.99078	0.02359
	0.04794	0.95206	0.05347
	0.11505	0.88495	0.08004
	0.20634	0.79366	0.10158
	0.31608	0.68392	0.11675
	0.43738	0.56262	0.12457

Table 8.1: *Abscissas and weights for Gaussian integration.*

8.1.4 Softcore Potentials in Thermodynamic Integration

Softcore potentials provide an additional way to perform thermodynamic integration calculations in Amber. The system setup has been simplified so that appearing and disappearing atoms can be present at the same time and no dummy atoms need to be introduced. For *sander*, two prmtop files, corresponding to the start and end states (V_0 and V_1) of the desired transformation need to be used. The common atoms that are present in both states need to appear in the same order in both prmtop files and must have identical starting positions. In addition to the common atoms, each process can have any number of unique soft core atoms, as specified by scmask. For *pmemd*, a single prmtop file is used, containing the unique atoms for both the start and end states. The soft core atoms are specified by scmask1 and scmask2 for V_0 and V_1 respectively.

A modified version of the vdW equation is used to smoothly switch off non-bonded interactions of these atoms with their common atom neighbors:

$$V_{V_0,disappearing} = 4\epsilon(1-\lambda)\left[\frac{1}{\left[\alpha\lambda + \left(\frac{r_{ij}}{\sigma}\right)^6\right]^2} - \frac{1}{\alpha\lambda + (\frac{r_{ij}}{\sigma})^6}\right] \tag{8.5}$$

$$V_{V_1,appearing} = 4\epsilon\lambda\left[\frac{1}{\left[\alpha(1-\lambda) + \left(\frac{r_{ij}}{\sigma}\right)^6\right]^2} - \frac{1}{\alpha(1-\lambda) + (\frac{r_{ij}}{\sigma})^6}\right] \tag{8.6}$$

Please refer to Ref [149] for a description of the implementation and performance testing when compared to the TI methods described above using *sander*. For similar information pertaining to pmemd please see Ref [150]. Note that the term "disappearing" is used here, but it would probably be better to say that atoms present in V_0 but not in V_1 are "decoupled" from their environment: the interactions among the "disappearing" atoms are not changed, and do not contribute to $\langle \partial V/\partial\lambda \rangle$. If the disappearing atoms are a separate molecule (say a non-covalently-bound ligand), this can be viewed as a transfer to the gas-phase.

Note that a slightly different setup is required for using soft core potentials compared to older TI-implementations. Specifically, the difference is that to add or remove atoms without soft core potentials, they are transformed into interactionless dummy particles, so both end state prmtop files have the same number of atoms. When using soft core potentials instead, no dummy atoms are needed and the end states should be built without them. Therefore prmtop files for non soft core simulations may have to be adapted to be used with soft core potentials and vice versa.

All bonded interactions of the unique atoms are recorded separately in the output file (see below). Any bond, angle, dihedral or 1-4 term that involves at least one appearing or disappearing atom is not scaled by λ and does not contribute to $\langle \partial V/\partial\lambda \rangle$. Therefore, output from both processes will not be identical when soft core potentials are used. Softcore transformations avoid the origin singularity effect and therefore linear mixing can (and should) always be used with them. Since the unique atoms become decoupled from their surroundings at high or low lambdas and energy exchange between them and surrounding solvent becomes inefficient, a Berendsen type thermostat should not be used for SC calculations. Unlike in previous versions, SHAKE constraints are not automatically removed from bonds between common and unique atoms. Instead, the coordinates corresponding to common atoms in $V_0$are synchronized with those of V_1. The original behavior can be restored using *tishake*. The icfe and klambda parameters should be set to 1 for a soft core run and the desired lambda value will be specified by clambda. When using softcore potentials with *sander*, λ values should be picked so that $0.01 <$ clambda < 0.99. The *pmemd* implementation allows lambda to be set to any value between 0.0 and 1.0, thus simulations at the endpoints are possible.

Additionally, the following parameters are available to control the TI calculation:

`ifsc` Flag for soft core potentials

- **= 0** SC potentials are not used (default)
- **= 1** SC potentials are used. Be sure to use prmtop files that are suitable for this, i.e. not-containing dummy atoms (see above)

`scalpha` The α parameter in 8.5 and 8.6, its default value is 0.5. Other values have not been extensively tested

`logdvdl` If set to .ne. 0, a summary of all $\partial V/\partial\lambda$ values calculated during every step of the run will be printed out at the end of the simulation for postprocessing.

`dvdl_norest` This option is now deprecated. Restraints involving soft core atoms are now decoupled from the rest of the system. The energy is listed separately and does not contribute to $\partial V/\partial\lambda$.

`dynlmb` If set to a value .gt. zero, clambda is increased by dynlmb every ntave steps. This can be used to perform simulations with dynamically changing lambdas.

`crgmask` Specifies a number of atoms (in ambmask format) that will have their atomic partial charges set to zero. This is mainly for convenience because it removes the need to build additional prmtop files with uncharged atoms for TI calculations involving the removal of partial charges.

8.1.4.1 Input flags specific to Sander

`scmask` Specifies the unique (soft core) atoms for this process in ambmask format. This, along with crgmask, is the only parameter that will frequently be different in the two mdin files for V_0 and V_1. It is valid to set scmask to an empty string. A summary of the atoms in scmask is printed at the end of mdout.

8.1.4.2 One step transformations using soft core electrostatics

Alternatively to the two-step process of removing charges from atoms first and then changing the vdW parameters of chargeless atoms in a second TI calculation, *sander and pmemd* also have a soft core version of the Coulomb equation implemented for single step transformations under periodic boundary conditions. This is automatically applied to all atoms in scmask and their interactions with common atoms are given by:

$$V_{V_0,disappearing} = (1-\lambda)\frac{q_i q_j}{4\pi\epsilon_o\sqrt{\beta\lambda + r_{ij}^2}} \tag{8.7}$$

for disappearing atoms. Replace λ by $(1-\lambda)$ and vice versa for the form for appearing atoms. This introduces a new parameter β which controls the 'softness' of the potential. This is set in the input file via:

`scbeta` The parameter β in 8.7. Default value is 12A^2, other values have not been extensively tested.

With the use of soft core vdW and electrostatics interactions, arbitrary changes between systems are possible in single TI calculations. However, due to the unusual potential function forms introduced, it is not always clear that a single-step calculation will converge faster than one broken down into several steps. Ref. [151] contains detailed information on the performance of such single step TI calculations.

8.1.5 Collecting potential energy differences for FEP calculations

In addition to the Thermodynamic Integration capabilities described above, *sander* can also collect potential energy values during free energy simulation runs for postprocessing by e.g. the Bennett acceptance ratio scheme. This will make sander calculate at given points during the simulation the total potential energy of the system as it would be for different λ-values at this conformation. This functionality is controlled by:

`ifmbar` If set to 1 (Default = 0), additional output is generated for later postprocessing.

mbar_states number of lambda windows considered.

`mbar_lambda` lambda windows simulated.

`For` example, if you want to run mbar with 15 lambda windows at 0.00, 0.05, 0.10, 0.15, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.85, 0.90, 0.95, 1.00, you would use the following options:

```
ifmbar = 1,
mbar_states = 15,
mbar_lambda = 0.00, 0.05, 0.10, 0.15, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80, 0.85, 0.90, 0.95,
```

`The` options below have been deprecated. They are here for anyone using AMBER 16 or older, but will not work in AMBER 18.

`bar_intervall` Compute potential energies every bar_intervall steps (Default = 100)

`bar_l_min` Minimum λ-value (Default = 0.1)

`bar_l_max` Maximum λ-value (Default = 0.9)

`bar_l_incr`	The increment to increase λ by between the minimum and maximum (Default = 0.1)

Such energy collection will normally be part of a regular free energy calculation (using icfe=1 and ifsc=1) involving simulations at various λ-values. Activating this functionality will not have any influence on the simulation trajectory which will evolve according to the preset *clambda* value, it is merely a book-keeping scheme that removes the necessity of postprocessing output files later.

8.2 Linear Interaction Energies

sander contains rudimentary facilities to compute binding free energies using the linear interaction energy model.

`ilrt`	if set to 1, turns on the computation of LIE contributions (default=0)
`lrt_interval`	Computer LIE contributions every lrt_interval MD steps (default=50)
`lrtmask`	The 'solute'. Interaction Energies between the atoms in lrtmask and the remainder of the system are computed.

The LIE facilities work by computing the system energies several times using different charge and vdW-parameter sets. This results in reduced performance if lrt_interval is set to less than approx. 10. The LIE output at the end of the *mdout* file gives the electrostatic interaction energy between the solute and rest of the system *times 0.5*, i.e. in accordance with the original formulation of LIE theory. The solute SASA and vdW-interaction energy with its surroundings is calculated unscaled.

8.3 Adaptively Biased MD, Steered MD, Umbrella Sampling with REMD and String Method

8.3.1 Overview

The following describes a suite of modules useful for the calculation of the free energy associated with a reaction coordinate $\sigma(\mathbf{r}_1, \ldots, \mathbf{r}_N)$ (which is defined as a smooth function of the atomic positions $\mathbf{r}_1, \ldots, \mathbf{r}_N$):

$$f(\xi) = -k_B T \ln \left\langle \delta\left[\xi - \sigma(\mathbf{r}_1, \ldots, \mathbf{r}_N)\right] \right\rangle,$$

(the angular brackets denote an ensemble average, k_B is the Boltzmann constant and T is the temperature) that is also frequently referred to as the *potential of mean force*.

Specifically, new frameworks are provided for equilibrium umbrella sampling and steered molecular dynamics that enhance the functionality delivered by earlier implementations (described earlier in this manual), along with a new Adaptively Biased Molecular Dynamics (ABMD) method [152] that belongs to the general category of umbrella sampling methods with a time-dependent potential. Such methods were first introduced by Huber, Torda and van Gunsteren (the Local Elevation Method [153]) in the molecular dynamics (MD) context, and by Wang and Landau in the context of Monte Carlo simulations [154]. More recent approaches include the metadynamics method [155, 156]. All these methods estimate the free energy of a reaction coordinate from an evolving ensemble of realizations, and use that estimate to bias the system dynamics to flatten an effective free energy surface. Collectively, these methods may all be considered to be umbrella sampling methods with an evolving potential. The algorithms discussed here were developed by the group of Prof. Celeste Sagui (sagui@ncsu.edu) and Prof. Christopher Roland (cmroland@ncsu.edu); the current version was implemented by Dr. Volodymyr Babin.

The ABMD method grew out of attempts to speed up and streamline the metadynamics method for free energy calculations with a *controllable* accuracy. It is characterized by a favorable scaling in time, and only a few (two) control parameters. It is formulated in terms of the following equations:

$$m_a \frac{\mathrm{d}^2\mathbf{r}_a}{\mathrm{d}t^2} = \mathbf{F}_a + \frac{\partial}{\partial \mathbf{r}_a} U\left[t|\sigma\left(\mathbf{r}_1, \ldots, \mathbf{r}_N\right)\right],$$

$$\frac{\partial U(t|\xi)}{\partial t} = \frac{k_B T}{\tau_F} G\left[\xi - \sigma\left(\mathbf{r}_1, \ldots, \mathbf{r}_N\right)\right],$$

where the first equation represents Newton's law that governs ordinary MD (temperature and pressure regulation terms are not shown) augmented with an additional force coming from the time dependent biasing potential $U(t|\xi)$ $[U(t=0|\xi)=0]$, whose time evolution is given by the second equation. $G(\xi)$ is a positive definite and symmetric kernel, which may be thought of as a smoothed Dirac delta function. For large enough τ_F (the flooding timescale) and small kernel width, the biasing potential $U(t|\xi)$ converges towards $-f(\xi)$ as $t \to \infty$.

Our numerical implementation of the ABMD method involves the use of a bi-weight kernel along with the use of cubic B-splines (or products thereof) to discretize the biasing potential $U(t|\xi)$ w.r.t. ξ, and an Euler-like scheme for time integration. ABMD admits two important extensions, which lead to a more uniform flattening of $U(t|\xi) + f(\xi)$ due to an improved sampling of the "evolving" canonical distribution. The first extension is identical in spirit to the *multiple walkers metadynamics* [157, 158]. It amounts to carrying out several different MD simulations biased by the same $U(t|\xi)$, which evolves via:

$$\frac{\partial U(t|\xi)}{\partial t} = \frac{k_B T}{\tau_F} \sum_{\alpha} G\left[\xi - \sigma\left(\mathbf{r}_1^{\alpha}, \ldots, \mathbf{r}_N^{\alpha}\right)\right],$$

where α labels different MD trajectories. A second extension is to gather several different MD trajectories, each bearing its own biasing potential and, if desired, its own distinct collective variable, into a generalized ensemble for "replica exchange" with modified "exchange" rules [159–161]. Both extensions are advantageous and lead to a more uniform flattening of $U(t|\xi) + f(\xi)$.

In order to assess and improve the accuracy of the free energies, the ABMD accumulations may need to be followed up with equilibrium umbrella sampling runs, which make use of the biasing potential $U(t|\xi)$ as is. Such a procedure is very much in the spirit of adaptive umbrella sampling. With these runs, one calculates the biased probability density:

$$p^B(\xi) = \left\langle \delta\left[\xi - \sigma(\mathbf{r}_1, \ldots, \mathbf{r}_N)\right] \right\rangle_B.$$

The idea here is that if, as a result of an ABMD run, $f(\xi) + U(t|\xi) = 0$ exactly, then the biased probability density $p^B(\xi)$ would be flat (constant). In practice, this is typically not the case, but one can use $p^B(\xi)$ to "correct" the free energy via:

$$f(\xi) = -U(\xi) - k_B T \ln p^B(\xi).$$

With the ABMD procedure, one can obtain accurate free energy curves and equilibrium properties. We note that to obtain ABMD free energies requires a (minor) amount of post-processing by means of the nfe-umbrella-slice utility freely available in AmberTools as described in Subsection 8.3.7. This methodology has been applied to a variety of biomolecular systems, including small peptides [152, 162, 163], sugar puckering [164], polyproline systems [165–167], guest-host systems [168, 169], polyglutamine systems [170, 171], and DNA systems [172–176]. In addition, SMD simulations (discussed below) have been used to examine transition pathways and mechanisms, to estimate free energy differences [166, 177], and to calculate transition rates [178–180].

While the above represents the basic ABMD implementation, AMBER20 introduced three additional algorithms – the Well-Tempered (WT) ABMD, a selection mechanism for multiple walker ABMD and Driven ABMD (D-ABMD)[181] – all of which enhance the stability and convergence of an ABMD simulation. Also implemented is the Swarms-of-Trajectories String Method (STSM) [182], which gives a way of exploring the Minimum Free Energy Path (MFEP) on free energy lansdscape. Current version of these codes were inplemented by Dr. Mahmoud Moradi (moradi@uark.edu), Dr. Feng Pan (fpan3@ncsu.edu) and Ashkan Fakharzadeh (afakhar@ncsu.edu).

The Well-Tempered ABMD: An alternative to the follow-up equilibrium simulations for increased ABMD accuracy is provided by the WT-ABMD, which is implemented in the spirit of the WT-metadynamics [183]. In the original ABMD implementation, the history dependent biasing potential is built up at a fixed rate:

$$U(\xi,t) = U^0(\xi) + \int_0^t dt'\, \omega\, G(\xi - \xi^{t'}), \tag{8.8}$$

in which $U(\xi,t)$ is the biasing potential at time t, U^0 is an arbitrary function that typically represents the initial guess for the biasing potential (in the absence of a guess, this is assumed to be flat) and $\omega = k_B T/\tau_F$ is a constant, unbiased rate. As the simulation proceeds and reaches convergence, then $\langle U(\xi, t \to \infty)\rangle_a \approx U^s(\xi) + u(t)$, in which $\langle . \rangle_a$ is the ensemble-average over the adaptive trajectories, the stationary term is $U^s \approx -F(\xi)$, and $u(t)$ is an additive time-dependent constant [183]. Unfortunately, updating the biasing potential at the same rate throughout the simulation may lead to a poorly converged result, since the biasing potential ends up fluctuating around $-F(\xi)$ with an amplitude that depends on ω.

One way to resolve this problem is to update the kernel at a non-uniform rate by means of a "well-tempered" ω:

$$U(\xi,t) = U^0(\xi) + \int_0^t dt'\, \omega(\xi^{t'}, t')\, G(\xi - \xi^{t'}), \tag{8.9}$$

in which $\omega(\xi,t)$ is a time-dependent, non-uniform rate chosen to be $\omega_0\, \mathrm{e}^{-\beta' U(\xi,t)}$ ($1/\beta' = k_B T'$ where T' is a pseudo-temperature) that reduces to a constant ω_0 in the $\beta' \to 0$ limit (*i.e.*, resulting in conventional ABMD). With this choice, one can show that $\langle U(\xi, t \to \infty)\rangle_a \approx U^s(\xi) + u(t)$, ($u(t)$ is an additive constant) in which $U^s(\xi)$ and $F(\xi)$ are related via $U^s(\xi) = -(1 + \frac{\beta'}{\beta})^{-1} F(\xi)$ or $F(\xi) = -(1 + \frac{T}{T'}) U^s(\xi)$. This way of updating the biasing potential leads to a considerably smoother convergence to the desired free energy and more stable ABMD simulations.

Multiple walker selection algorithm: The ABMD multiple walker algorithm can be improved by allowing for periodic interactions between the different walkers and "resampling" on-the-fly. The rationale behind this is that not all walkers are equally effective in sampling the configuration space. A situation that is all too common is that different walkers end up being “bunched up" or clustered together in some local metastable region, because of hidden barriers that are oriented along orthogonal degrees of freedom to the reaction coordinate. To improve this situation, one would like to facilitate walkers that are sampling the undersampled regions of phase space, and force the walkers in the oversampled regions to move away and explore regions not yet covered. Such an algorithm has previously been implemented via scripts in the NAMD code for the adaptive biasing force algorithm [184].

A resampling or selection algorithms for interacting multiple walkers requires a continual monitoring of the walkers by means of a periodic evaluation of a fitness function and a resampling of the walkers according to their fitness efficiency[184]. Efficient walkers that are wandering in the undersampled regions are enhanced by being cloned, while inefficient walkers found in the oversampled regions of phase space are correspondingly killed. This procedure is then repeated periodically during the simulation, thereby accelerating convergence to a more uniform distribution of walkers and flattening of the free energy landscape.

Our specific interacting/resampling/selection multiple-walker algorithm is implemented as follows. Each walker n is assigned a weight w_n, which is evaluated at the end of each resampling period of time τ. At the i^{th} resampling period, i.e., from time $t_{(i-1)} = (i-1)\tau$ to $t_i = i\tau$, walker n moves through configuration space building up its own trajectory $\left(r_1^n, \ldots, r_N^n\right)$. The weights are then tested and updated every fixed time interval of length τ. Specifically, after the i^{th} time interval, weights are estimated by:

$$w_n = K^{-1} \exp\left(\int_{t_{i-1}}^{t_i} S(\xi_n^t) dt\right),$$

where ξ_n^t represents the collective variable evaluated at time t for trajectory n, $K = \sum_{n=1}^{N_w} w_n$ is the normalization factor, and

$$S(\xi) = C\nabla^2(\rho(\xi))/\rho(\xi),$$

with $\rho(\xi)$ representing the density of microstates in the collective variable space and C a constant. The quantity $S(\xi)$ will be positive typically if the walker is found in the undersampled regions, which have a convex density function. Similarly, a negative $S(\xi)$ value indicates that the system is in the concave region of the density function, which typically is oversampled. In the context of ABMD implementation, the biasing potential is approximately proportional to the histogram of the collective variable by construction, and represents a good estimate for ρ. The implementation is therefore straightforward; the integral above is estimated for each trajectory independently by summing over $S(\xi_n^t)$ at every step from $t = t_{i-1}$ to $t = t_i$, in which Δt is the MD *timestep*. At the end of each period the walkers send their unnormalized weight estimates to the "master processor" to normalize them. A stochastic resampling method is then used to clone/kill the replicas based on their weight factors [184]. The number of copies present in the next period for walker n is determined by the integer number:

$$\begin{aligned} W_1 &= \lfloor \eta_1 + N_w w_1 \rfloor, \\ W_n &= \lfloor \eta_n + N_w \sum_{m=1}^{n} w_m \rfloor - \lfloor \eta_n + N_w \sum_{m=1}^{n-1} w_m \rfloor, \quad \text{for } n > 1. \end{aligned}$$

in which $0 < \eta_n < 1$ is drawn from a uniform distribution (using a random number generator). The atomic coordinates and velocities of the walkers with $N_n > 0$ are "sent" to N_n walkers. The resampling algorithm above guarantees $\sum_n W_n = N_w$.

In terms of an ABMD simulation, the selection algorithm is most beneficial during the initial and middle parts of the simulation when there are large variations in the biasing potential. In the latter parts, when the effective free energy is almost flat, the distribution of walkers should be roughly uniform. In that case, the selection mechanism is unnecessary and, if one wishes to continue the simulation, it is best to proceed with the non interacting multiple walker algorithm. It has been found that a convenient stopping mechanism may be based on the entropy of the weights. Defining $H = \sum_n w_n log(w_n)$, the selection mechanism will be stopped if $E_w = H - log(1/N_w)$ goes below -$\epsilon log(1/N_w)$. Here, $log(1/N_w)$ represents the entropy of uniform weights, and the stopping parameter ϵ varies between $0 \leq \epsilon \leq 1$. When $\epsilon = 0$, the algorithm never stops, while $\epsilon = 1$ forces a stop irrespective of the values of the weights.

In addition to ϵ, there are also two other user-defined variables in the selection algorithm, including the constant C and the interval time τ. While the physical interpretation of τ is straightforward, C represents a pseudo diffusion constant. One may think of the selection algorithm as an induced diffusion in the reaction coordinate space; The larger the value of C, the faster the system will diffuse along the reaction coordinate space. Therefore C determines the strength or aggressiveness of the resampling algorithm. The most efficient value for C is dependent on the nature of the collective variable and the shape of its density ρ. Since the best choice of C for a given problem is somewhat of an art, we refer the interested reader to the ABMD tutorials on the AMBER webpage for insight into choosing this variable. Finally, we also note that the multiple walker selection mechanism can be invoked as is or in conjunction with the WT-ABMD for enhanced stability and convergence.

Driven ABMD: ABMD and SMD schemes are both powerful nonequilibrium sampling methods; however, each comes with its own practical limitations. For instance, SMD is often associated with a very slow convergence if used for free energy calculations. However it can be used to explore the transition paths, at least qualitatively; an advantage over ABMD, in which the system starting from one end of the configuration space (the reactant) may take a long time to visit the other end (the product). SMD and ABMD schemes, however can be integrated into a novel driven adaptive-bias 3 scheme, termed driven ABMD (D-ABMD) that takes advantage of both its driven and adaptive-bias components and is advantageous over both components in isolation. D-ABMD has an advantage over conventional (or well-tempered) ABMD in that it ensures the exploration of the transition pathway (from one end to the other) in the early stages of the simulation and gradually improves the estimate of the free energies almost uniformly along the reaction coordinate. D-ABMD has also an advantage over the conventional SMD in that the effective free energy surface gradually becomes smooth and flat such that the system can move along the reaction coordinate with progressively less amount of work. The D-ABMD method is similar to D-MetaD method, which was recently introduced in Ref.[181] as an example of driven,

```
&colvar
   cv_type = STRING
   cv_ni = N, cv_nr = M
   cv_i = i1, i2, ..., iN
   cv_r = r1, r2, ..., rM
/
```

Figure 8.1: *Syntax of reaction coordinate definition:* `cv_type` *is a* `STRING`*,* `cv_i` *is a list of integer numbers and* `cv_r` *is a list of real numbers.*

adaptive-bias schemes.

In order to combine the two schemes described above, we have developed a driven adaptive-bias scheme that adds an adaptive $U_a(\xi,t)$ and a driving $U_d(\xi,t)$ potential to the Hamiltonian. We use an iterative approach in which an independent simulation is performed from time t = 0 to t = T in the n^{th} iteration (n = 1, 2, . . .), biased by the potential $U_d(\xi,t) + U_a^n(\xi,t)$ in which $U_d(\xi,t) = \frac{k}{2}(\xi - \eta(t))^2$ for all n ($\eta(t)$ is moving center of the SMD harmonic potential in the ξ space), and:

$$U_a^n(\xi,t) = U^{n-1}(\xi) + \int_0^t dt' \omega(\xi^{t'},t')K(\xi - \xi^{t'})e^{-\beta\omega^{t'}}$$

in which ω^t is either defined as the accumulated work or the transferred work. The accumulated and transferred works are defined as $\omega_{ac}^t = \int_0^t dt' \frac{\partial}{\partial t'} U_d(\xi^{t'},t')$ and $\omega_{tr}^t = \omega_{ac}^t - U_d(\xi^t,t)$. Theoretically the $e^{-\beta\omega_{tr}^{t'}}$ factor or "constant weight" is more accurate but for practical reasons the $e^{-\beta\omega_{ac}^{t'}}$ factor or "pulling wight" is preferred. Particularly, in our algorithm, the constant weight $e^{-\beta\omega_{tr}^{t'}} = e^{-\beta\omega_{ac}^{t'}}e^{\beta U_d(\xi^{t'},t')}$ may become instable for large biasing potentials. To avoid the instability in either case a cutoff for ω^t is used (i.e., the algorithm will not be applied if ω^t is smaller than the cutoff). At the moment, Driven ABMD is only applicable to one-dimensional reaction coordinate.

If any of these modules prove to be useful, please consider quoting the following papers: V. Babin, C. Roland and C. Sagui, *"Adaptively biased molecular dynamics for free energy calculations"*, J. Chem. Phys. **128**, 134101 (2008); V. Babin, V. Karpusenka, M. Moradi, C. Roland and C. Sagui, *"Adaptively biased molecular dynamics: an umbrella sampling method with a time-dependent potential"*, Int. J. Quant. Chem. **109**, 3666 (2009).

From Amber16, we implement these modules from SANDER to PMEMD and the modules are GPU compatible. To keep the consistency in format, we do a series of changes and updates to the usage of these modules. One big change is that you must set ***infe*** = 1 in &cntrl to activate these modules. Also, the input format has been changed to namelist style and reaction coordinate variables will be read from separate files. For the details, please read Subsection **??**

infe This variable controls the usage of the non-equilibrium free energy method. When *infe=0*, the ABMD and related methods are turned off; when *infe=1*, they are turned on and the blocks `&smd`, `&pmd`, `&abmd`, `&bbmd` and `&stsm` will be recognized. The default value is 0. Note that use of these algorithms may require a (minor) amount of post-processing by means of the nfe-umbrella-slice utility freely available in AmberTools described in Subsection 8.3.7.

8.3.2 Reaction Coordinates

A reaction coordinate is defined in the `colvar` namelist in a separate file. (see Fig. 8.1). This section must contain a `cv_type` keyword along with a value of type `STRING` and a list of integers `cv_i` (the number of integers is defined by `cv_ni`). For some types of reaction coordinates the `colvar` section must also contain a list of real numbers, `cv_r`, whose length is defined by `cv_nr`.

The following reaction coordinates (specfied by `cv_type`) are currently implemented:

DISTANCE: distance (in Å) between two atoms whose indexes are read from the list `cv_i`.

COM_DISTANCE: distance between the center of mass of two atom groups. The `cv_i` list is interpreted as a list of indexes of participating atoms. Zeros separate the groups, the last zero is optional. eg: `cv_i = a1,...,aN,0,b1,...,bM,0`.

DF_COM_DISTANCE: difference of distances between the center of mass of first two atom groups and second two atom groups. The `cv_i` list is interpreted as a list of indexes of participating atoms. Zeros separate the groups, the last zero is optional. eg: `cv_i = a1,...,aN,0,b1,...bM,0,c1,...cL,0,d1,...,dK,0`, `DF_COM_DISTANCE` is `COM_DISTANCE(a1,...,aN,0,b1,...,bM) - COM_DISTANCE(c1,...cL,0,d1,...,dK)`.

LCOD: linear combination of distances (in Å) between pairs of atoms listed in `cv_i` with the coefficients read from `cv_r` list. For example, `i = 1,2,3,4` and `r = 1.0,-1.0` define the difference between 1-2 and 3-4 distances, *i.e.* LCOD = r1*distance(1, 2) + r2*distance(3,4).

ANGLE: angle (in radians) between the lines joining atoms with indexes `i1` and `i2` and atoms with indexes `i2` and `i3`.

COM_ANGLE: angle (in radians) formed by the center of mass of three atom groups. The `cv_i` list is interpreted as a list of indexes of participating atoms. Zeros separate the groups, the last zero is optional. eg: `cv_i = a1,...,aN,0,b1,...bM,0,c1,...,cK,0`.

TORSION: dihedral angle (in radians) formed by atoms with indexes `i1`, `i2`, `i3` and `i4`.

COM_TORSION: dihedral angle (in radians) formed by the center of mass of four atom groups. The `cv_i` list is interpreted as a list of indexes of participating atoms. Zeros separate the groups, the last zero is optional. eg: `cv_i = a1,...,aN,0,b1,...bM,0,c1,...,CK,0,d1,...,dL,0`.

COS_OF_DIHEDRAL: sum of cosines of dihedral angles formed by atoms with indexes in the list `cv_i`. The number of atoms must be a multiple of four.

SIN_OF_DIHEDRAL: sum of sines of dihedral angles formed by atoms with indexes in the list `cv_i`. The number of atoms must be a multiple of four.

PAIR_DIHEDRAL: sum of cosines of a list of angles each formed by summing two neighboring dihedral angles from a list formed by atoms with indices `cv_i`. The number of atoms must be a multiple of four. For a list of dihedral angles such as $\{\alpha_1, \ldots, \alpha_N\}$, PAIR_DIHEDRAL is $\sum_{i=1}^{N-1} cos(\alpha_i + \alpha_{i+1})$ which ranges between $-N+1$ and $N-1$.

PATTERN_DIHEDRAL: a particular pattern-recognizing function defined on a list of dihedral angles formed by atoms with indices `cv_i`. The number of atoms must be a multiple of four. The definition is particularly relevant for the dihedral angles with a binary-like behavior of being either around 0 or 180 (e.g., ω backbone dihedral angle). For a list of dihedral angles such as $\{\alpha_1, \ldots, \alpha_N\}$, PATTERN_DIHEDRAL is $\sum_{i=1}^{N} cos^2(\alpha_i/2)2^{i-1}$ which ranges between 0 and $2^N - 1$.

R_OF_GYRATION: radius of gyration (in Å) of atoms with indexes given in the `cv_i` list (mass weighted).

MULTI_RMSD: `RMS` (in Å, mass weighted) of `RMSD`s of several groups of atoms w.r.t. reference positions provided in the `cv_r` list. The `cv_i` list is interpreted as a list of indexes of participating atoms. Zeros separate the groups. An atom may enter several groups simultaneously. The `cv_r` array is expected to contain the reference positions (without zero sentinels). The implementation uses the method (and the code) introduced in Ref. [185]. An example of variable of this type is presented in Fig. 8.2. Two groups are defined here: one comprises the atoms with indexes 1, 2, 3, 4 (line 3 in Fig. 8.2, numbers prior to the first zero) and another one of atoms with indexes 3, 4, 5. The code will

```
&colvar
   cv_type = 'MULTI_RMSD'
   cv_ni = 9, cv_nr = 21,
   cv_i = 1, 2, 3, 4, 0, 3, 4, 5, 0 ! the last zero is optional
   cv_r = 1.0, 1.0, 1.0, ! group #1, atom 1
        2.0, 2.0, 2.0, ! group #1, atom 2
        3.0, 3.0, 3.0, ! group #1, atom 3
        4.0, 4.0, 4.0, ! group #1, atom 4
       23.0, 23.0, 23.0, ! group #2, atom 3
          4.0, 4.0, 4.0, ! group #2, atom 4
          5.0, 5.0, 5.0 ! group #2, atom 5
/
```

Figure 8.2: *An example of `MULTI_RMSD` variable definition.*

first compute the (mass weighted) RMSD (R_1) of atoms belonging to the first group w.r.t. reference coordinates provided in the `cv_r` array (first $12 = 4 \times 3$ real numbers of it; lines 4, 5, 6, 7 in Fig. 8.2). Next, the (mass weighted) RMSD (R_2) of atoms of the second group w.r.t. the corresponding reference coordinates (last $9 = 3 \times 3$ elements of the `cv_r` array in Fig. 8.2) will be computed. Finally, the code will compute the value of the variable as follows:

$$\text{value} = \sqrt{\frac{M_1}{M_1 + M_2} R_1^2 + \frac{M_2}{M_1 + M_2} R_2^2}\,,$$

where M_1 and M_2 are the total masses of atoms in the corresponding groups.

N_OF_BONDS:

$$\text{value} = \sum_p \frac{1 - \left(r_p/r_0\right)^6}{1 - \left(r_p/r_0\right)^{12}}\,,$$

where the sum runs over pairs of atoms p, r_p denotes distance between the atoms of pair p and r_0 is a parameter measured in Å. The `cv_r` array must contain exactly one element that is interpreted as r_0. The `cv_i` array is expected to contain pairs of indexes of participating atoms. For example, if 1 and 2 are the indexes of Oxygen atoms and 3, 4, 5 are the indexes of Hydrogen atoms and one intents to count all possible O-H bonds, the `cv_i` list must be (1, 3, 1, 4, 1, 5, 2, 3, 2, 4, 2, 5), that is, it must explicitly list all the pairs to be counted.

HANDEDNESS:

$$\text{value} = \sum_a \frac{\mathbf{u}_{a,3} \cdot [\mathbf{u}_{a,1} \times \mathbf{u}_{a,2}]}{|\mathbf{u}_{a,1}|\,|\mathbf{u}_{a,2}|\,|\mathbf{u}_{a,3}|}\,,$$

where

$$\begin{aligned}
\mathbf{u}_{a,1} &= \mathbf{r}_{a+1} - \mathbf{r}_a \\
\mathbf{u}_{a,2} &= \mathbf{r}_{a+3} - \mathbf{r}_{a+2} \\
\mathbf{u}_{a,3} &= (1-w)(\mathbf{r}_{a+2} - \mathbf{r}_{a+1}) + w(\mathbf{r}_{a+3} - \mathbf{r}_a)\,,
\end{aligned}$$

and $\mathbf{r}_a$ denote the positions of participating atoms. The `cv_i` array is supposed to contain indexes of the atoms and the `cv_r` array may provide the value of w ($0 \le w \le 1$, the default is zero).

```
&colvar
   cv_type = 'N_OF_STRUCTURES'
   cv_ni = 9, cv_nr = 23,
   cv_i = 1, 2, 3, 4, 0, 3, 4, 5, 0 ! the last zero is optional
   cv_r = 1.0, 1.0, 1.0, ! group #1, atom 1
        2.0, 2.0, 2.0, ! group #1, atom 2
        3.0, 3.0, 3.0, ! group #1, atom 3
        4.0, 4.0, 4.0, ! group #1, atom 4
        1.0,           ! R0 for group #1
      23.0, 23.0, 23.0, ! group #2, atom 3
        4.0, 4.0, 4.0, ! group #2, atom 4
        5.0, 5.0, 5.0, ! group #2, atom 5
        2.0           ! R0 for group #2
/
```

Figure 8.3: *An example of* `N_OF_STRUCTURES` *variable.*

N_OF_STRUCTURES:

$$\text{value} = \sum_g \frac{1-\left(R_g/R_{0,g}\right)^6}{1-\left(R_g/R_{0,g}\right)^{12}},$$

where the sum runs over groups of atoms, R_g denotes the `RMSD` of the group g w.r.t. some reference coordinates and $R_{0,g}$ are positive parameters measured in Å. The `cv_i` array is expected to contain indexes of participating atoms with zeros separating different groups. The elements of the `cv_r` array are interpreted as the reference coordinates of the first group followed by their corresponding R_0; then followed by the reference coordinates of the atoms of the second group, followed by the second R_0, and so forth. To make the presentation clearer, let us consider the example presented in Fig. 8.3. The atomic groups and reference coordinates are the same as the ones shown in Fig. 8.2. Lines 7 and 11 in Fig. 8.3 contain additional entries that set the values of the threshold distances $R_{0,1}$ and $R_{0,2}$. To compute the variable, the code first computes the mass weighted `RMSD` values R_1 and R_2 for both groups –much like in the `MULTI_RMSD` case– and then combines those in a manner similar to that used in the `N_OF_BONDS` variable.

$$\text{value} = \frac{1-\left(R_1/R_{0,1}\right)^6}{1-\left(R_1/R_{0,1}\right)^{12}} + \frac{1-\left(R_2/R_{0,2}\right)^6}{1-\left(R_2/R_{0,2}\right)^{12}}.$$

In other words, the variable "counts" the number of structures that match (stay close in `RMSD` sense) with the reference structures.

QUATERNIONS: Describing large-scale atomistic conformational changes in biomolecular systems requires one to deal with orientational changes of atomistic domains with large numbers of atoms. While there are several ways of defining a collective variable that quantifies an orientation based conformational change, the orientation quaternion technique[186–189] has proven successful as a well-behaved, flexible method for defining system-specific CVs, specifically aimed at inducing interdomain orientational changes or restraining the orientation of certain domains. The CVs in the orientation quaternion class, are all derived from an 'optimal rotation' between a set of reference coordinates $\mathbf{X}_k$ ($1 \le k \le N$; where N is the number of atoms involved) and the set of target coordinates $\mathbf{Y}_k$. A 'quaternion' is introduced as a four-component vector that can be expressed as $q_0 + q_1\hat{i} + q_2\hat{j} + q_3\hat{k}$ where q_0 and $q_1\hat{i} + q_2\hat{j} + q_3\hat{k}$ are called scalar and vector parts respectively. The optimal rotation can be parametrized[1] by a unit quaternion, $\hat{q} = (q_0, q_1, q_2, q_3)$, that minimize $< ||\hat{q}\mathbf{X}_k\hat{q}^* - \mathbf{Y}_k||^2 >$ in which $< . >$ denotes an average over k,q^* is the conjugate of q and $||q||^2 = q^*q$ (see Ref.[186] for more details). The optimal rotation unit quaternion (or orientation quaternion)$\hat{q}$ can be written as $\left(cos(\theta/2), sin(\theta/2)\hat{u}\right)$, where θ is the optimal rotation angle and $\hat{u}$ is a unit vector associated with the optimal axis of rotation. To deal with large atomistic conformational changes, a set of quaternion-based CVs has implemented in AMBER20. For the details of usage, send emails to Ashkan Fakharzadeh (afakhar@ncsu.edu), Dr. Feng Pan (fpan3@ncsu.edu), and Prof. Mahmoud Moradi (moradi@uark.edu). The specific quaternion-based CVs implemented are: ORIENTATION_ANGLE, ORIENTATION_PROJ, TILT, SPINANGLE, QUATERNION0, QUATERNION1, QUATERNION2, and QUATERNION3.

Orientation (QUATERNION0,...,QUATERNION3): These define the orientation of several atoms with respect to a set of reference coordinates in terms of a unit quaternion vector $\hat{q} = (q_0, q_1, q_2, q_3)$ according to the method introduced in Ref.[186, 187]. These variables return the best-fit rotation, also used in best-fit RMSD calculation procedures, to superimpose the coordinates $\mathbf{X}$ onto a set of reference coordinates $\mathbf{X}_0$. The unit quaternion $\hat{q} = (q_0, q_1, q_2, q_3)$ can be written as $\left(cos(\theta/2), sin(\theta/2)\hat{u}\right)$, where θ is the rotation angle and $\hat{u}$ is a unit vector associated with the axis of rotation; for example, a rotation of $90°$ around the z axis $(0, 0, 1)$ is expressed as$(cos(90°/2), 0.0, 0.0, sin(90°/2)) = (\sqrt{2}/2, 0, 0, \sqrt{2}/2)$. The components of the unit quaternion (q_0, q_1, q_2, q_3) were implemented separately as QUATERNION0, QUATERNION1, QUATERNION2, and QUATERNION3 CVs. To find the orientation, all four CVs QUATERNION0,...,QUATERNION3 are being used. To calculate the quaternion CVs one needs to specify a list of participating atoms and also their reference coordinates. The reference coordinates may be passed to AMBER either via direct specification inside the CV call, or by passing the name of a reference coordinates file. It is recommended that if the set of participating atoms is small (say no larger than 15), then these are specified directly inside the CV call. Otherwise, the passing of information via filename is recommended since these lists may contain hundreds if not thousands of atoms. Relevant parameters pertaining to the input of this information are: *cv_ni* represents the number of participating atoms; *cv_i* represents the list of the indices of all participating atoms; *cv_r* represents the reference coordinates (when passed directly) and $refcrd\_file$ is the filename for the reference coordinates when they are to be read from file. The file $refcrd\_file$ should be an AMBER coordinates/restart file containing coordinates, velocities, etc. of all atoms. The list participating atoms, *cv_i*, and their reference coordinates (*cv_r* and or $refcrd\_file$) must be the same for all QUATERNION0,...,QUATERNION3. The CVs are linked together using an attribute*'q_index'*. The*'q_index'* accepts an integer between $1, ..., 100$, where its default value is one. The Fig. 8.4 is an example of Quaternion CVs syntax. An example this type of CVs is presented in the Fig. 8.5. Two set of orientations are defined here: each set consists of QUATERNION0,.., QUATERNION3. The first set comprises 18 atoms with indexes $11, 41, 48, 74, 104, \ldots$ and another one of 24 atoms with indexes $12, 16, 46, 55, 75, \ldots$. A file, 'inpcrd' is used as an AMBER coordinate/restart file to read reference coordinates. There is no need to set *'q_index'* for the first four quaternions since the default value is one, but it is set to be 2 for all quaternion CVs in the second set to link and normalize them. The returned value of each QUATERNION0,...,QUATERNION3 CVs is the corresponding component of the unit orientation vector $\hat{q} = (q_0, q_1, q_2, q_3)$.

ORIENTATION_ANGLE: The angle of rotation $\theta = 2cos^{-1}(q_0)$ between the current and the reference positions. This angle is between $0°$ to $180°$. The *cv_i* list is interpreted as a list of indexes of participating atoms.

$$\textit{orientation angle: } \theta = 2cos^{-1}(q_0)$$

ORIENTATION_PROJ: The cosine of the angle of rotation θ between the current and the reference positions. While ORIENTATION_ANGLE diverges near $\theta = 0$, because of $\nabla_X\theta$, ORIENTATION_PROJ might be used instead to apply forces. The range of ORIENTATION_PROJ is $[-1, 1]$. The *cv_i* array

[1]Assuming both sets have been already shifted to bring their barycenters to the origin (optimum translation).

```
&colvar
   cv_type = 'QUATERNION0'

   ! number of participating atoms
   cv_ni = ni

   ! index of participating atoms
   cv_i = a1, a2,..., aN

   ! AMBER coordinate/restart file to read reference coordinates
   refcrd_file = 'refcrd_file'

   ! number of references which must be 3*ni; Should not be set if
   ! refcrd_file is being used
   cv_nr = nr

   ! reference coordinates of participating atoms; Should not be set if
   ! refcrd_file is being used
   cv_r = a1x, a1y, a1z, a2x, a2y, a2z, a3x, a3y, a3z, ...

   ! an arbitrary integer between 1 to 100
   q_index = n
/
```

Figure 8.4: *Syntax of Quaternion reaction coordinates.*

is supposed to contain indexes of the atoms.

$$\textit{orientation proj: } 2q_0^2 - 1$$

SPINANGLE: Angle of rotation ϕ around a given unit axis $\hat{\mathbf{e}}$. The axis $\hat{\mathbf{e}}$ is being used to decompose a complete orientation rotation in two sub-rotations, spin ϕ and tilt ω. An advantage of this decomposition is ϕ and ω have the same values, regardless of which one is applied first (in comparison to Euler angles methods). The participating atoms with indexes are given in the *cv_i*. The*'axis'* must provide three components of the axis[2]$\hat{\mathbf{e}}$ in A°. The default axis of rotation is $(0.0, 0.0, 1.0)$. The range of SPINANGLE is between$[-180 : 180]$ degrees. The reference coordinates are specified either via *cv_r* or *refcrd_file*.

$$\textit{spin angle: } \phi = 2tan^{-1}(\mathbf{q}.\mathbf{e}/q_0)$$

where $\mathbf{q}$ is the vector part of quaternion, namely (q_1, q_2, q_3). An example of SPINANGLE cv is presented in Fig. 8.6. The same atoms as example one are used, but the axis of rotation is set to be 'x-axis'. The reference coordinates are given by *cv_nr*, *cv_r* options.

TILT: Cosine of the rotation orthogonal to an unit given axis. The tilt angle ω, shows a rotation away from the direction $\hat{\mathbf{e}}$. The tilt combined with the 'spin' sub-rotation provides the complete orientation rotation of a group of atoms. Similar to ORIENTATION_PROJ, to avoid the discontinuity around 0° and 180°, the cosine of the tilt is implemented instead of the tilt angle itself, so that derivatives are continuous almost everywhere. The *cv_i* and *'axis'* are the participating atoms with indexes and the given axis, respectively. The reference coordinates are specified either via *cv_r* or *refcrd_file*. The value of TILT is between -1 to 1, where the value 1 represents an orientation fully

[2]The axis is from the origin$(0.0, 0.0, 0.0)$ to that point.

parallel to $\hat{\mathbf{e}}$ ($\omega = 0°$), and the value -1 represents an anti-parallel orientation.

$$\textit{tilt: } t = cos(\omega) = 2\left(\frac{q_0}{cos(\frac{tan^{-1}\mathbf{q}.\mathbf{e}}{q_0})}\right)^2 - 1$$

8.3.3 Steered Molecular Dynamics

The `&smd` namelist, if present in the `MDIN` file, activates the steered `MD` code (the method itself is extensively described in the literature: see for example Ref. [190] and references therein). The prefix NFE appears in several switches to do with steered MD: this stands for "Non-equilibrium Free Energy".

The following is recognized within the `&smd` namelist:

output_file sets the output file name. Default is '`nfe-smd.txt`'.

output_freq sets the output frequency (in `MD` steps). Default is 50.

cv_file sets the collective variable file name. Default is '`nfe-smd-cv`'.

There must be at least one reaction coordinate defined (that is, there must be at least one `&colvar` namelist in the `cv_file`). The steered `MD` code requires that additional entries be present in the `&colvar` namelist:

path the steering path whose elements must be real numbers. The `path` must include at least two elements. The upper limit on the number of entries is 20000. The elements define Catmull-Rom spline used for steering.

npath sets the number of elements in `path`. Default is 0.

path_mode The way steering paths are constructed. There are two modes available. In `SPLINE` mode (default) the path is approximated by a spline that passes through the given points; in `LINES` mode the path is represented by the line segments joining the control points.

harm specifies the harmonic constant. If a single number is provided, e.g., `harm = 10.0`, then it is constant throughout the run. If two or more numbers are provided, e.g., `harm = 10.0, 20.0`, then the harmonic constant follows a Catmull-Rom spline built upon the provided values.

nharm sets the number of elements in `harm`. Default is 0.

harm_mode The way harmonical paths are constructed, similar with `path_mode`.

An example of `MDIN` file and `CV.IN` file for steered `MD` is shown in Fig. 8.7. The reaction coordinate is defined in `cv.in`. The spring constant is set constant throughout the run and the steering path is configured from 5.0 to 3.0. The values of the reaction coordinate, harmonic constant and the work performed on the system are requested to be dumped to the `smd.txt` file every 50 MD steps.

8.3.4 Umbrella sampling

To activate the umbrella sampling code, the `&pmd` namlist must be present in the `MDIN` file.`&pmd` is currently available to both SANDER and PMEMD, and also can be fully applied in GPU accelerated PMEMD. `The output_file`, `output_freq` and `cv_file` entries are recognized just as in the steered `MD` case presented earlier. The `cv_file` must contain at least one `&colvar` namelist section. For umbrella sampling, the `&colvar` section(s) must contain two additional entries:

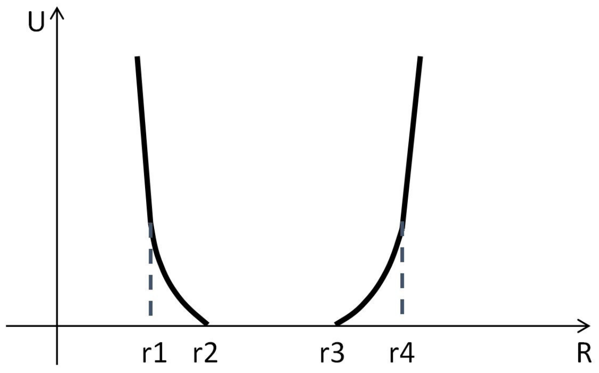
anchor_position: this consists of four real numbers ($r1, r2, r3, r4$) that determine the rectangle of the umbrella (harmonic) potential. The default value is that all of the $r's$ is set to zero.

anchor_strength: two non-negative real numbers ($k1, k2$) that set the harmonic constant for the umbrella (harmonic) potential. The default value is zero.

The umbrella (harmonic) potential U is determined by (supposing R is the value of reaction coordinate)

- $U = k1 * (r1 - r2) * R \quad (R \leqslant r1)$
- $U = 0.5 * k1 * (R - r2)^2 \quad (r1 < R \leqslant r2)$
- $U = 0 \quad (r2 < R \leqslant r3)$
- $U = 0.5 * k2 * (R - r3)^2 \quad (r3 < R \leqslant r4)$
- $U = k2 * (r4 - r3) * R \quad (R > r4)$

A plot of the umbrella potential is shown below



eg1: if $r2 = r3$, $r1 \ll r2$ and $r4 \gg r3$, then the generated U is simply the traditional harmonic potential.

eg2: if $r1$ is slightly less than $r2$ and $r4$ is slightly larger than $r3$, also with very large $k1$, $k2$, the reaction coordinate is restrained in the range ($r2$, $r3$) with no potential added.

An example of an `MDIN` file and `CV.IN` file for an umbrella sampling simulation is shown in Fig. 8.8. The first reaction coordinate here is the angle formed by the lines joining the 5th with 9th and 9th with 15th atoms. It is to be harmonically restrained near 1.0 *rad* (`anchor_position` entry) using the spring of strength 10.0 $kcal/mol/rad^2$ (`anchor_strength` entry). The second reaction coordinate requested in Fig. 8.8 is a dihedral angle (`type = 'TORSION'`) formed by the 1st, 2nd, 3rd and 4th atoms (the `cv_i` array). It is to be restrained near zero with strength 23.8 $kcal/mol/rad^2$. The values of the reaction coordinate(s) are to be dumped every 50 `MD` steps to the `pmd.txt` file. Another example of restraining reaction coordinate in a specific range is shown in Fig. 8.9. The reaction coordinates here are φ and ψ angles of dialanine. φ is restrained between -2.0 *rad* and 2.0 *rad*, ψ is restrained between -1.8 *rad* and 1.8 *rad*.

The `NFE` implementation of umbrella sampling works correctly with the Amber standard replica-exchange MD described earlier in this manual (compatible with different types of REMD for different values of `-rem` flag in both SANDER and PMEMD). For example, the typical umbrella sampling with Hamiltonian Replica Exchange can be performed by setting `-rem` to 3. In this case, both `anchor_position` and `anchor_strength` may be different for different temperatures. Even the number and type of reaction coordinate(s) could vary for different replicas. The output files (set by the `output_file` keyword on a per-replica basis) are `MDIN`-bound, consistent with `-rem`.

8.3.5 Adaptively Biased Molecular Dynamics

The implementation has a very simple and intuitive interface: the code is activated if either an `&abmd` (both SANDER and PMEMD) or an `&bbmd` (both SANDER and PMEMD) namelist is present in the `MDIN` file (the difference between those “flavors” is purely technical and will become clear later). Unlike in the `&smd` and `&pmd` cases, the dimensionality of a reaction coordinate (the number of `&colvar` namelists in the `cv_file`) cannot exceed five (though three is already hardly useful due to statistical reasons).

As previously noted, in order to activate the ABMD and related algorithm, the variable ***infe*** in &cntrl must be set to unity (*i.e.* ***infe*** = 1; default value ***infe*** = 0).

In addition to the `cv_file` entry, the following entries are recognized within the `&abmd` (or `&bbmd`) namelist:

`mode` sets the execution mode. There are three modes available: `'ANALYSIS' | 'UMBRELLA' | 'FLOODING'`. In `ANALYSIS` mode the dynamics is not altered. The only effect of this mode is that the value(s) of the reaction coordinate(s) is(are) dumped every `monitor_freq` to `monitor_file`. In `UMBRELLA` mode, biasing potential from the `umbrella_file` is used to bias the simulation ($\tau_F = \infty$, biasing potential does not change). In `FLOODING` mode the adaptive biasing is enabled.

`monitor_file` sets the name of the file to which value(s) of reaction coordinate(s) (along with the magnitude of biasing potential in `FLOODING` mode) are dumped.

`monitor_freq` the frequency of the output to the `monitor_file`.

`timescale` τ_F, the flooding timescale in picoseconds (only required in `FLOODING` mode).

`umbrella_file` biasing potential file name (the file must exist for the `UMBRELLA` mode).

In `FLOODING` mode, the following two entries are optional:

`snapshots_basename` sets the name of the file to which the biasing potential is dumped during the simulation for snapshot.

`snapshots_freq` the frequency of dumping snapshot biasing potential (in MD steps). If `snapshots_freq` is not specified, the snapshot biasing potential will not be dumped.

and the `&colvar` namelist for `&abmd` method must also contain the following entries:

`cv_min` smallest desired value of the reaction coordinate (required, unless the reaction coordinate is limited from below).

`cv_max` largest desired value of the reaction coordinate (required, unless the reaction coordinate is limited from above).

`resolution` the “spatial” resolution for the reaction coordinate.

To access the biasing potential files created in the course of `FLOODING` simulations, the `nfe-umbrella-slice` utility is provided (it prints a short description of itself if invoked with `--help` option).

The multiple-walker selection algorithm can improve the simulation by resampling between different walkers. The well-tempered ABMD can lead to a smoother convergence to the desired free energy. These two algorithm are implemented to SANDER and PMEMD from Amber16 onwards.

The multiple-walker selection algorithm currently works with `&abmd` only. The algorithm should be used only within the multiple-walker scheme (*i.e.*, when command-line **-rem** flag is set to zero). The following entries are recognized regarding with the selection algorithm (selection algorithm can work with `FLOODING` and `UMBRELLA` mode):

`selection_freq` positive integer number that sets the frequency of the resampling algorithm (in MD steps). If `selection_freq` is not specified, the selection algorithm will not be used.

selection_constant positive real number that sets the parameter *C*. if `selection_freq` is specified, specifying `selection_constant` is required (no default value). Parameter *C* is to determine how strong the selection mechanism is. If *C* is too large, all the walkers will be replaced with the most dominant one. If *C* is too small, there will be no killing/duplicating of walkers.

selection_epsilon positive real number (typically less than unity) that sets the stopping criterion parameter ε . Parameter ε determines the threshold for stopping the selection algorithm. If `selection_epsilon` is not specified, there will be no stop to the algorithm. If `selection_epsilon` is equal or larger than one, the algorithm will be stopped after the first attempt.

The well-tempered flavor can be used within either `&abmd` or `&bbmd` namelist. There are two entries relevant to the well-tempered feature:

wt_temperature positive real number that sets the pseudo-temperature T′. If this flag is not specified, conventional ABMD will be used (*i.e.*, $T' \to \infty$ or $\beta' \to 0$). The smaller the T′; the smoother/slower the convergence.

wt_umbrella_file the file name of true biasing potential after modification by 1 + (T /T′) in which T is the reference temperature of the system (temp0).

An example `MDIN` file and `CV.IN` file for the `&abmd` flavor of `ABMD` is shown in the Fig. 8.10.

In this example, the reaction coordinate is defined as the distance between the 5th and 9th atoms (more than one reaction coordinates might be requested by mere inclusion of additional `&colvar` subsections). The `mode` is set to `FLOODING` thus enabling the adaptive biasing with flooding timescale $\tau_F = 100ps$. The region of interest of the reaction coordinate is specified to be between -1 Å and 10 Å and the resolution is set to 0.5 Å. The lower bound (-1 Å) could have been omitted for `DISTANCE` variable: the default value of zero would be used in such case. The code will try to load the biasing potential from the `umbrella.nc` file and use it as the value of $U(t|\xi)$ at the beginning of the run. The biasing potential built in the course of simulation will be saved to the same file (`umbrella.nc`) every time the `RESTRT` file is written. The selection algorithm is used with the frequency of selection defined as 10000 MD steps and selection constant defined as 0.001. The well-tempered algorithm is also used, with the pseudo-temperature defined as 10000 K in and the true biasing potential will be dumped as `wt_umbrella.nc` file. The `nfe-umbrella-slice` utility can then be used to access its content. An `MDIN` file for the follow up biased run at equilibrium would look much like the one shown in the Fig. 8.10, but with `mode` changed from `FLOODING` to `UMBRELLA`.

Driven ABMD can be performed using &smd block (for the SMD part of the algorithm) along with &abmd block (for the ABMD part of the algorithm). There is no additional flag for the &smd block relevant to the algorithm; however, there are two additional flags to ABMD relevant to the “driven” feature.

driven_weight string that sets the weighting scheme. The default option (i.e., not using the flag) is NONE which indicates no reweighting is used (NOT RECOMMENDED if SMD is performed along ABMD). Other options include CONSTANT and PULLING for constant and pulling reweighting protocols.

driven_cutoff positive real number that sets a cutoff for work for applying the reweighting algorithm (default: 0.0). If the work (accumulated or transferred depending on the scheme) at any given time is lower than the cutoff, no reweighting is done at that particular time. If the cutoff is too small, it may result in instability of the algorithm.

For both SANDER and PMEMD since Amber18, the `&abmd` code works correctly with Amber replica-exchange similar with `&pmd` (that is, for `-rem` flag set to different values). If`-rem` is set to 3, ABMD with replica-exchange is carried out. In such case different replicas can have different temperatures, collective variables and even different `mode`.The monitor and umbrella files are `MDIN`-bound. If number of sander groups exceeds one (the flag -ng is greater than one) and -rem flag is set to zero, the code runs *multiple walkers* ABMD. In both cases the number and type(s) of variable(s) must be the same across all replicas.

Finally, the &bbmd flavor allows one to run replica-exchange (AB)MD with different reaction coordinates and different modes (ANALYSIS, UMBRELLA or FLOODING) in different replicas (along with different temperatures, if desired). This module is outdated since &abmd has been compatible with -rem equals 3. The only advantage of &bbmd is that the number of replicas can be odd numbers if desired by runs, while this cannot be achieved in any -rem types. To applying &bbmd module, the -rem flag must be set to zero and the &bbmd sections must be present in all MDIN files. The MDIN file for the replica of rank zero (first line in the group file) is expected to contain additional information as compared to &abmd case (an example of such MDIN file for replica zero is shown in Fig. 8.11). The MDIN files for all other replicas except zero do not need any additional information, and therefore take the same form as in the &abmd flavor (except that the namelist is changed from &abmd to &bbmd, thus activating a slightly different code path). Each MDIN file may define its own reaction coordinates, have different mode and temperature if desired.

Within the first replica &bbmd namelist the following additional entries are recognized:

exchange_freq number of MD steps between the exchange attempts.

exchange_log_file the name of the file to which exchange statistics is to be reported.

exchange_log_freq frequency of exchange_log_file updates.

mt19937_seed seed for the random generator (Mersenne twister [191]).

mt19937_file the name of the file to which the state of the Mersenne twister is dumped periodically (for restarts).

The MDOUT, MDCRD, RESTRT, umbrella_file and monitor_file files are MDIN-bound in course of the bbmd-enabled run. An example that uses this kind of replica exchange is presented in Ref.164.

8.3.6 Swarms-of-Trajectories String Method

ABMD is a robust method for calculating free energy landscapes as a function of a small number of collective variables. Since the required computer time grows enormously with the number of collective variables, ABMD is best for exploring one- or two-dimensional phase spaces. However, rather than calculating full n-dimensional free energy maps, it is often fruitful to focus on the so-called Minimum Free Energy Path (MFEP) which the system is likely to take when transitioning between two minima. Calculating a MFEP in a complicated phase space is often difficult, and so-called ”string methods"[192][182] represent one of the best approaches for finding the MFEP. Since sampling in string methods is essentially limited to regions around the MFEP, the cost of the method scales linearly with the length of the string or path, but only weakly on the number of collective variables. This results in considerable computational savings since the full free energy landscape is not calculated.

The swarms-of-trajectories string method (STSM)[182] is one of the most popular versions of the string method and has been implemented here by Dr. Moradi (moradi@uark.edu). The module is available in both SANDER and PMEMD from Amber18 onwards. It is a path-finding algorithm that refines a putative transition pathway iteratively until the path is deemed to have been converged. The string is defined by a number of nodes or images parameterized in a high-dimensional space of collective variables, whose position is updated iteratively. The center of each image is first used as a restraining center to generate representative conformations at the current center before allowing of a small change in this center for the next iteration. The change in the center of each image is estimated by averaging over the drifts of a swarm of short unbiased trajectories all starting at the current image position (generated using the constrained simulations. Thus, each iteration consists of a series of restrained and free simulations. In the current serial version of the code, these simulations are performed independently. In parallel versions -- which are more efficient -- a very large number of replicas is required which are run in parallel; this method is particularly efficient on large supercomputers.

To invoke the swarms-of-trajectories string method, the `&stsm` must be invoked in the `MDIN` file. For a string consisting of N_s nodes each requiring M copies $N_s \times M$ replicas will be required. The parallel implemention of the STSM method is based on iterative restrained and free MD simulations followed by a reparametrization of the image centers defined in a multidimensional collective variable space ξ. For the i^{th} iteration, first M copies of the n^{th} image are generated around the old center ξ_n^{i-1} by MD equilibration lasting τ_E timesteps. The generated M copies of the n^{th} image are expected to be close to ξ_n^{i-1} for time τ_E, assuming that the invoked harmonic constant k for the restraining potential is large enough. The parameters τ_E and k thus need to be appropriately chosen in order to ensure that all copies of each image will be close to the image center. The restraint is then released, and each copy (swarm) is allowed to drift for τ_R timesteps. The newly shifted center ξ_n^i for the n^{th} image is then determined by averaging over all drifted copies $\xi_{n,m}^t$ at time $t = \tau_E + \tau_R$. The resulting string of images is then smoothed using a linear interpolation protocol. A smoothing parameter ϵ with $0 \leq \epsilon \leq 1$ determines the smoothness of the curve; it is recommended that ϵ be of the order of $1/(N_s - 1)$. The last setep is a reparameterization, which gain follows a linear interpolation protocol in order to generate N_s equidistant centers along the string. The two key parameters of the method are M and τ_R. Generally, the large the M and the shorter τ_R, the smoother (but slower) the evolution of the MFEP will be. These variables must be optimized empirically, but typically 10 - 30 copies and 5 - 20 ps are reasonable values. It is often advantageous to set $\tau_E = \tau_R$.

An improved sequential repeat version of the algorithm has also been implemented, which avoids the large number of copies and does not require a large number of processors to run. Here a new variable N_R is introduced, as the number of repeat runs for each replica. Now for each copy, it will run around the old center ξ_n^{i-1} for N_R times sequentially. And each repeat run can be equally considered as a parallel run of a new copy around the old center. Namely, the new shifted center will be determined by averaging on $N_R \times M$ copies. So the number of processors needed will be reduced to $1/N_R$, while the running time will be multipled by N_R.

The following is recognized within the &stsm namelist:

image positive integer number that sets the image id (between 1 and N). Default is 0.

repeats positive integer number that sets the number of repeat runs, should be the same for each image and each copy. Equal to parallel implementation when not set. Default is 1.

equilibration non-negative integer number that sets the number of MD steps specified for biased equlibration (restraining) at each iteration. Default is 0.

release Number of MD steps specified for the release (drift) at each iteration. Note: the total number of iterations is determined by the total simulation time (`nstlim` flag in `mdin` file) divided by total time for each iteration given by `equilibration+release`.

smoothing positive number that sets the smoothing parameter for reparametrization (between 0 and 1). Smoothing parameter should be, preferably, on the order of $1/(N_s - 1)$. If this flag is not used, no smoothing will be performed.

report_centers a string that determines if drifted and/or smoothed and/or reparametrized centers will be reported. The default value is `NONE` and other available options include `ALL`,`DRIFT`,`SMOOTHED`, `REPARAMETRIZED`, `NO_DRIFT`, `NO_SMOOTHED`, `NO_REPARAMETRIZED`.

`The output_file`, `output_freq` and `cv_file` entries are recognized just as `&smd` and `&pmd`, the information of reaction coordinates will be read from `cv_file`. The number of collective variables can not exceed five. (here be attention that the `anchor_postion` and `anchor_strength` will be defined using the traditional harmonical potential, different with `&pmd`!). An example of `MDIN` file and `CV.IN` file for STSM in parallel case is shown in Fig. 8.12. Here we run 8 images along the path, with `I` defining the image ID. We run 980 MD steps for equlibration and 20 MD steps release at each iteration, so there are totally 1000 MD steps for each iteration. With nstlim set to 10000, 10 iterations will be carried out.

The smoothing parameter is set to 0.1 and all the centers will be reported. For each image, 16 copies will be run in parallel, with `J` defining the copy ID. The evolution of reaction coordinate will recorded in the file `stsm.00I.J.txt`. For this run, at least $128(8 \times 16)$ processors are needed. Another example of `MDIN` file of equivalent sampling level in sequential case is shown in Fig. 8.13. Here we still have 8 images to run. We set the number of repeats to be 16, namely 16 repeat runs for each image to get the new drifted center. Therefore, 16000 MD steps are needed for one iteration, and so we set nstlim to 160000 to complete 10 iterations. For this run, 8 processors are needed at least.

Part of sample `MDOUT` file is shown in Fig. 8.14. The restoring restraint part will be only in sequential run, since the restraint needs to be restored after each repeat. The values of reaction coordinates before reporting centers are the averaged value over repeats for this copy and the instantaneous value. All the centers will be reported only in the `MDOUT` file of first copy of first image. The drifted centers are the averaged value over copies, and also the smoothed ans reparametrized centers can be reported. Always the reparametrized centers will be extracted to draw the MFEP in the phase space.

8.3.7 Post-processing of biasing potential

When you get the biasing potential (`*.nc` file), you can always use the `nfe-umbrella-slice` utility to access its content and get a friendly-written ASCII file from which one can obtain the free energy map. The output is the free energy value, which is the opposite of the biasing potential ($f = -U$ $(units\ kcal/mol)$). The `nfe-umbrella-slice` utility has been included in AmberTools.

Usage: `nfe-umbrella-slice [options] bias_potential.nc`

Options:

`-h,--help` Print out a usage summary

`-p,--pretend` Only print out the basic properties of source without biasing potential data (off by default)

`-g,--gradient` Print out the gradients (off by default)

`-r,--reset` Set the value of minimum to zero (off by default)

`-t,--translate` Translate the numerical value of biasing potential by a real number (0 by default)

`-d,--dimensions` Set the way of slice in different dimensions. The format is "D1:D2:...:Dn", where n is the number of dimensions. Each D can only be set with one number or three numbers separated by commas. If only one number is set, the variable will be fixed at that value. If three numbers are set, the first two define the boundary of the slice and the last one defines the number of points.

Example:

- `nfe-umbrella-slice -r -d "-5.0,5.0,50" 1d-bias.nc > FE.dat`

This processes the 1-dimensional biasing potential file `1d-bias.nc` and prints out the results to `FE.dat`. The minimum of free energy will be set to zero. The variable will be taken from -5.0 to 5.0 using 50 points.

- `nfe-umbrella-slice -g -t 50.0 -d "1.0:-2.0,2.0,20" 2d-bias.nc > FE.dat`

This processes the 2-dimensional biasing potential file `2d-bias.nc` and prints out the results to `FE.dat`. All the free energy will be incremented by a constant 50.0. The gradients in both dimensions will be printed out. For the first dimension, the variable will be fixed at 1.0; for the second dimension, the variable will be taken from -2.0 to 2.0 using 20 points.

- `nfe-umbrella-slice wt_umbrella.nc > wt_FE.dat`

This processes the biasing potential after WT-ABMD and prints out the results to `wt_FE.dat`. The default dimensional information is obtained and used by the program from the biasing potential file.

```
&colvar
   cv_type = 'QUATERNION0'
   cv_ni = 18,
   cv_i = 11 , 41 , 48 , 74 , 104 , 111 , 137 , 167 , 174 , 199 , 229 , 236, 262 ,
          292 , 299 , 325 , 355 , 362 ,
   refcrd_file = 'inpcrd'
/
&colvar
   cv_type = 'QUATERNION1'
   cv_ni = 18,
   cv_i = 11 , 41 , 48 , 74 , 104 , 111 , 137 , 167 , 174 , 199 , 229 , 236, 262 ,
          292 , 299 , 325 , 355 , 362 ,
   refcrd_file = 'inpcrd'
/
&colvar
   cv_type = 'QUATERNION2'
   cv_ni = 18,
   cv_i = 11 , 41 , 48 , 74 , 104 , 111 , 137 , 167 , 174 , 199 , 229 , 236, 262 ,
          292 , 299 , 325 , 355 , 362 ,
   refcrd_file = 'inpcrd'
/
&colvar
   cv_type = 'QUATERNION3'
   cv_ni = 18,
   cv_i = 11 , 41 , 48 , 74 , 104 , 111 , 137 , 167 , 174 , 199 , 229 , 236, 262 ,
          292 , 299 , 325 , 355 , 362 ,
   refcrd_file = 'inpcrd'
/
&colvar
   cv_type = 'QUATERNION0'
   cv_ni = 24,
   cv_i = 12 , 16 , 46 , 55 , 75 , 79 , 109 , 118 , 138 , 142 , 172 , 181 , 200 ,
          204 , 234 , 243 , 263 , 267 , 297 , 306 , 326 , 330 , 360 , 369 ,
   refcrd_file = 'inpcrd',
   q_index = 2
/
&colvar
   cv_type = 'QUATERNION1'
   cv_ni = 24,
   cv_i = 12 , 16 , 46 , 55 , 75 , 79 , 109 , 118 , 138 , 142 , 172 , 181 , 200 ,
          204 , 234 , 243 , 263 , 267 , 297 , 306 , 326 , 330 , 360 , 369 ,
   refcrd_file = 'inpcrd',
   q_index = 2
/
&colvar
   cv_type = 'QUATERNION2'
   cv_ni = 24,
   cv_i = 12 , 16 , 46 , 55 , 75 , 79 , 109 , 118 , 138 , 142 , 172 , 181 , 200 ,
          204 , 234 , 243 , 263 , 267 , 297 , 306 , 326 , 330 , 360 , 369 ,
   refcrd_file = 'inpcrd',
   q_index = 2
/
&colvar
   cv_type = 'QUATERNION3'
   cv_ni = 24,
   cv_i = 12 , 16 , 46 , 55 , 75 , 79 , 109 , 118 , 138 , 142 , 172 , 181 , 200 ,
          204 , 234 , 243 , 263 , 267 , 297 , 306 , 326 , 330 , 360 , 369 ,
   refcrd_file = 'inpcrd',
   q_index = 2
/
```

Figure 8.5: *An example of* `Orientation` *variable.*

```
&colvar
   cv_type = 'SPINANGLE'
   cv_ni = 18, cv_nr = 54,
   cv_i = 11 , 41 , 48 , 74 , 104 , 111 , 137 , 167 , 174 , 199 , 229 ,
          236, 262 , 292 , 299 , 325 , 355 , 362 ,
   cv_r = 0.96 , -4.47 , -0.31 , 3.48 , -3.00 , 3.06 , 0.88 , 0.01 ,
          3.36 ,4.55 , -0.51 , 6.46 , 3.93 , 2.38 , 9.81 , 0.26 ,
          0.84 , 10.12 , 1.90 , 4.16 , 13.21 , -1.06 , 4.47 , 16.58 ,
         -0.71 , 0.52 , 16.88 , -0.96 , -4.47 , 17.21 , -3.48 ,
         -3.00 , 13.84 , -0.88 , 0.01 , 13.54 , -4.55 , -0.51 , 10.44 ,
         -3.93 , 2.38 , 7.09 , -0.26 , 0.84, 6.78 , -1.90 , 4.16 ,
          3.69 , 1.06 , 4.47 , 0.32 , 0.71 , 0.52 , 0.02 ,
   axis = 1.0, 0.0, 0.0
/
```

Figure 8.6: *An example of* `SPINANGLE` *variable.*

```
title line
&cntrl
..., infe = 1
/

&smd
   output_file = 'smd.txt'
   output_freq = 50
   cv_file = 'cv.in'
/
```

```
cv_file
&colvar
  cv_type = 'DISTANCE'
  cv_ni = 2
  cv_i = 5, 9
  npath = 2, path = 5.0, 3.0, path_mode = 'LINES',
  nharm = 1, harm = 10.0
/
```

Figure 8.7: *An example* `MDIN` *file and* `CV.IN` *file for steered* `MD`*. Only the relevant part is shown.*

```
title line
&cntrl
..., infe = 1
/

&pmd
   output_file = 'pmd.txt'
   output_freq = 50
   cv_file = 'cv.in'
/
```

```
cv_file
&colvar ! first
   cv_type = 'ANGLE'
   cv_ni = 3, cv_i = 5, 9, 15
   anchor_position = -10.0,1.0,1.0,10.0
   anchor_strength = 10.0,10.0
/
&colvar ! second
   cv_type = 'TORSION'
   cv_ni = 4, cv_i = 1, 2, 3, 4
   anchor_position = -10.0,0.0,0.0,10.0
   anchor_strength = 23.8,23.8
/
```

Figure 8.8: *An example* `MDIN` *file and* `CV.IN` *file for umbrella sampling (only relevant part is presented in full).*

```
cv_file
&colvar ! phi
   cv_type = 'TORSION'
   cv_ni = 4, cv_i = 5, 7, 9, 15
   anchor_position = -2.05,-2.0,2.0,2.05
   anchor_strength = 500.0,500.0
/
&colvar ! psi
   cv_type = 'TORSION'
   cv_ni = 4, cv_i = 7, 9, 15, 17
   anchor_position = -1.85,-1.8,1.8,1.85
   anchor_strength = 500.0,500.0
/
```

Figure 8.9: *An example* `CV.IN` *file to restrain the* φ *and* ψ *of dialanine.*

```
title line
&cntrl
..., infe = 1
/

&abmd
   mode = 'FLOODING'

   monitor_file = 'abmd.txt'
   monitor_freq = 33
   cv_file = 'cv.in'

   umbrella_file = 'umbrella.nc'

   timescale = 100.0 ! in ps

   selection_freq = 10000
   selection_constant = 0.001

   wt_temperature = 10000.0
   wt_umbrella_file = 'wt_umbrella.nc'
/
```

```
cv_file
&colvar
  cv_type = 'DISTANCE'
  cv_ni = 2, cv_i = 5, 9
  cv_min = -1.0, cv_max = 10.0 ! min is not needed for DISTANCE
  resolution = 0.5 ! required for mode = FLOODING
/
```

Figure 8.10: *An example `MDIN` file and `CV.IN` file for `ABMD` (only the relevant part is presented in full).*

```
title line
&cntrl
..., infe = 1
/

&bbmd

   ! 0th replica only

   exchange_freq = 100 ! try for exchange every 100 steps

   exchange_log_file = 'bbmd.log'
   exchange_log_freq = 25

   mt19937_seed = 123455 ! random generator seed
   mt19937_file = 'mt19937.nc' ! file to store/load the PRG

   ! not specific for 0th replica

   mode = 'ANALYSIS'

   monitorfile = 'bbmd.01.txt' ! it is wise to have different
                               ! names in different replicas
   monitor_freq = 123
   cv_file = 'cv.in'
/
```

```
cv_file
&colvar
  cv_type = 'DISTANCE'
  cv_ni = 2, cv_i = 5, 9
/
```

Figure 8.11: *An example* `MDIN` *file and* `CV.IN` *file for* `&bbmd` *flavor of* `ABMD` *(only the relevant part is presented in full).*

```
title line
&cntrl
..., nstlim = 10000
..., infe = 1
/

&stsm      ! parallel case, I from 1 to 8, J from 1 to 16
  image = I
  equilibration = 980
  release = 20
  smoothing = 0.1
  report_centers = 'ALL'

  output_file = 'stsm.00I.J.txt'
  output_freq = 10
  cv_file = 'cv.I'
/
```

```
cv_file
&colvar ! phi
   cv_type = 'TORSION'
   cv_ni = 4, cv_i = 5, 7, 9, 15
   anchor_position = -3.00
   anchor_strength = 20.0
/
&colvar ! psi
   cv_type = 'TORSION'
   cv_ni = 4, cv_i = 7, 9, 15, 17
   anchor_position = 3.00
   anchor_strength = 20.0
/
```

Figure 8.12: *An example* `MDIN` *file and* `CV.IN` *file for* `&stsm` *in parallel case (only the relevant part is presented in full)*

```
title line
&cntrl
..., nstlim = 160000
..., infe = 1
/

&stsm       ! sequential case, I from 1 to 8
  image = I
  repeats = 16
  equilibration = 980
  release = 20
  smoothing = 0.1
  report_centers = 'ALL'

  output_file = 'stsm.00I.txt'
  output_freq = 10
  cv_file = 'cv.I'
/
```

Figure 8.13: *An example `MDIN` file for `&stsm` in sequential case (only the relevant part is presented in full)*

```
NFE : #   restoring restraint:
NFE : #   << colvar(1) = -3.000000 >>
NFE : #   << colvar(2) = 3.000000 >>
NFE : #   equilibration begins...
......
NFE : #   << colvar(1) = -2.500688 -2.586429 >>
NFE : #   << colvar(2) = 2.782725 3.082205 >>
NFE : #  drifted center of image 1 :           8      -2.54041796       2.70644813
NFE : #  drifted center of image 2 :           8      -2.54963153       2.71715138
......
NFE : #  drifted center of image 8 :           8       1.02191205       0.16837852
NFE : #  smoothed center of image 1 :           8      -2.54041796       2.70644813
NFE : #  smoothed center of image 2 :           8      -2.60416697       2.75924174
......
NFE : #  smoothed center of image 8 :           8       1.02191205       0.16837852
NFE : #  reparametrized center of image 1 :           8      -2.54041796
2.70644813
NFE : #  reparametrized center of image 2 :           8      -2.06027108
2.47738701
......
NFE : #  reparametrized center of image 8 :           8       1.02191205
0.16837852
```

Figure 8.14: *An example of `MDOUT` file for STSM run (only part is presented, and some centers are also omitted)*

9 NMR refinement

We find the *sander* module to be a flexible way of incorporating a variety of restraints into a optimization procedure that includes energy minimization and dynamical simulated annealing. The "standard" sorts of NMR restraints, derived from NOE and J-coupling data, can be entered in a way very similar to that of programs like DISGEO, DIANA or X-PLOR; an aliasing syntax allows for definitions of pseudo-atoms, connections with peak numbers in spectra, and the use of "ambiguous" constraints from incompletely-assigned spectra. More "advanced" features include the use of time-averaged constraints, use of multiple copies (LES) in conjunction with NMR refinement, and direct refinement against NOESY intensities, paramagnetic and diamagnetic chemical shifts, or residual dipolar couplings. In addition, a key strength of the program is its ability to carry out the refinements (usually near the final stages) using an explicit-solvent representation that incorporates force fields and simulation protocols that are known to give pretty accurate results in many cases for unconstrained simulations; this ability should improve predictions in regions of low constraint density and should help reduce the number of places where the force field and the NMR constraints are in "competition" with one another.

Since there is no generally-accepted "recipe" for obtaining solution structures from NMR data, the comments below are intended to provide a guide to some commonly-used procedures. Generally speaking, the programs that need to be run to obtain NMR structures can be divided into three parts:

1. *front-end* modules, which interact with NMR databases that provide information about assignments, chemical shifts, coupling constants, NOESY intensities, and so on. We have tried to make the general format of the input straightforward enough so that it could be interfaced to a variety of programs. We generally use the FELIX and NMRView codes, but the principles should be similar for other ways of keeping track of a database of NMR spectral information. As the flow-chart in Section 9.7 indicates, there are only a few files that need to be created for NMR restraints; these are indicated by the solid rectangles. The primary distance and torsion angle files have a fairly simple format that is largely compatible with the DIANA programs; if one wishes to use information from ambiguous or overlapped peaks, there is an additional "MAP" file that makes a translation from peak identifiers to ambiguous (or partial) assignments. Finally, there are some specialized (but still pretty straightforward) file formats for chemical shift or residual dipolar coupling restraints.

   There are a variety of tools, besides the ones described below, that can assist in preparing input for structure refinement in Amber.

   - The SANE (Structure Assisted NOE Evaluation) package, *https://ambermd.org/sane.zip*, is widely used at The Scripps Research Institute.[193]
   - If you use Bruce Johnson's NmrView package, you might also want to look at the additions to that: *http://garbanzo.scripps.edu/nmrgrp/wisdom/pipe/tips_scripts.html*. In particular, the *xpkTOupl* and *starTOupl* scripts there convert NmrView peak lists into the "7-column" needed for input to makeDIST_RST.
   - Users of the MARDIGRAS programs from UCSF can use the *mardi2amber* program to do conversion to Amber format: *http://picasso.ucsf.edu/mardihome.html*

2. *restrained molecular dynamics,* which is at the heart of the conformational searching procedures. This is the part that *sander* itself handles.

3. *back-end* routines that do things like compare families of structures, generate statistics, simulate spectra, and the like. For many purposes, such as visualization, or the running of procheck-NMR, the "interface" to such programs is just the set of PDB files that contain the family of structures to be analyzed. These general-purpose structure analysis programs are available in many locations

and are not discussed here. The principal *sander*-specific tool is *sviol*, which prepares tables and statistics of energies, restraint violations, and the like.

9.1 Distance, angle and torsional restraints

Distance, angle, and other restraints are read from the DISANG file if *nmropt* > 0. Namelist rst ("&rst") contains the following variables; it is read repeatedly until a namelist &rst statement is found with IAT(1)=0, or until reaching the end of the DISANG file.

[In many cases, the user will not prepare this section of the input by hand, but will use the auxiliary programs *makeDIST_RST, makeANG_RST* and *makeCHIR_RST* to prepare input from simpler files. See also the programs *cyanarest_to_amberRST* and *nef_to_RST* if you have restraints in Cyana or NEF (NMR Exchange Format) formats.]

9.1.1 Variables in the &rst namelist:

`iat(1)→iat(8)`

- *If IRESID = 0 (normal operation):* The atoms defining the restraint. Type of restraint is determined (in order) by:

  1. If IAT(3) = 0, this is a distance restraint.
  2. If IAT(4) = 0, this is an angle restraint.
  3. If IAT(5) = 0, this is a torsional (or J-coupling, if desired) restraint or a genereralized distance restraint of 4 atoms, a type of restraint new as of Amber 10 (*sander* only, see below).
  4. If IAT(6) = 0, this is a plane-point angle restraint, a second restraint new as of Amber 10 (*sander* only). The angle is measured between the normal of a plane defined by IAT(1)..IAT(4) and the vector from the center of mass of atoms IAT(1)..IAT(4) to the position of IAT(5). The normal is defined by (r1 - r2) × (r3 - r4), where rn is the position of IAT(n).
  5. If IAT(7) = 0, this is a generalized distance restraint of 6 atoms (see below).
  6. Otherwise, if IAT(1)..IAT(8) are all nonzero, this is a plane- plane angle restraint, a third new restraint type as of Amber 10 (*sander* only, or a generalized distance restraint of 8 atoms (see below). For the plane-plane restraint, the angle is measured between the two normals of the two planes, which are defined by (r1 - r2) × (r3 - r4) and (r5 - r6) × (r7 - r8). In the case of either planar restraint, the plane may be defined using three atoms instead of four simply by using one atom twice.

  If any of IAT(n) are < 0, then a corresponding group of atoms is defined below, and the coordinate-averaged position of this group will be used in place of atom IAT(n). A new feature as of Amber 10, atom groups may be used not only in distance restraints, but also in angle, torsion, the new plane restraints, or the new generalized restraints. If this is a distance restraint, and IAT1 <0, then a group of atoms is defined below, and the coordinate-averaged position of this group will be used in place of the coordinates of atom 1 [IAT(1)]. Similarly, if IAT(2) < 0, a group of atoms will be defined below whose coordinate-averaged position will be used in place of the coordinates for atom 2 [IAT(2)].

- *If IRESID=1:* IAT(1)..IAT(8) point to the *residues* containing the atoms comprising the internal. Residue numbers are the absolute in the entire system. In this case, the variables ATNAM(1)..ATNAM(8) must be specified and give the character names of the atoms within the respective residues. If any of IAT(n) are less than zero, then group input will still be read in place of the corresponding atom, as described below.

- *Defaults for IAT(1)→IAT(8) are 0.*

`rstwt(1)`→`rstwt(4)` New as of Amber 10 (*sander* only), users may now define a single restraint that is a function of multiple distance restraints, called a "generalized distance coordinate" restraint. The energy of such a restraint has the following form:

$$U = k(w_1|\mathbf{r}_1 - \mathbf{r}_2| + w_2|\mathbf{r}_3 - \mathbf{r}_4| + w_3|\mathbf{r}_5 - \mathbf{r}_6| + w_4|\mathbf{r}_7 - \mathbf{r}_8| - r_0)^2$$

where the weights $w_\mathrm{n}$are given in rstwt(1)..rstwt(4) and the positions $\mathbf{r}_\mathrm{n}$are the positions of the atoms in iat(1)..iat(8).

Generalized distance coordinate restraints must be defined with either 4, 6, or 8 atoms and 2, 3, or 4 corresponding nonzero weights in rstwt(1)..rstwt(4). Weights may be any positive or negative real number.

If all the weights in rstwt(1)..rstwt(4) are zero and four atoms are given in iat(1)..iat(4) for the restraint, the restraint is a torsional or J-coupling restraint. If eight atoms are given in iat(1)..iat(8) and all weights are zero, the restraint is a plane-plane angle restraint. However, if the weights are nonzero, the restraint will be a generalized distance coordinate restraint.

Default for rstwt(1)..rstwt(4) is 0.0

`restraint` New as of Amber 10 (*sander* only), users may now use a "natural language" system to define restraints by using the RESTRAINT character variable. Valid restraints defined in this manner will begin with a "distance( )" "angle( )" "torsion( )" or "coordinate( )" keyword. Within the parentheses, the atoms that make up the restraint are specified. Atoms may be defined either with an explicit atom number or by using ambmask format, namely :(residue#)@(atom name). Atoms may be separated by commas, spaces, or parentheses. Additionally, negative integers may be used if atom groupings are defined in other variables in the namelist as described below. In addition to the principle distance, angle, torsion, and coordinate keywords, Some keywords may be used within the principle keywords to define more complicated restraints. The keyword "plane( )" may be used once or twice within the parentheses of the "angle( ) keyword to define a planar restraint. Defining one plane grouping plus one other atom in this manner will create a plane-point angle restraint as described above. Defining two plane groupings will create a plane-plane angle restraint. The keyword "plane( )" may only be used inside of "angle( )," and is necessary to define either a plane-point or plane-plane restraint.
Within the "coordinate( )" keyword, the user must use 2 to 4 "distance( )" keywords to define a generalized distance coordinate restraint. The "distance( )" keyword functions just like it does when used to define a traditional distance restraint. The user may specify any two atom numbers, masks, or negative numbers corresponding to atom groups defined outside of RESTRAINT. Additionally, following each "distance( )" keyword inside "coordinate( )" the user must specify a real-number weight to be applied to each distance making up the generalized coordinate.
The "com( )" keyword may be used within any other keyword to define a center of mass grouping of atoms. Within the parenthesis, the user will enter a list of atom numbers or masks. Negative numbers, which correspond to externally-defined groups, may not be used.
Any type of parenthetical character, i.e., ( ), [ ], or { }, may be used wherever parentheses have been used above.

The following are all examples of valid restraint definitions:

```
restraint = "distance( (45) (49) )"
= "angle (:21@C5' :21@C4' 108)"
= "torsion[-1,-1,-1, com(67, 68, 69)]"
= "angle( -1, plane(81, 85, 87, 90) )"
= "angle(plane(com(9,10),:5@CA,31,32),plane(14,15,15,16))"
= "coordinate(distance(:5@C3',:6@O5'),-1.0,distance(134,-1),1.0)"
```

There is a 256 character limit on RESTRAINT, so if a particularly large atom grouping is desired, it is necessary to specify a negative number instead of "com( )" and define the group as described below. RESTRAINT will only be parsed if IAT(1) = 0, otherwise the information in IAT(1) .. IAT(8) will define the restraint. *Default for restraint is ' '.*

`atnam` If IRESID = 1, then the character names of the atoms defining the internal are contained in ATNAM(1)→ATNAM(8). Residue IAT(1) is searched for atom name ATNAM(1); residue IAT(2) is searched for atom name ATNAM(2); etc. *Defaults for ATNAM(1)→ATNAM(8) are ' '.*

`iresid` Indicates whether IAT(I) points to an atom # or a residue #. See descriptions of IAT() and ATNAM() above. If RESTRAINT is used to define the internal instead of IAT(), IRESID has no effect on how RESTRAINT is parsed. However, it will affect the behavior of atom group definitions as described below if negative numbers are specified within RESTRAINT. *Default = 0.*

`nstep1, nstep2` This restraint is applied for steps/iterations NSTEP1 through NSTEP2. If NSTEP2 = 0, the restraint will be applied from NSTEP1 through the end of the run. Note that the first step/iteration is considered step zero (0). *Defaults for NSTEP1, NSTEP2 are both 0.*

`irstyp` Normally, the restraint target values defined below (R1→R4) are used directly. If IRSTYP = 1, the values given for R1→R4 define relative displacements from the current value (value determined from the starting coordinates) of the restrained internal. For example, if IRSTYP=1, the current value of a restrained distance is 1.25, and R1 (below) is -0.20, then a value of R1=1.05 will be used. *Default is IRSTYP=0.*

`ialtd` Determines what happens when a distance restraint gets very large. If IALTD=1, then the potential "flattens out", and there is no force for large violations; this allows for errors in constraint lists, but might tend to ignore constraints that *should* be included to pull a bad initial structure towards a more correct one. When IALTD=0 the penalty energy continues to rise for large violations. See below for the detailed functional forms that are used for distance restraints. Set IALTD=0 to recover the behavior of earlier versions of *sander*. Default value is 0, or the last value that was explicitly set in a previous restraint. This value is set to 1 if *makeDIST_RST* is called with the *-altdis* flag.

`ifvari` If IFVARI $>$ 0, then the force constants/positions of the restraint will vary with step number. Otherwise, they are constant throughout the run. If IFVARI $>$0, then the values R1A→R4A, RK2A, and RK3A must be specified (see below). *Default is IFVARI=0.*

`ninc` If IFVARI $>$ and NINC $>$ 0, then the change in the target values of of R1→R4 and K2,K3 is applied as a step function, with NINC steps/ iterations between each change in the target values. If NINC = 0, the change is effected continuously (at every step). *Default for NINC is the value assigned to NINC in the most recent namelist where NINC was specified. If NINC has not been specified in any namelist, it defaults to 0.*

`imult` If IMULT=0, and the values of force constants RK2 and RK3 are changing with step number, then the changes in the force constants will be linearly interpolated from rk2→rk2a and rk3→rk3a as the step number changes. If IMULT=1 and the force constants are changing with step number, then the changes in the force constants will be effected by a series of multiplicative scalings, using a single factor, R, for all scalings. *i.e.*

```
rk2a = R**INCREMENTS * rk2
rk3a = R**INCREMENTS * rk3.
```

INCREMENTS is the number of times the target value changes, which is determined by NSTEP1, NSTEP2, and NINC. *Default for IMULT is the value assigned to IMULT in the most recent namelist where IMULT was specified. If IMULT has not been specified in any namelist, it defaults to 0.*

`r1→r4, rk2, rk3, r1a→r4a, rk2a, rk3a` If IALTD=0, the restraint is a well with a square bottom with parabolic sides out to a defined distance, and then linear sides beyond that. If R is the value of the restraint in question:

- R < r1 Linear, with the slope of the "left-hand" parabola at the point R=r1.
- r1 <= R < r2 Parabolic, with restraint energy $k_2(R - r_2)^2$.
- r2 <= R < r3 E = 0.
- r3 <= R < r4 Parabolic, with restraint energy $k_3(R - r_3)^2$.
- r4 <= R Linear, with the slope of the "right-hand" parabola at the point R=r4.

For torsional restraints, the value of the torsion is translated by +-n*360, if necessary, so that it falls closest to the mean of r2 and r3. Specified distances are in Angstroms. Specified angles are in degrees. Force constants for distances are in kcal/mol-Å2 Force constants for angles are in kcal/mol-rad 2 . (Note that angle positions are specified in degrees, but force constants are in radians, consistent with typical reporting procedures in the literature).

If IALTD=1, distance restraints are interpreted in a slightly different fashion. Again, If R is the value of the restraint in question:

- R < r2 Parabolic, with restraint energy $k_2(R - r_2)^2$.
- r2 <= R < r3 E = 0.
- r3 <= R < r4 Parabolic, with restraint energy $k_3(R - r_3)^2$.
- r4 <= R Hyperbolic, with energy $k_3[b/(R - r_3) + a]$, where $a = 3(r_4 - r_3)^2$ and $b = -2(r_4 - r_3)^3$. This function matches smoothly to the parabola at $R = r_4$, and tends to an asymptote of ak_3at large R. The functional form is adapted from that suggested by Michael Nilges, *Prot. Eng.* **2,** 27-38 (1988). Note that if *ialtd=1*, the value of r1 is ignored.

`ifvari`

= 0 The values of r1→r4, rk2, and rk3 will remain constant throughout the run.

> 0 The values r1a, r2a, r3a, r4a, r2ka and r3ka are also used. These variables are defined as for r1→r4 and rk2, rk3, but correspond to the values appropriate for NSTEP = NSTEP2: e.g., if IVARI >0, then the value of r1 will vary between NSTEP1 and NSTEP2, so that, e.g. r1(NSTEP1) = r1 and r1(NSTEP2) = r1a. Note that you *must* specify an explicit value for *nstep1* and *nstep2* if you use this option. *Defaults for r1→r4,rk2,rk3,r1a→r4a,rk2a and rk3a are the values assigned to them in the most recent namelist where they were specified. They should always be specified in the first* &rst *namelist.*

`r0, k0, r0a, k0a` New as of Amber 10 (*sander* only), the user may more easily specify a large parabolic well if desired by using R0 and K0, and then R0A and K0A if IFVARI > 0. The parabolic well will have its zero at R = R0 and a force constant of K0. These variables simply map the disired parabolic well into r1→r4, rk2, rk3, r1a→r4a, rk2a, and rk3a in the following manner:

- R1 = 0 for distance, angle, and planar restraints, R1 = R0 - 180 for torsion restraints
- R1A = 0 for distance, angle, and planar restraints, R1A = R0A - 180 for torsion restraints
- R2 = R0; R3 = R0

- R2A = R0A; R3A = R0A
- R4 = R0 + 500 for distance restraints, R4 = 180 for angle and planar restraints, R4 = R0 + 180 for torsion restraints
- RK2 = K0; RK3 = K0
- RK2A = K0A; RK3A = K0A

`rjcoef(1)`→`rjcoef(3)` By default, 4-atom sequences specify torsional restraints. It is also possible to impose restraints on the vicinal 3 J-coupling value related to the underlying torsion. J is related to the torsion τ by the approximate Karplus relationship: $J = A\cos^2(\tau) + B\cos(\tau) + C$. If you specify a nonzero value for either RJCOEF(1) or RJCOEF(2), then a J-coupling restraint, rather than a torsional restraint, will be imposed. At every MD step, J will be calculated from the Karplus relationship with A = RJCOEF(1), B = RJCOEF(2) and C = RJCOEF(3). In this case, the target values (R1->R4, R1A->R4A) and force constants (RK2, RK3, RK2A, RK3A) refer to J-values for this restraint. RJCOEF(1)->RJCOEF(3) must be set individually for each torsion for which you wish to apply a J-coupling restraint, and RJCOEF(1)->RJCOEF(3) may be different for each J-coupling restraint. With respect to other options and reporting, J-coupling restraints are treated identically to torsional restraints. This means that if time-averaging is requested for torsional restraints, it will apply to J-coupling restraints as well. The J-coupling restraint contribution to the energy is included in the "torsional" total. And changes in the relative weights of the torsional force constants also change the relative weights of the J-coupling restraint terms. Setting RJCOEF has no effect for distance and angle restraints. *Defaults for RJCOEF(1)->RJCOEF(3) are 0.0.*

igr1(i),i=1→200, igr2(i),i=1→200, ... igr8(i),i..1=1→200 If IAT(n) < 0, then IGRn() gives the atoms defining the group whose coordinate averaged position is used to define "atom n" in a restraint. Alternatively, if RESTRAINT is used to define the internal, then if the nth atom specified is a number less than zero, IGRn() gives the atoms defining the group whose coordinate averaged position is used to define "atom n" in a restraint. If IRESID = 0, absolute atom numbers are specified by the elements of IGRn(). If IRESID = 1, then IGRn(I) specifies the number of the residue containing atom I, and the name of atom I must be specified using GRNAMn(I). A maximum of 200 atoms (N # of atoms if using pmemd) are allowed in any group. Only specify those atoms that are needed. Default value for any unspecified element of IGRn(i) is 0.

`fxyz` If iat(3)=0 and igr1 and/or igr2 is defined then it is possible to weight the x, y, z components of the force in the restraint to 0 (no force) or 1 (full restraint force). Ex: fxyz=0, 0, 1. This sets no additional restraint force on the x component or y-component of the restraint force, and full z-component restraint force. Default fxyz=1,1,1. Note: When setting fxyz, the r1, r2, r3, r4 values should be set relative to a weighted distance $\sqrt{(w_x * d_x)^2 + (w_y * d_y)^2 + (w_z * d_z)^2}$, so if fxyz=0,0,1 then the only distance taken into account when comparing to r1,r2,r3,r4 is the z distance between the molecule and the center of mass. Note that the DUMPAVE value when outxyz=0 is also just the weighted distance.

`outxyz` If iat(3)=0 and igr1 and/or igr2 is defined then it is possible to output the x, y, z components of the force in the restraint if outxyz is set to 1. Default outyz=0. When outxyz is set to 1, the components of the distance and total distance are outputed in DUMPAVE in the order of the x-component, y-component, z-component, total distance.

`grnam1(i),i=1→200,` grnam2(i),i=1→200, ... grnam8(i),i=1→200 If group input is being specified (IGRn(1) > 0), *and* IRESID = 1, then the character names of the atoms defining the group are contained in GRNAMn(i), as described above. In the case IAT(1) < 0, each residue IGR1(i) is searched for an atom name GRNAM1(i) and added to the first group list. In the case IAT(2) < 0, each residue IGR2(i) is searched for an atom name GRNAM2(i) and added to the second group list. *Defaults for GRNAMn(i) are ' '.*

`ir6` If a group coordinate-averaged position is being used (see IGR1 and IGR2 above), the average position can be calculated in either of two manners: If IR6 = 0, center-of-mass averaging will be used. If IR6=1, the $\langle r^{-6}\rangle^{-1/6}$ average of all interaction distances to atoms of the group will be used. *Default for IR6 is the value assigned to IR6 in the most recent namelist where IR6 was specified. If IR6 has not been specified in any namelist, it defaults to 0.*

`ifntyp` If time-averaged restraints have been requested (see DISAVE/ANGAVE/TORAVE above), they are, by default, applied to all restraints of the class specified. Time-averaging can be overridden for specific internals of that class by setting IFNTYP for that internal to 1. IFNTYP has no effect if time-averaged restraint are not being used. *Default value is IFNTYP=0.*

`ixpk, nxpk` These are user-defined integers than can be set for each constraint. They are typically the "peak number" and "spectrum number" associated with the cross-peak that led to this particular distance restraint. Nothing is ever done with them except to print them out in the "violation summaries", so that NMR people can more easily go from a constraint violation to the corresponding peak in their spectral database. Default values are zero.

`iconstr` If *iconstr* > 0, (default is 0) a Lagrangian multiplier is also applied to the two-center internal coordinate defined by IAT(1) and IAT(2). The effect of this Lagrangian multiplier is to maintain the initial orientation of the internal coordinate. The rotation of the vector IAT(1)->IAT(2) is prohibited, though translation is allowed. For each defined two-center internal coordinate, a separate Lagrangian multiplier is used. Therefore, although one can use as many multipliers as needed, defining centers should NOT appear in more than one multiplier. This option is compatible with mass centers (i.e., negative IAT(1) or IAT(2)). ICONSTR can be used together with harmonic restraints. RK2 and RK3 should be set to 0.0 if the two-center internal coordinate is a simple Lagrangian multiplier. An example has been included in $AMBERHOME/example/lagmul.

Namelist &rst is read for each restraint. Restraint input ends when a namelist statement with iat(1) = 0 (or iat(1) not specified) is found. Note that comments can precede or follow any namelist statement, allowing comments and restraint definitions to be freely mixed.

9.2 NOESY volume restraints

After the previous section, NOESY volume restraints may be read. This data described in this section is only read if NMROPT = 2. The molecule may be broken in overlapping submolecules, in order to reduce time and space requirements. Input *for each submolecule* consists of namelist "&noeexp", followed *immediately* by standard Amber "group" cards defining the atoms in the submolecule. In addition to the submolecule input ("&noeexp"), you may also need to specify some additional variables in the cntrl namelist; see the "NMR variables" description in that section.

In many cases, the user will not prepare this section of the input by hand, but will use the auxiliary program *makeDIST_RST* to prepare input from simpler files.

Variables in the &noeexp namelist:

For each submolecule, the namelist "&noeexp" is read (either from *stdin* or from the NOESY redirection file) which contains the following variables. There are no effective defaults for *npeak, emix, ihp, jhp,* and *aexp*: you must specify these.

`npeak(imix)` Number of peaks for each of the "imix" mixing times; if the last mixing time is *mxmix*, set NPEAK(*mxmix+1*) = -1. End the input when NPEAK(1) < 0.

`emix` *(imix)* Mixing times (in seconds) for each mixing time.

`ihp` *(imix,ipeak)*, `jhp` *(imix,ipeak)* Atom numbers for the atoms involved in cross-peak "ipeak" at mixing time "imix"

`aexp(imix,ipeak)` Experimental target integrated intensity for this cross peak. If AEXP is negative, this cross peak is part of a set of overlapped peaks. The computed intensity is added to the peak that follows; the next time a peak with AEXP > 0 is encountered, the running sum for the calculated peaks will be compared to the value of AEXP for that last peak in the list. In other words, a set of overlapped peaks is represented by one or more peaks with AEXP < 0 followed by a peak with AEXP > 0. The computed total intensity for these peaks will be compared to the value of AEXP for the final peak.

`arange` *(imix,ipeak)* "Uncertainty" range for this peak: if the calculated value is within $\pm$ARANGE of AEXP, then no penalty will be assessed. Default uncertainties are all zero.

`awt` *(imix,ipeak)* Relative weight for this cross peak. Note that this will be multiplied by the overall weight given by the NOESY weight change cards in the weight changes section (Section 1). Default values are 1.0, unless INVWT1,INVWT2 are set (see below), in which case the input values of AWT are ignored.

`invwt1,invwt2` Lower and upper bounds on the weights for the peaks respectively, such that the relative weight for each peak is 1/intensity if 1/intensity lies between the lower and upper bounds. This is the intensity after being scaled by *oscale*. The inverse weighing scheme adopted by this option prevents placing too much influence on the strong peaks at the expense of weaker peaks and was previously invoked using the compilation flag "INVWGT". Default values are INVWT1=INVWT2=1.0, placing equal weights on all peaks.

`omega` Spectrometer frequency, in Mhz. Default is 500. It is possible for different sub-molecules to have different frequencies, but omega will only change when it is explicitly re-set. Hence, if all of your data is at 600 Mhz, you need only set *omega* to 600. in the first submolecule.

`taurot` Rotational tumbling time of the molecule, in nsec. Default is 1.0 nsec. Like *omega*, this value is "sticky", so that a value set in one submolecule will remain until it is explicitly reset.

`taumet` Correlation time for methyl jump motion, in ns. This is only used in computing the intra-methyl contribution to the rate matrix. The ideas of Woessner are used, specifically as recommended by Kalk & Berendsen.[194] Default is 0.0001 ns, which is effectively the fast motion limit. The default is consistent with the way the rest of the rate matrix elements are determined (also in the fast motion limit,) but probably is not the best value to use, since methyl groups appear to have T1 values that are systematically shorter than other protons, and this is likely to arise from the fact that the methyl correlation time can be near to the inverse of the spectrometer frequency. A value of 0.02 - 0.05 ns is probably better than 0.0001, but this is still an active research area, and you are on your own here, and should consult the literature for further discussion.[195] As with *omega*, *taumet* can be different for different sub-molecules, but will only change when it is explicitly re-set.

`id2o` Flag for determining if exchangeable protons are to be included in the spin-diffusion calculation. If ID2O=0 (default) then all protons are included. If ID2O=1, then all protons bonded to nitrogen or oxygen are assumed to not be present for the purposes of computing the relaxation matrix. No other options exist at present, but they could easily be added to the subroutine *indexn*. Alternatively, you can manually rename hydrogens in the *prmtop* file so that they do not begin with "H": such protons will not be included in the relaxation matrix. (*Note:* for technical reasons, the HOH proton of tyrosine must always be present, so setting ID2O=1 will not remove it; we hope that this limitation will be of minor importance to most users.) The *id2o* variable retains its value across namelist reads, *i.e.* its value will only change if it is explicitly reset.

`oscale`	overall scaling factor between experimental and computed volume units. The experimental intensities are multiplied by *oscale* before being compared to calculated intensities. This means that the weights WNOESY and AWT always refer to "theoretical" intensity scales rather than to the (arbitrary) experimental units. The *oscale* variable retains its value across namelist reads, *i.e.* its value will only change if it is explicitly reset. The initial (default) value is 1.0.

The atom numbers *ihp* and *jhp* are the absolute atom numbers. For methyl groups, use the number of the last proton of the group; for the delta and epsilon protons of aromatic rings, use the delta-2 or epsilon-2 atom numbers. Since this input requires you to know the absolute atom numbers assigned by Amber to each of the protons, you may wish to use the separate *makeDIST_RST* program which provides a facility for more turning human-readable input into the required file for *sander*.

Following the &noeexp namelist, give the Amber "group" cards that identify this submolecule. This combination of "&noeexp" and "group" cards can be repeated as often as needed for many submolecules, subject to the limits described in the *nmr.h* file. As mentioned above, this input section ends when NPEAK(1) < 0, or when and end-of-file is reached.

9.3 Chemical shift restraints

After reading NOESY restraints above (if any), read the chemical shift restraints in namelist *&shf*, or the pseudocontact restraints in namelist *&pcshift*. Reading this input is triggered by the presence of a SHIFTS line in the I/O redirection section. In many cases, the user will not prepare this section of the input by hand, but will use the auxiliary programs *shifts* or *fantasian* to prepare input from simpler files.

Variables in the &shf namelist.

(Defaults are only available for *shrang*, *wt*, *nter*, and *shcut*; you must specify the rest.)

`nring`	Number of rings in the system.
`natr`(i)	Number of atoms in the *i-th* ring.
`iatr`(j,i)	Absolute atom number for the *j-th* atom of the *i-th* ring.
`namr`(i)	Eight-character string that labels the *i-th* ring. The first three characters give the residue name (in caps); the next three characters contain the residue number (right justified); column 7 is blank; column 8 may optionally contain an extra letter to distinguish the two rings of trp, or the 5 or 8 rings of the heme group.
`str`(i)	Ring current intensity factor for the *i-th* ring. Older values are summarized by Cross and Wright;[196] more recent empirical parametrizations seem to give improved results.[197, 198]
`nprot`	Number of protons for which penalty functions are to be set up.
`iprot`(i)	Absolute atom number of the *i-th* proton whose shifts are to be evaluated. For equivalent protons, such as methyl groups or rapidly flipping phenylalanine rings, enter all two or three atom numbers in sequence; averaging will be controlled by the *wt* parameter, described below.

`obs`(i)	Observed secondary shift for the *i-th* proton. This is typically calculated as the observed value minus a random coil reference value.
`shrang`(i)	"Uncertainty" range for the observed shift: if the calculated shift is within ±SHRANG of the observed shift, then no penalty will be imposed. The default value is zero for all shifts.
`wt`(i)	Weight to be assigned to this penalty function. Note that this value will be multiplied by the overall weight (if any) given by the SHIFTS command in the assignment of weights (above). Default values are 1.0. For sets of equivalent protons, give a negative weight for all but the last proton in the group; the last proton gets a normal, positive value. The average computed shift of the group will be compared to *obs* entered for the last proton.
`shcut`	Values of calculated shifts will be printed only if the absolute error between calculated and observed shifts is greater than this value. *Default = 0.3 ppm.*
`nter`	Residue number of the N-terminus, for protein shift calculations; *default = 1.*
`cter`	Residue number of the C-terminus, for protein shift calculations. Believe it or not, the current code cannot figure this out for itself.

In typical usage, the *shifts* program (`http://casegroup.rutgers.edu/shifts.html`) would be used to create this file, with a typical command line:

```
shifts -readobs -sander '::H*' gcg10
```

Sample input and output files are in `$AMBERHOME/test/rdc`.

9.4 Pseudocontact shift restraints

The PCSHIFT module allows the inclusion of pseudocontact shifts as constraints in energy minimization and molecular dynamics calculations on paramagnetic molecules. The pseudocontact shift depends on the magnetic susceptibility anisotropy of the metal ion and on the location of the resonating nucleus with respect to the axes of the magnetic susceptibility tensor. For the nucleus i, it is given by:

$$\delta^i_{pc} = \sum_j \frac{1}{12\pi r^3_{ij}} \left[\Delta\chi^j_{ax}(3n^2_{ij} - 1) + (3/2)\Delta\chi^j_{rh}(l^2_{ij} - m^2_{ij}) \right]$$

where l_{ij}, m_{ij}, and n_{ij} are the direction cosines of the position vector of atom i with respect to the j-th magnetic susceptibility tensor coordinate system, r_{ij} is the distance between the j-th paramagnetic center and the proton i, $\Delta\chi_{ax}$ and $\Delta\chi_{rh}$ are the axial and the equatorial (rhombic) anisotropies of the magnetic susceptibility tensor of the j-th paramagnetic center. For a discussion, see Ref. [199].

The PCSHIFT module to be used needs a namelist file which includes information on the magnetic susceptibility tensor and on the paramagnetic center, and a line of information for each nucleus. This module allows to include more than one paramagnetic center in the calculations. To include pseudocontact shifts as constraints in energy minimization and molecular dynamics calculations the NMROPT flag should be set to 2, and a *PCSHIFT=filename* statement entered in the I/O redirection section.

To perform molecular dynamics calculations it is necessary to eliminate the rotational and translational degree of freedom about the center of mass (this because during molecular dynamics calculations the relative orientation between the external reference coordinate system and the magnetic anisotropy tensor coordinate system has to be fixed).This option can be obtained with the NSCM flag of *sander*.

Variables in the pcshift namelist

`nprot` number of pseudocontact shift constraints.

`nme` number of paramagnetic centers.

`nmpmc` name of the paramagnetic atom

`optphi(n), opttet(n), optomg(n), opta1(n), opta2(n)` the five parameters of the magnetic anisotropy tensor for each paramagnetic center.

`optkon` force constant for the pseudocontact shift constraints

Following this, there is a line for each nucleus for which the pseudocontact shift information is given has to be added. Each line contains :

`iprot(i)` atom number of the i-th proton whose shift is to be used as constraint.

`obs(i)` observed pseudocontact shift value, in ppm

`wt(i)` relative weight

`tolpro(i)` relative tolerance ix mltpro

`mltpro(i)` multiplicity of the NMR signal (for example the protons of a methyl group have mltprot(i)=3)

Example

Here is a &pcshf namelist example: a molecule with three paramagnetic centers and 205 pseudocontact shift constraints.

```
&pcshf
nprot=205,
nme=3,
nmpmc='FE ',
optphi(1)=-0.315416,
opttet(1)=0.407499,
optomg(1)=0.0251676,
opta1(1)=-71.233,
opta2(1)=1214.511,
optphi(2)=0.567127,
opttet(2)=-0.750526,
optomg(2)=0.355576,
opta1(2)=-60.390,
opta2(2)=377.459,
optphi(3)=0.451203,
opttet(3)=-0.0113097,
optomg(3)=0.334824,
opta1(3)=-8.657,
opta2(3)=704.786,
optkon=30,
iprot(1)=26, obs(1)=1.140, wt(1)=1.000, tolpro(1)=1.00, mltpro(1)=1,
iprot(2)=28, obs(2)=2.740, wt(2)=1.000, tolpro(2)=.500, mltpro(2)=1,
iprot(3)=30, obs(3)=1.170, wt(3)=1.000, tolpro(3)=.500, mltpro(3)=1,
iprot(4)=32, obs(4)=1.060, wt(4)=1.000, tolpro(4)=.500, mltpro(4)=3,
iprot(5)=33, obs(5)=1.060, wt(5)=1.000, tolpro(5)=.500, mltpro(5)=3,
iprot(6)=34, obs(6)=1.060, wt(6)=1.000, tolpro(6)=.500, mltpro(6)=3,
...
```

```
...
iprot(205)=1215, obs(205)=.730, wt(205)=1.000, tolpro(205)=.500,
mltpro(205)=1,
/
```

An *mdin* file that might go along with this, to perform a maximum of 5000 minimization cycles, starting with 500 cycles of steepest descent. PCSHIFT=./pcs.in redirects the input from the namelist "pcs.in" which contains the pseudocontact shift information.

```
Example of minimization including pseudocontact shift constraints
&cntrl
ibelly=0,imin=1,ntpr=100,
ntr=0,maxcyc=500,
ncyc=50,ntmin=1,dx0=0.0001,
drms=.1,cut=10.,
nmropt=2,pencut=0.1, ipnlty=2,
/
&wt type='REST', istep1=0,istep2=1,value1=0.,
value2=1.0, /
&wt type='END' /
DISANG=./noe.in
PCSHIFT=./pcs.in
LISTOUT=POUT
```

9.5 Direct dipolar coupling restraints

Energy restraints based on direct dipolar coupling constants are entered in this section. All variables are in the namelist &align; reading of this section is triggered by the presence of a DIPOLE line in the I/O redirection section.

When dipolar coupling restraints are turned on, the five unique elements of the alignment tensor are treated as additional variables, and are optimized along with the structural parameters. Their effective masses are determined by the *scalm* parameter entered in the &cntrl namelist. Unlike some other programs, the variables used are the Cartesian components of the alignment tensor in the axis system defined by the molecule itself: *e.g.* $S_{mn} \equiv \langle (3\cos\theta_m \cos\theta_n - \delta_{mn})/2 \rangle$, where $m, n = x, y, z$, and θ_x is the angle between the x axis and the spectrometer field.[200] The factor of 10^5 is just to make the values commensurate with atomic coordinates, since both the coordinates and the alignment tensor values will be updated during the refinement. The calculated dipolar splitting is then

$$D_{calc} = -\left(\frac{10^{-5}\gamma_i\gamma_j h}{2\pi^2 r_{ij}^3}\right) \sum_{m,n=xyz} \cos\phi_m \cdot S_{mn} \cdot \cos\phi_n$$

where ϕ_x is the angle between the internuclear vector and the x axis. Geometrically, the splitting is proportional to the transformation of the alignment tensor onto the internuclear axis. This is just Eqs. (5) and (13) of the above reference, with any internal motion corrections (which might be a part of S_{system}) set to unity. If there is an internal motion correction which is the same for all observations, this can be assimilated into the alignment tensor. The current code does not allow for variable corrections for internal motion. See Ref. [201] for a fuller discussion of these issues.

At the end of the calculation, the alignment tensor is diagonalized to obtain information about its principal components. This allows the alignment tensor to be written in terms of the "axial" and "rhombic" components that are often used to describe alignment.

Variables in the &align namelist.

`ndip`	Number of observed dipolar couplings to be used as restraints.

`id, jd` Atom numbers of the two atoms involved in the dipolar coupling.

`dobsl, dobsu` Limiting values for the observed dipolar splitting, in Hz. If the calculated coupling is less than *dobsl*, the energy penalty is proportional to $(D_{calc} - D_{obs,l})^2$; if it is larger than *dobsu*, the penalty is proportional to $(D_{calc} - D_{obs,lu})^2$. Calculated values between *dobsl* and *dobsu* are not penalized. Note that *dobsl* must be less than *dobsu*; for example, if the observed coupling is -6 Hz, and a 1 Hz "buffer" is desired, you could set *dobsl* to -7 and *dobsu* to -5.

`dwt` The relative weight of each observed value. Default is 1.0. The penalty function is thus:

$$E^i_{align} = D^i_{wt}(D^i_{calc} - D^i_{obs(u,l)})^2$$

where D_{wt} may vary from one observed value to the next. Note that the default value is arbitrary, and a smaller value may be required to avoid overfitting the dipolar coupling data.[201]

`dataset` Each dipolar peak can be associated with a "dataset", and a separate alignment tensor will be computed for each dataset. This is generally used if there are several sets of experiments, each with a different sample or temperature, etc., that would imply a different value for the alignment tensor. By default, there is one dataset to which each observed value is assigned.

`num_datasets` The number of datasets in the constraint list. Default is 1.

`s11,s12,s13,s22,s23` Initial values for the Cartesian components of the alignment tensor. The tensor is traceless, so S33 is calculated as -(S11+S22). In order to have the order of magnitude of the S values be roughly commensurate with coordinates in Angstroms, the alignment tensor values must be multiplied by 10^5.

`gigj` Product of the nuclear "g" factors for this dipolar coupling restraint. These are related to the nuclear gyromagnetic rations by $\gamma_N = g_N\beta_N/$. Common values are ^{1}H = 5.5856, ^{13}C = 1.4048, ^{15}N = -0.5663, ^{31}P = 2.2632.

`dij` The internuclear distance for observed dipolar coupling. If a nonzero value is given, the distance is considered to be fixed at the given value. If a *dij* value is zero, its value is computed from the structure, and it is assumed to be a variable distance. For one-bond couplings, it is usually best to treat the bond distance as "fixed" to an effective zero-point vibration value.[202]

`dcut` Controls printing of calculated and observed dipolar couplings. Only values where abs(dobs(u,l) - dcalc) is greater than *dcut* will be printed. Default is 0.1 Hz. Set to a negative value to print all dipolar restraint information.

`freezemol` If this is set to *.true.*, the molecular coordinates are not allowed to vary during dynamics or minimization: only the elements of the alignment tensor will change. This is useful to fit just an alignment tensor to a given structure. Default is *.false.*.

9.6 Residual CSA or pseudo-CSA restraints

Resonance positions in partially aligned media will be shifted from their positions in isotropic media, and this can provide information that is very similar to residual dipolar coupling constriants. This section shows how to input these sorts of restraints. The entry of the alignment tensor is done as in Section 9.5, so you must have a DIPOLE file (with an &align namelist) even if you don't have any RDC restraints. Then, if there is a CSA line in I/O redirection section, that file will be read with the following inputs:

Variables in the &csa namelist.

`ncsa` Number of observed residual CSA peaks to be used as restraints.

`icsa,jcsa,kcsa` Atom numbers for the csa of interest: *jcsa* is the atom whose $\Delta\sigma$ value has been measured; *icsa* and *kcsa* are two atoms bonded to it, used to define the local axis frame for the CSA tensor. See *amber12/test/pcsa/RST.csa* for examples of how to set these.

`cobsl, cobsu` Limiting values for the observed residual CSA, in Hz (not ppm or ppb!). If the calculated value of $\Delta\sigma$ is less than *cobsl*, the energy penalty is proportional to $(\Delta\sigma_{calc} - \Delta\sigma_{obs,l})^2$; if it is larger than *cobsu*, the penalty is proportional to $(\Delta\sigma_{calc} - \Delta\sigma_{obs,u})^2$. Calculated values between *cobsl* and *cobsu* are not penalized. Note that *cobsl* must be less than *cobsu*.

`cwt` The relative weight of each observed value. Default is 1.0. The penalty function is thus:

$$E^i_{csa} = C^i_{wt}(\Delta\sigma^i_{calc} - \Delta\sigma^i_{obs(u,l)})^2$$

where C_{wt} may vary from one observed value to the next. Note that the default value is arbitrary, and a smaller value may be required to avoid overfitting the data.

`datasetc` Each residual CSA can be associated with a "dataset", and a separate alignment tensor will be computed for each dataset. This is generally used if there are several sets of experiments, each with a different sample or temperature, etc., that would imply a different value for the alignment tensor. By default, there is one dataset to which each observed value is assigned. The tensors themselves are entered for each dataset in the DIPOLE file.

`field` Magnetic field (in MHz) for the residual CSA being considered here. This is indexed from 1 to *ncsa,* and is nucleus dependent. For example, if the proton frequency is 600 MHz, then *field* for ^{13}C would be 150, and that for ^{15}N would be 60.

`sigma11, sigma22, sigma12, sigma13, sigma23` Values of the CSA tensor (in ppm) for atom *icsa*, in the local coordinate frame defined by atoms *icsa*, *jcsa* and *kcsa*. See *$AMBERHOME/test/pcsa/RST.csa* for examples of how to set these.

`ccut` Controls printing of calculated and observed residual CSAs. Only values where abs(cobs(u,l) - ccalc) is greater than *ccut* will be printed. Default is 0.1 Hz. Set to a negative value to print all information.

The residual CSA facility is new as of Amber 10, and has not been used as much as other parts of the NMR refinement package. You should study the example files listed above to see how things work. The residual CSA values should closely match those found by the RAMAH package (*http://www-personal.umich.edu/~hashimi/Software.html*), and testing this should be a first step in making sure you have entered the data correctly.

9.7 Preparing restraint files for Sander

Fig. 9.1 shows the general information flow for auxiliary programs that help prepare the restraint files. Once the restraint files are made, Fig. 9.2 shows a flow-chart of the general way in which *sander* refinements are carried out.

The basic ideas of this scheme owe a lot to the general experience of the NMR community over the past decade. Several papers outline procedures in the Scripps group, from which a lot of the NMR parts of *sander* are derived.[193, 203–207] They are by no means the only way to proceed. We hope that the flexibility incorporated into *sander* will encourage folks to experiment with refinement protocols.

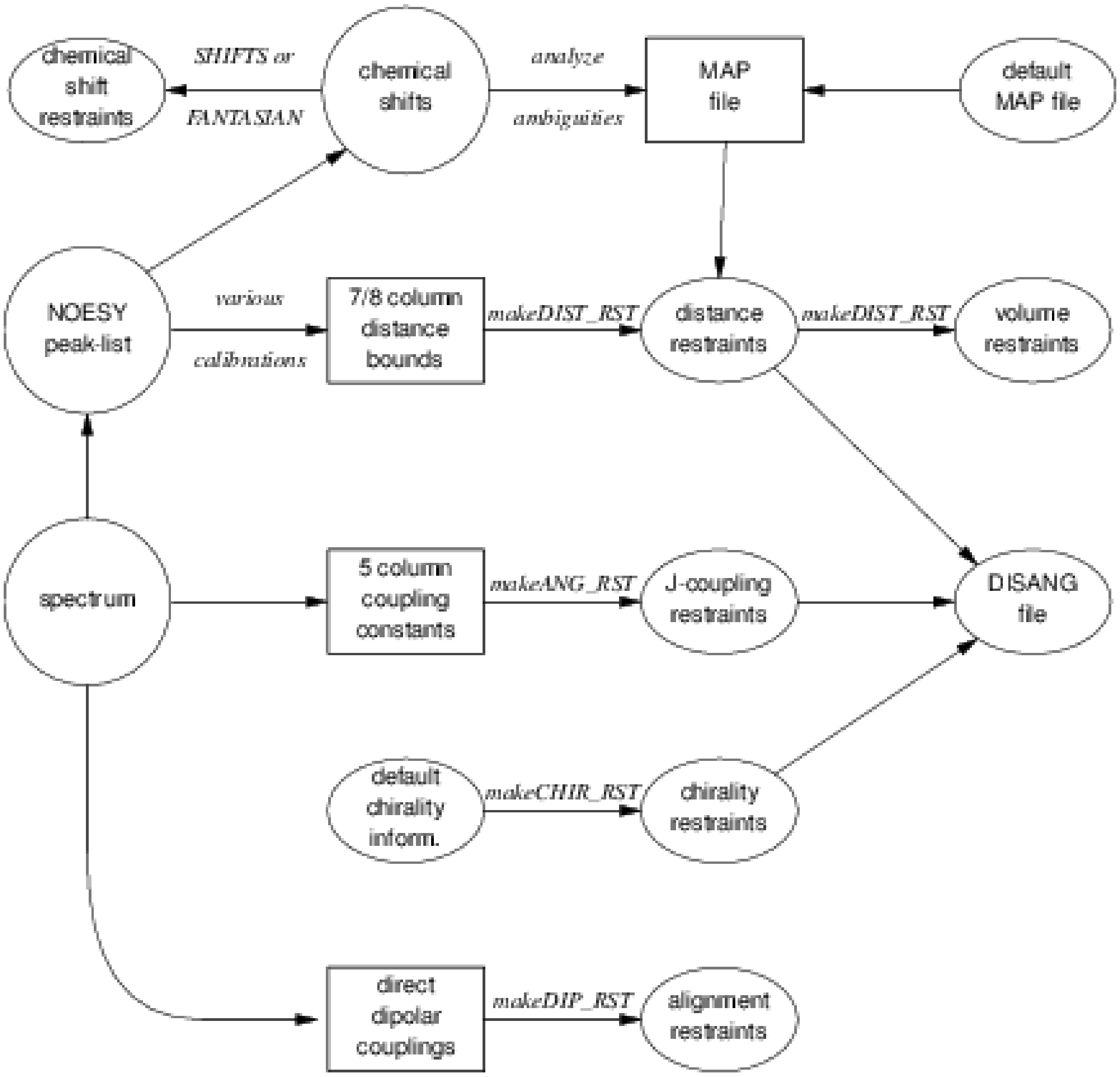


Figure 9.1: *Notation:* circles *represent logical information, whose format might differ from one project to the next;* solid rectangles *are in a specific format (largely compatible with DIANA and other programs), and are intended to be read and edited by the user;* ellipses *are specific to* sander, *and are generally not intended to be read or edited manually. The conversion of NOESY volumes to distance bounds can be carried out by a variety of programs such as* mardigras *or* xpk2bound *that are not included with Amber. Similarly, the analysis and partial assignment of ambiguous or overlapped peaks is a separate task; at TSRI, these are typically carried out using the programs* xpkasgn *and* filter.pl

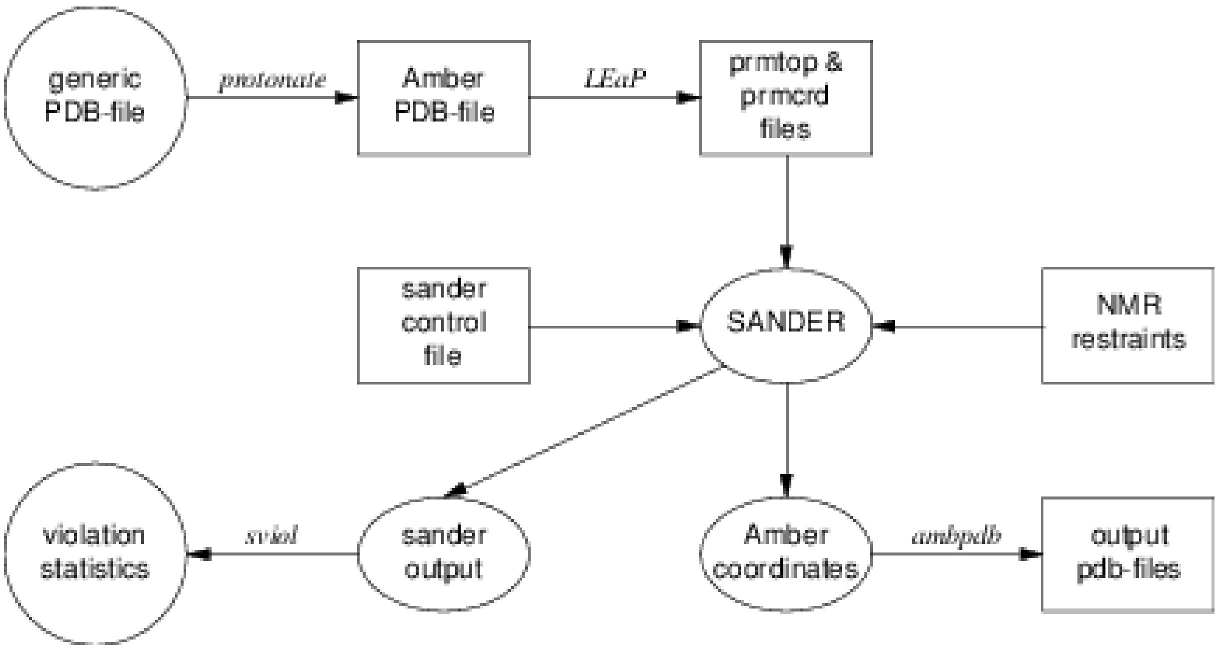


Figure 9.2: *General organization of NMR refinement calculations.*

9.7.1 Preparing distance restraints: makeDIST_RST

The *makeDIST_RST* program converts a simplified description of distance bounds into a detailed input for *sander*. A variety of input and output filenames may be specified on the command line:
input:

```
-upb <filename> 7-col file of upper distance bounds, OR
-ual <filename> 8-col file of upper and lower bounds, OR
-vol <filename> 7-col file of NOESY volumes
-pdb <filename> Brookhaven format file
-map <filename> MAP file (default:map.DG-AMBER)
-les <filename> LES atom mappings, made by addles
```

output:

```
-dgm <filename> DGEOM95 restraint format
-rst <filename> SANDER restraint format
-svf <filename> Sander Volume Format, for NOESY refinement
```

other options:

```
-help (gives you this explanation, overrides other parameters)
-report (gives you short runtime diagnostic output)
-nocorr (do not correct upper bound for r**-6 averaging)
-altdis (use alternative form for the distance restraints)
```

The *7/8 column distance bound* file is essentially that used by the DIANA or DISGEO programs. It consists of one-line per restraint, which would typically look like the following:

```
23 ALA HA 52 VAL H 3.8 # comments go here
```

The first three columns identify the first proton, the next three the second proton, and the seventh column gives the upper bound. Only the first three letters of the residue name are used, so that DIANA files that contain residues like "ASP-" will be correctly interpreted. An alternate, 8-column, format has both upper and lower bounds as the seventh and eighth columns, respectively. A typical line might in an "8-col" file might look like this:

```
23 ALA HA 52 VAL H 3.2 3.8 # comments go here
```

Here the lower bound is 3.2 Åand the upper bound is 3.8 Å. Comments typically identify the spectrum and peak-number or other identification that allow cross-referencing back to the appropriate spectrum. If the comment contains the pattern "<integer>:<integer>", then the first integer is treated as a peak-identifier, and the second as a spectrum-identifier. These identifiers go into the *ixpk* and *nxpk* variables, and will later be printed out in *sander*, to facilitate going back to the original spectra to track down violations, etc.

The format for the *-vol* option is the same as for the *-upb* option except that the seventh column holds a peak intensity (volume) value, rather than a distance upper bound.

The input PDB file must exactly match the Amber *prmtop* file that will be used; use the ambpdb -aatm command to create this.

If all peaks involved just single protons, and were fully assigned, this is all that one would need. In general, though, some peaks (especially methyl groups or fast-rotating aromatic rings) represent contributions from more than one proton, and many other peaks may not be fully assigned. *Sander* handles both of these situations in the same way, through the notion of an "ambiguous" peak, that may correspond to several assignments. These peaks are given two types of special names in the 7/8-column format file:

1. Commonly-occurring ambiguities, like the lack of stereospecific assignments to two methylene protons, are given names defined in the default MAP file. These names, also more-or-less consistent with DIANA, are like the names of "pseudo-atoms" that have long been used to identify such partially assigned peaks, e.g. "QB" refers to the (HB2,HB3) combination in most residues, and "MG1" in valine refers collectively to the three methyl protons at position CG1, etc.

2. There are generally also molecule-specific ambiguities, arising from potential overlap in a NOESY spectrum. Here, the user assigns a unique name to each such ambiguity or overlap, and prepares a list of the potential assignments. The names are arbitrary, but might be constructed, for example, from the chemical shifts that identify the peak, e.g. "p_2.52" might identify the set of protons that could contribute to a peak at 2.52 ppm. The chemical shift list can be used to prepare a list of potential assignments, and these lists can often be pruned by comparison to approximate or initial structures.

The default and molecule-specific MAP files are combined into a single file, which is used, along with the 7-column restraint file, the the program *makeDIST_RST* to construct the actual *sander* input files. You should consult the help file for makeDIST_RST for more information. For example, here are some lines added to the MAP file for a recent TSRI refinement:

```
AMBIG n2:68 = HE 86 HZ 86
AMBIG n2:72 = HE 24 HD 24 HZ 24
AMBIG n2:73 = HN 81 HZ 13 HE 13 HD 13 HZ 24
AMBIG n2:78 = HN 76 HZ 13 HE 13 HZ 24
AMBIG n2:83 = HN 96 HN 97 HD 97 HD 91
AMBIG n2:86 = HD1 66 HZ2 66
AMBIG n2:87 = HN 71 HH2 66 HZ3 66 HD1 66
```

Here the spectrum name and peak number were used to construct a label for each ambiguous peak. Then, an entry in the restraint file might look like this:

```
123 GLY HN 0 AMB n2:68 5.5
```

indicating a 5.5 Åupper bound between the amide proton of Gly 123 and a second proton, which might be either the HE or HZ protons of residue 86. (The "zero" residue number just serves as a placeholder, so that there will be the same number of columns as for non-ambiguous restraints.) If it is possible that the ambiguous list might not be exhaustive (e.g. if some protons have not been assigned), it is safest to set *ialtd*=1, which will allow "mistakes" to be present in the constraint list. On the other hand, if you want to be sure that every violation is "active", set *ialtd*=0.

If the *-les* flag is set, the program will prepare distance restraints for multiple copies (LES) simulations. In this case, the input PDB file is one *without* LES copies, i.e. with just a single copy of the molecule. The "lesfile" specified by this flag is created by the *addles* program, and contains a mapping from original atom numbers into the copy numbers used in the multiple-copies simulation.

The *-rst* and *-svf* flags specify outputs for *sander*, for distance restraints and NOESY restraints, respectively. In each case, you may need to hand-edit the outputs to add additional parameters. You should make it a habit to compare the outputs with the descriptions given earlier in this chapter to make sure that the restraints are what you want them to be.

It is common to run makeDIST_RST several times, with different inputs that correspond to different spectra, different mixing times, etc. It is then expected that you will manually edit the various output files to combine them into the single file required by *sander*.

9.7.2 Preparing torsion angle restraints: makeANG_RST

There are fewer "standards" for representing coupling constant information. We have followed the DIANA convention in the program *makeANG_RST*. This program takes as input a five-column torsion angle constraint file along with an Amber PDB file of the molecule. It creates as output (to standard out) a list of constraints in RST format that is readable by Amber.

```
Usage: makeANG_RST -help
makeANG_RST -pdb ambpdb_file [-con constraint] [-lib libfile]
[-les lesfile ]
```

The input torsion angle constraint file can be read from standard in or from a file specified by the -con option on the command line. The input constraint file should look something like this:

```
1 GUA PPA 111.5 144.0
2 CYT EPSILN 20.9 100.0
2 CYT PPA 115.9 134.2
3 THY ALPHA 20.4 35.6
4 ADE GAMMA 54.7 78.8
5 GLY PHI 30.5 60.3
6 ALA CHI 20.0 50.0
....
```

Lines beginning with "#" are ignored. The first column is the residue number; the second is the residue name (three letter code, or as defined in your personal torsion library file). Only the first three letters of the residue name are used, so that DIANA files that contain residues like "ASP-" will be correctly interpreted. Third is the angle name (taken from the torsion library described below).The fourth column contains the lower bound, and the fifth column specifies the upper bound. Additional material on the line is (presently) ignored.

Note: It is assumed that the lower bound and the upper bound define a region of allowed conformation on the unit circle that is swept out in a clockwise direction from $lb \rightarrow ub$. If the number in the *lb* column is greater than the the number in the *ub* column, 360°will successively be subtracted from the *lb* until $lb < ub$. This preserves the clockwise definition of the allowed conformation space, while also making the number that specifies the lower bound less than the number that specifies the upper bound, as is required by Amber. If this occurs, a warning message will be printed to *stderr* to notify the user that the data has been modified.

The angles that one can constrain in this manner are defined in the library file that can be optionally specified on the command line with the -lib flag, or the default library "tordef.lib" (written by Garry P. Gippert) will be used. If you wish to specify your own nomenclature, or add angles that are not already defined in the default file, you should make a copy of this file and modify it to suit your needs. The general format for an entry in the library is:

```
LEU PSI N CA C N+
```

where the first column is the residue name, the second column is the angle name that will appear in the input file when specifying this angle, and the last four columns are the atom names that define the torsion angle. When a torsion angle contains atom(s) from a preceding or succeeding residue in the structure, a "-" or "+" is appended to those atom names in the library, thereby specifying that this is the case. In the example above, the atoms that define PSI for LEU residues are the N, CA, and C atoms of that same LEU and the N atom of the residue after that LEU in the primary structure. Note that the order of atoms in the definition is important and should reflect that the torsion angle rotates about the two central atoms as well as the fact that the four atoms are bonded in the order that is specified in the definition.

If the first letter of the second field is "J", this torsion is assumed to be a J-coupling constraint. In that case, three additional floats are read at the end of the line, giving the A,B and C coefficients for the Karplus relation for this torsion. For example:

```
ALA JHNA H N CA HA 9.5 -1.4 0.3
```

will set up a J-coupling restraint for the HN-HA 3-bond coupling, assuming a Karplus relation with A,B, C as 9.5, -1.4 and 0.3. (These particular values are from Brüschweiler and Case, JACS 116: 11199 (1994).)

This program also supports pseudorotation phase angle constraints for prolines and nucleic acid sugars; each of these will generate restraints for the 5 component angles which correspond to the *lb* and *ub* values of the input pseudorotation constraint. In the torsion library, a pseudorotation definition looks like:

```
PSEUDO CYT PPA NU0 NU1 NU2 NU3 NU4
CYT NU0 C4' O4' C1' C2'
CYT NU1 O4' C1' C2' C3'
CYT NU2 C1' C2' C3' C4'
CYT NU3 C2' C3' C4' O4'
CYT NU4 C3' C4' O4' C1'
```

The first line describes that a PSEUDOrotation angle is to be defined for CYT that is called PPA and is made up of the five angles NU0-NU4. Then the definition for NU0-NU4 should also appear in the file in the same format as the example given above for LEU PSI.

PPA stands for Pseudorotation Phase Angle and is the angle that should appear in the input constraint file when using pseudorotation constraints. The program then uses the definition of that PPA angle in the library file to look for the 5 other angles (NU0-NU4 in this case) which it then generates restraints for. PPA for proline residues is included in the standard library as well as for the DNA nucleotides.

If the *-les* flag is set, the program will prepare torsion angle restraints for multiple copies (LES) simulations. In this case, the input PDB file is one *without* LES copies, i.e. with just a single copy of the molecule. The "lesfile" specified by this flag is created by the *addles* program, and contains a mapping from original atom numbers into the copy numbers used in the multiple-copies simulation.

Torsion angle constraints defined here cannot span two different copy sets, i.e., there cannot be some atoms of a particular torsion that are in one multiple copy set, and other atoms from the same torsion that are in other copy sets. It *is* OK to have some atoms with single copies, and others with multiple copies in the same torsion. The program will create as many duplicate torsions as there are copies.

A good alternative to interpreting J-coupling constants in terms of torsion angle restraints is to refine directly against the coupling constants themselves, using an appropriate Karplus relation. See the discussion of the variable RJCOEF, above.

9.7.3 Chirality restraints: makeCHIR_RST

```
Usage: makeCHIR_RST <pdb-file>  <output-constraint-file>
```

We also find it useful to add chirality constraints and *trans*-peptide ω constraints (where appropriate) to prevent chirality inversions or peptide bond flips during the high-temperature portions of simulated annealing runs. The program *makeCHIR_RST* will create these constraints. Note that you may have to edit the output of this program to change *trans* peptide constraints to *cis*, as appropriate.

9.7.4 Direct dipolar coupling restraints: makeDIP_RST

For simulations with residual dipolar coupling restraints, the *makeDIP_RST.protein*, *makeDIP_RST.dna* and *makeDIP_RST.diana* are simple codes to prepare the input file. Use *-help* to obtain a more detailed description of the usage. For now, this code only handles backbone NH and CαH data. The header specifying values for various parameters needs to be manually added to the output of *makeDIP_RST*.

Use of residual dipolar coupling restraints is new both for Amber and for the general NMR community. Refinement against these data should be carried out with care, and the optimal values for the force constant, penalty function, and initial guesses for the alignment tensor components are still under investigation. Here are some suggestions from the experiences so far:

1. Beware of overfitting the dipolar coupling data in the expense of Amber force field energy. These dipolar coupling data are very sensitive to tiny changes in the structure. It is often possible to drastically improve the fitting by making small distortions in the backbone angles. We recommend inclusion of explicit angle restraints to enforce ideal backbone geometry, especially for those residues that have corresponding residual dipolar coupling data.

2. The initial values for the Cartesian components of the alignment tensor can influence the final structure and alignment if the structure is not fixed (ibelly = 0). For a fixed structure (ibelly = 1), these values do not matter. Therefore, the current "best" strategy is to fit the experimental data to the fixed starting structure, and use the alignment tensor[s] obtained from this fitting as the initial guesses for further refinement.

3. Amber is capable of simultaneously fitting more than one set of alignment data. This allows the use of individually obtained datasets with different alignment tensors. However, if the different sets of data have equal directions of alignment but different magnitudes, using an overall scaling factor for these data with a single alignment tensor could greatly reduce the number of fitting parameters.

4. Because the dipolar coupling splittings depend on the square root of the order parameters ($0 \leq S2 \leq 1$), these order parameters describing internal motion of individual residues are often neglected (N. Tjandra and A. Bax, *Science* **278**, 1111-1113, 1997). However, the square root of a small number can still be noticeably smaller than 1, so this may introduce undesirable errors in the calculations.

9.7.5 Using NMR exchange format (NEF) files

The NMR community, in collaboration with the worldwide PDB, is developing a common format for encoding of NMR restraints, including all of the kinds discussed above. This format is not yet finalized, but we are including here a conversion script, *nef_to_RST*, that would convert these files to *sander* format. Because this format is so new, and is still subject to revisions, care should be taken in using this script: make sure that the output files do what they should be doing. Here are the usage instructions (which you can also get by typing “nef_to_RST -help” at the command line:

```
# nef_to_RST
convert NEF restraints to Amber format
input:
   -nef <filename>: NEF file
   -pdb <filename>: PDBFILE using AMBER nomenclature and numbering
   -map <filename>: MAP file  (default:map.NEF-AMBER)
output:
   -rst <filename>:  SANDER restraint format
   -rdc <filename>:  SANDER DIP format

other options:
   -nocorr (do not correct upper bound for r**-6 averaging)
   -altdis (use alternative form for the distance restraints)
   -help   (gives you this explanation, overrides other parameters)
   -report (gives you short runtime diagnostic output)
errors come to stderr.
```

9.8 Getting summaries of NMR violations

If you specify LISTOUT=POUT when running *sander*, the output file will contain a lot of detailed information about the remaining restraint violations at the end of the run. When running a family of structures, it can be useful to process these output files with *sviol*, which takes a list of *sander* output files on the command line, and sends a summary of energies and violations to STDOUT. If you have more than 20 or so structures to analyze, the output from *sviol* becomes unwieldy. In this case you may also wish to use *sviol2*, which prints out somewhat less detailed information, but which can be used on larger families of structures. The *senergy* script gives a more detailed view of force-field energies from a series of structures. (We thank the TSRI NMR community for helping to put these scripts together, and for providing many useful suggestions.)

9.9 Time-averaged restraints

The model of the previous sections involves the "single-average-structure" idea, and tries to fit all constraints to a single model, with minimal deviations. A generalization of this model treats distance constraints arising from from NOE crosspeaks (for example) as being the average distance determined from a trajectory, rather than as the single distance derived from an average structure.

Time-averaged bonds and angles are calculated as

$$\bar{r} = (1/C)\left\{\int_0^t e^{(t'-t)/\tau} r(t')^{-i} dt'\right\}^{-1/i} \tag{9.1}$$

where

$\bar{r}$	= time-averaged value of the internal coordinate (distance or angle)
t	= the current time
τ	= the exponential decay constant
$r(t')$	= the value of the internal coordinate at time t′
i	= average is over internals to the inverse of i. Usually i = 3 or 6 for NOE distances, and -1 (linear averaging) for angles and torsions.
C	= a normalization integral.

Time-averaged torsions are calculated as

$$<\varphi>= \tan^{-1}(<\sin(\phi)> / <\cos(\phi)>)$$

where ϕ is the torsion, and $< \sin(\phi) >$ and $< \cos(\phi) >$ are calculated using the equation above with $\sin(\phi(t'))$ or $\cos(\phi(t'))$ substituted for $r(t')$.

Forces for time-averaged restraints can be calculated either of two ways. This option is chosen with the DISAVI / ANGAVI / TORAVI commands. In the first (the default),

$$\partial E/\partial x = (\partial E/\partial \bar{r})(\partial \bar{r}/\partial r(t))(\partial r(t)/\partial x) \tag{9.2}$$

(and analogously for y and z). The forces then correspond to the standard flat-bottomed well functional form, with the instantaneous value of the internal replaced by the time-averaged value. For example, when $r_3 < \bar{r} < r_4$,

$$E = k_3(\bar{r} - r_3)^2$$

and similarly for other ranges of $\bar{r}$.

When the second option for calculating forces is chosen (IINC = 1 on a DISAVI, ANGAVI or TORAVI card), forces are calculated as

$$\partial E/\partial x = (\partial E/\partial \bar{r})(\partial r(t)/\partial x) \tag{9.3}$$

For example, when $r_3 < \bar{r} < r_4$,

$$\partial E/\partial x = 2k_3(\bar{r} - r_3)(\partial r(t)/\partial x)$$

Integration of this equation does not give Eq. 9.2, but rather a non-intuitive expression for the energy (although one that still forces the bond to the target range). The reason that it may sometimes be preferable to use this second option is that the term $\partial \bar{r}/\partial r(t)$, which occurs in the exact expression [Eq. 9.2], varies as $(\bar{r}/r(t))^{1+i}$. When i=3, this means the forces can be varying with the fourth power the distance, which can possibly lead to very large transient forces and instabilities in the molecular dynamics trajectory. [Note that this will not be the case when linear scaling is performed, i.e. when $i = -1$, as is generally the case for valence and torsion angles. Thus, for linear scaling, the default (exact) force calculation should be used].

It should be noted that forces calculated using Eq. 9.3 are not conservative forces, and would cause the system to gradually heat up, if no velocity rescaling were performed. The temperature coupling algorithm should act to maintain the average temperature near the target value. At any rate, this heating tendency should not be a problem in simulations, such as fitting NMR data, where MD is being used to sample conformational space rather than to extract thermodynamic data.

This section has described the methods of time-averaged restraints. For more discussion, the interested user is urged to consult studies where this method has been used.[208–212]

9.10 Multiple copies refinement using LES

NMR restraints can be made compatible with the multiple copies (LES) facility; see the following chapter for more information about LES. To use NMR constraints with LES, you need to do two things:

(1) Add a line like "file wnmr name=(lesnmr) wovr" to your input to *addles*. The filename (lesnmr in this example) may be whatever you wish. This will cause *addles* to output an additional file that is needed at the next step.

(2) Add "-les lesnmr" to the command line arguments to *makeDIST_RST*. This will read in the file created by *addles* containing information about the copies. All NMR restraints will then be interpreted as "ambiguous" restraints, so that if any of the copies satisfies the restraint, the penalty goes to zero.

Note that although this scheme has worked well on small peptide test cases, we have yet not used it extensively for larger problems. This should be treated as an experimental option, and users should use caution in applying or interpreting the results.

9.11 Some sample input files

The next few pages contain excerpts from some sample NMR refinement files used at TSRI. The first example just sets up a simple (but often effective) simulated annealing run. You may have to adjust the length, temperature maximum, etc. somewhat to fit your problem, but these values work well for many "ordinary" NMR problems.

9.11.1 1. Simulated annealing NMR refinement

```
15ps simulated annealing protocol
&cntrl
  nstlim=15000, ntt=1, !(time limit, temp. control)
  ntpr=500, pencut=0.1, !(control of printout)
  ipnlty=1, nmropt=1, !(NMR penalty function options)
  vlimit=10, !(prevent bad temp. jumps)
  ntb=0, !(non-periodic simulation)
  igb=8, !(generalize Born solvent model)
/
#
# Simple simulated annealing algorithm:
#
# from steps 0 to 1000: raise target temperature 10-1200K
# from steps 1000 to 3000: leave at 1200K
# from steps 3000 to 15000: re-cool to low temperatures
#
&wt type='TEMP0', istep1=0,istep2=1000,value1=10.,
  value2=1200., /
&wt type='TEMP0', istep1=1001, istep2=3000, value1=1200.,
  value2=1200.0, /
&wt type='TEMP0', istep1=3001, istep2=15000, value1=0.,
  value2=0.0, /
#
# Strength of temperature coupling:
# steps 0 to 3000: tight coupling for heating and equilibration
# steps 3000 to 11000: slow cooling phase
# steps 11000 to 13000: somewhat faster cooling
# steps 13000 to 15000: fast cooling, like a minimization
#
&wt type='TAUTP', istep1=0,istep2=3000,value1=0.2,value2=0.2, /
&wt type='TAUTP', istep1=3001,istep2=11000,value1=4.0,value2=2.0, /
&wt type='TAUTP', istep1=11001,istep2=13000,value1=1.0,value2=1.0, /
&wt type='TAUTP', istep1=13001,istep2=14000,value1=0.5,value2=0.5, /
&wt type='TAUTP', istep1=14001,istep2=15000,value1=0.05,value2=0.05, /
#
# "Ramp up" the restraints over the first 3000 steps:
#
&wt type='REST', istep1=0,istep2=3000,value1=0.1,value2=1.0, /
&wt type='REST', istep1=3001,istep2=15000,value1=1.0,value2=1.0, /
&wt type='END' /
LISTOUT=POUT (get restraint violation list)
DISANG=RST.f (file containing NMR restraints)
```

The next example just shows some parts of the actual RST file that *sander* would read. This file would ordinarily *not* be made or edited by hand; rather, run the programs *makeDIST_RST*, *makeANG_RST* and *makeCHIR_RST*, combining the three outputs together to construct the RST file.

9.11.2 Part of the RST.f file referred to above

```
# first, some distance constraints prepared by makeDIST_RST:
 # (comment line is input to makeRST, &rst namelist is output)
 #
 #( proton 1 proton 2 upper bound)
 #---------------------------------------------
 #
 # 2 ILE HA 3 ALA HN 4.00
 #
 &rst iat= 23, 40, r3= 4.00, r4= 4.50,
 r1 = 1.3, r2 = 1.8, rk2=0.0, rk3=32.0, ir6=1, /
 #
 # 3 ALA HA 4 GLU HN 4.00
 #
 &rst iat= 42, 50, r3= 4.00, r4= 4.50, /
 #
 # 3 ALA HN 3 ALA MB 5.50
 #
 &rst iat= 40, -1, r3= 6.22, r4= 6.72,
 igr1= 0, 0, 0, 0, igr2= 44, 45, 46, 0, /
 #
 # .......etc......
 #
 # next, some dihedral angle constraints, from makeANG_RST:
 #
 &rst iat= 213, 215, 217, 233, r1=-190.0,
 r2=-160.0, r3= -80.0, r4= -50.0, /
 &rst iat= 233, 235, 237, 249, r1=-190.0,
 r2=-160.0, r3= -80.0, r4= -50.0, /
 # .......etc.......
 #
 # next, chirality and omega constraints prepared by makeCHIR_RST:
 #
 #
 # chirality for residue 1 atoms: CA CG HB2 HB3
 &rst iat= 3 , 8 , 6 , 7 ,
 r1=10., r2=60., r3=80., r4=130., rk2 = 10., rk3=10., /
 #
 # chirality for residue 1 atoms: CB SD HG2 HG3
 &rst iat= 5 , 11 , 9 , 10 , /
 #
 # chirality for residue 1 atoms: N C HA CB
 &rst iat= 1 , 18 , 4 , 5 , /
 #
 # chirality for residue 2 atoms: CA CG2 CG1 HB
 &rst iat= 22 , 26 , 30 , 25 , /
 #
 ......etc........
 # trans-omega constraint for residue 2
 &rst iat= 22 , 20 , 18 , 3 ,
 r1=155., r2=175., r3=185., r4=205., rk2 = 80., rk3=80., /
 #
 # trans-omega constraint for residue 3
 &rst iat= 41 , 39 , 37 , 22 , /
 #
 # trans-omega constraint for residue 4
 &rst iat= 51 , 49 , 47 , 41 , /
 #
 # ......etc........
 #
 The next example is an input file for volume-based NOE refinement. As with the dist
```

9.11.3 3. Sample NOESY intensity input file

```
# A part of a NOESY intensity file:
 &noeexp
 id2o=1, (exchangeable protons removed)
 oscale=6.21e-4, (scale between exp. and calc. intensity units)
 taumet=0.04, (correlation time for methyl rotation, in ns.)
 taurot=4.2, (protein tumbling time, in ns.)
 NPEAK = 13*3, (three peaks, each with 13 mixing times)
 EMIX = 2.0E-02, 3.0E-02, 4.0E-02, 5.0E-02, 6.0E-02,
 8.0E-02, 0.1, 0.126, 0.175, 0.2, 0.25, 0.3, 0.35,
 (mixing times, in sec.)
 IHP(1,1) = 13*423, IHP(1,2) = 13*1029, IHP(1,3) = 13*421,
 (number of the first proton)
 JHP(1,1) = 78*568, JHP(1,2) = 65*1057, JHP(1,3) = 13*421,
 (number of the second proton)
 AEXP(1,1) = 5.7244, 7.6276, 7.7677, 9.3519,
 10.733, 15.348, 18.601,
 21.314, 26.999, 30.579,
 33.57, 37.23, 40.011,
 (intensities for the first cross-peak)
 AEXP(1,2) = 8.067, 11.095, 13.127, 18.316,
 22.19, 26.514, 30.748,
 39.438, 44.065, 47.336,
 54.467, 56.06, 60.113,
 AEXP(1,3) = 7.708, 13.019, 15.943, 19.374,
 25.322, 28.118, 35.118,
 40.581, 49.054, 53.083,
 56.297, 59.326, 62.174,
 /
 SUBMOL1
 RES 27 27 29 29 39 41 57 57 70 70 72 72 82 82 (residues in this submol)
END END
```

Next, we illustrate the form of the file that holds residual dipolar coupling restraints. Again, this would generally be created from a human-readable input using the program *makeDIP_RST*.

9.11.4 Residual dipolar restraints, prepared by makeDIP_RST:

```
&align
ndip=91, dcut=-1.0, gigj = 37*-3.1631, 54*7.8467,
s11=3.883, s22=53.922, s12=33.855, s13=-4.508, s23=-0.559,
id(1)=188, jd(1)=189, dobsu(1)= 6.24, dobsl(1)= 6.24,
id(2)=208, jd(2)=209, dobsu(2)= -10.39, dobsl(1)= -10.39,
id(3)=243, jd(3)=244, dobsu(3)= -8.12, dobsl(1)= -8.12,
....
id(91)=1393, jd(91)=1394, dobsu(91)= -19.64, dobsl(91) = -19.64,
/
```

Finally, we show how the detailed input to *sander* could be used to generate a more complicated restraint. Here is where the user would have to understand the details of the RST file, since there are no "canned" programs to create this sort of restraint. This illustrates, though, the potential power of the program.

9.11.5 A more complicated constraint

```
# 1) Define two centers of mass. COM1 is defined by
# {C1 in residue 2; C1 in residue 3; N2 in residue 4; C1 in residue 5}.
# COM2 is defined by {C4 in residue 1; O4 in residue 1; N* in residue 1}.
# (These definitions are effected by the igr1/igr2 and grnam1/grnam2
# variables; You can use up to 200 atoms to define a center-of-mass
# group)
#
# 2) Set up a distance restraint between COM1 and COM2 which goes from a
# target value of 5.0A to 2.5A, with a force constant of 1.0, over steps 1-5000.
#
# 3) Set up a distance restraint between COM1 and COM2 which remains fixed
# at the value of 2.5A as the force slowly constant decreases from
# 1.0 to 0.01 over steps 5001-10000.
#
# 4) Sets up no distance restraint past step 10000, so that free (unrestrained)
# dynamics takes place past this step.
#
&rst iat=-1,-1, nstep1=1,nstep2=5000,
  iresid=1,irstyp=0,ifvari=1,ninc=0,imult=0,ir6=0,ifntyp=0,
  r1=0.00000E+00,r2=5.0000,r3=5.0000, r4=99.000,rk2=1.0000,rk3=1.0000,
  r1a=0.00000E+00,r2a=2.5000,r3a=2.5000, r4a=99.000,rk2a=1.0000,rk3a=1.0000,
  igr1 = 2,3,4,5,0, grnam1(1)='C1',grnam1(2)='C1',grnam1(3)='N2',
  grnam1(4)='C1', igr2 = 1,1,1,0, grnam2(1)='C4',grnam2(2)='O4',grnam2(3)='N*',
/
&rst iat=-1,-1, nstep1=5001,nstep2=10000,
  iresid=1,irstyp=0,ifvari=1,ninc=0,imult=0,ir6=0,ifntyp=0,
  r1=0.00000E+00,r2=2.5000,r3=2.5000, r4=99.000,rk2=1.0000,rk3=1.0000,
  r1a=0.00000E+00,r2a=2.5000,r3a=2.5000, r4a=99.000,rk2a=1.0000,rk3a=0.0100,
  igr1 = 2,3,4,5,0, grnam1(1)='C1',grnam1(2)='C1',grnam1(3)='N2',
  grnam1(4)='C1', igr2 = 1,1,1,0, grnam2(1)='C4',grnam2(2)='O4',grnam2(3)='N*',
/
```

10 Xray and cryoEM refinement

10.1 EMAP restraints for rigid and flexible fitting into EM maps

EMAP restrained simulation[20, 213] was developed to incorporate electron microscopy (EM) image information into macromolecular structure determination. Different from NMR and X-ray data, EM images have low resolutions (5~50Å). However, EM images of large molecular assemblies up to millions of atoms and in various biologically relevant environments are available. These low resolution images provide precious structural information that can help to determine structures of many molecular assemblies and machineries[213–223].

With EMAP restraints, Sander and PMEMD can be used to perform both rigid[213] and flexible[20] fitting of molecules into experimental maps of complexes to obtain both complex structures and conformations agreeing with experimental maps. In addition to experimental map information, homologous structural information can be used by EMAP to perform targeted conformational search (TCS) to induce simulation systems to form structures of interest.

If the restraint map or structure is very different from the starting conformation, SGLD is recommended to induce large conformational change by setting *isgld*=1. This is often used to simulate conformational transition between different states. See the Sampling and free energy search section 7.1 for details on running SGLD.

If domain motion is desired while domain structures need to be maintained, one can use an EMAP restraint generated from the initial coordinates for each domain and set *move*=1 to allow the restraint map to move with the domain, so that domains can search the conformational space without unfolding or changing shape.

Each EMAP restraint is defined by a map file and a selection of atoms, as well as related parameters. Multiple EMAP restraints can be defined. The map can be either input from an image file, or generated from a pdb structure or derived from the starting coordinates. The definition of EMAP restraints are read in from the input file as “&emap” namelists. The following are variables in each &emap namelist.

`mapfile`	The filename of a restraint map or structure. The restraint maps must be in “map”, “ccp4”, or “mrc” format. The structure must be in pdb format. The structure need not be the same as the simulation system. A resolution can be specified for the conversion to a density map. When a blank filename is specified, *mapfile*=”, the input coordinates of the masked atoms will be used to generate a restraint map (default=”).
`atmask`	The atom mask for selecting atoms to be restrained (default=’:*’).
`fcons`	The restraining constant (default=0.05 kcal/g).
`move`	Allow the restraint map to move when *move*>0 (default=0).
`resolution`	The resolution used to convert an atomic structure to a map (default=2 Å).
`ifit`	Perform rigid fitting before simulation when *ifit*>0. One would do this when the initial coordinates don’t match those of the map (default=0). When ifit=1, the map is transformed (by translation and rotation) to match the coordinates; the coordinates are not altered. EMAP allows output of the re-oriented map (*mapfit*=...) that matches the (final) simulation coordinates, *and/or* output of the coordinates (*molfit*=...) that would match the orientation of the original map. When ifit=2, the masked atoms will be transformed to fit the map and the transformed coordinates will be used for the following simulation. For periodic systems, ifit=2 may cause atoms to clash with periodic image atoms.

`grids` Grid numbers in x,y,z,phi,psi,theta dimensions for grid-threading rigid fitting[213]. For example, grids=2,2,2,3,3,3 defines 2 grid points in each of x,y,z directions between the minimum and maximum coordinates, and 3 grid points in each of phi (0-360), psi(0-360), theta(0-180) angles. A search for local minimums starts from every grid point and the global minimum is identified from all the local minimums (default=1,1,1,1,1,1).

`mapfit` The filename for the final constraint map after rigid fitting and/or moving. The filename must has an extension of .map, .ccp4, or .mrc (default=”, for no map output).

`molfit` The filename for the final restrained atom coordinates after rigid fitting and/or simulation. The filename must have an extension of .pdb (default=”, for no structure output).

Here is an example input file for an EMAP constrained SGLD simulation:

```
Map Constraint Self-Guided Langevin dynamics
&cntrl ntx=1, ntb=0,nstlim=100000, imin=0, maxcyc=1, ntc=2, ntf=2, cut=9.0,
ntpr=1000, ntwr=100000,ntwx=10000, ntt=3, gamma_ln=10.0,nscm=100, dt=0.001,
ntb=0,igb=0, isgld=1, tsgavg=1.0, sgft=0.5,tempsg=0,         (SGLD)
iemap=1,                    (turn on EMAP )
/
&emap          (EMAP restraint 1 )
mapfile='data/1gb1.ccp4',   (map is input from a map file)
atmask=':1-20',             (residues 1-20 are restrained)
fcons=0.1,
move=1,                     (restraint map can move)
ifit=1,                     (perform rigid fitting first)
mapfit='scratch/gb1n_1.ccp4',  (final map)
molfit='scratch/gb1n_1.pdb', / (final restrained atoms related to initial map)
&emap          (EMAP restraint 2)
mapfile='data/1gb1.pdb',  (map is generated from a pdb file)
atmask=':22-37',          (residues 22-37 are restrained)
fcons=0.1,move=0,         (restraint map is fixed)
ifit=1,                   (perform rigid fitting first)
mapfit='scratch/gb1h_1.ccp4',   (final map, same as initial)
molfit='scratch/gb1h_1.pdb', / (final restrained atoms related to initial map)
&emap          (EMAP restraint 3)
mapfile='',    (map is generated from initial coordinates)
atmask=':41-56',  (residues 41-56 are restrained)
fcons=0.1,move=1, (restraint map can move)
ifit=1,       (perform rigid fitting first)
mapfit='scratch/gb1c_1.ccp4',  (final map)
molfit='scratch/gb1c_1.pdb', / (final restrained atoms related to initial map)
```

10.2 Setting up crystal simulations

David S. Cerutti

Simulations of biomolecular crystals are in principle no different than any of the simulations that AMBER does in periodic boundary conditions. However, the setup of these systems is not trivial and probably cannot be accomplished with the LEaP software. Of principal importance are the construction of the solvent conditions (packing precise amounts of multiple solvent species into the simulation cell), and tailoring the unit cell dimensions to accommodate the inherently periodic nature of the system. The LEaP software, designed to construct simulations of molecules in solution, will overlay a pre-equilibrated solvent mask over the (biomolecular) solute, tile that mask throughout the simulation cell, and then prune solvent residues which clash with the solute. The result of this procedure is a system which will likely contract under constant pressure dynamics as the pruning process has left vacuum bubbles at the solute:solvent interface. Simulations of biomolecular crystals require that the simulation cell begin at a size corresponding to the crystallographic unit cell, and deviate very little from that size over the course of equilibration and onset of constant pressure dynamics. This demands a different strategy for placing solvent in the simulation cell. Four programs in the *AmberTools* release are designed to accomplish this. An example of their use in given in a web-based tutorial at *https://ambermd.org/tutorials/advanced/tutorial13/XtalTutor1.html* . A recent (2018) review of crystal simulations is also worth consulting.[224]

For brevity, only basic descriptions of the programs are given in this manual. All of the programs may be run with command line input; the input options to each program may be listed by running each program with no arguments.

10.2.1 UnitCell

A macromolecular crystal contains many repeating unit cells which stack like blocks in three dimensional space just as simulation cells do in periodic boundary conditions. Each unit cell, in turn, may contain multiple symmetry-related clusters of atoms. A PDB file contains one set of coordinates for the irreducible unit of the crystal, the “asymmetric unit,” and also information about the crystal space group and unit cell dimensions. The *UnitCell* program reads PDB files, seeking the SMTRY records within the REMARKs to enumerate the rotation and translation operations which may be applied to the coordinates given in the PDB file to reconstruct one complete unit cell.

Usage of the *UnitCell* program is as follows. The simple command rests on a critical assumption, that the PDB file contains an accurate CRYST1 record and that the REMARK 290 SMTRY records provide its space group symmetry operations.

```
UnitCell -p MyProtein.pdb -o UnitCell.pdb
```

10.2.2 PropPDB

Simulations in periodic boundary conditions require a minimum unit cell size: the simulation cell must be able to enclose a sphere of at least the nonbonded direct space cutoff radius plus a small buffer region for nonbonded pairlist updates. Many biomolecular crystal unit cells come in “shoebox” dimensions that may have one very short side; many unit cells are also not rectangular but triclinic, meaning that the size of the largest sphere they can enclose is further reduced. The workhorse simulation engine, pmemd.cuda, even requires that the simulation cell be at least three times as thick as the cutoff plus some buffer margin in order to run safely: for typical sum conditions this thickness is about 30Å. For these reasons, and perhaps to ensure that the rigid symmetry imposed by periodic boundary conditions does not create artifacts (crystallographic unit cells are equivalent when averaged over all time and space, but are not necessarily identical at any given moment), it may be necessary to include multiple unit cells within the simulation cell. This is the purpose of the *PropPDB* program: to propagate a unit cell in one or more directions so that the complete simulation cell meets minimum size requirements.

Drawing on the hypothetical example above, if the unit cell is too small we can extend it in the x and z dimensions:

```
PropPDB -p UnitCell -o ExpandedCell.pdb -ix 2 -iy 1 -iz 2
```

10.2.3 AddToBox

The *AddToBox* program handles placement of solvent within a crystal unit cell or supercell (as may be created by PropPDB). As described in the introduction, the basic strategy is to place solvent such that added solvent molecules do not clash with biomolecule solutes, but *may* clash with one another initially. This compromise is necessary because enough solvent must be added to the system to ensure that the correct unit cell dimensions are maintained in the long run, but it is not acceptable to place solvent within the interior of a biomolecule where it might not belong and never escape.

The *AddToBox* program takes a PDB file providing the coordinates of a complete biomolecular unit cell or supercell (argument -c), the dimensions by which that supercell repeats in space (the unit cell dimensions are taken from the CRYST1 record of this file), a PDB file describing the solvent residue to add (argument -a), and the number of copies of that solvent molecule to add (argument -na). *AddToBox* inherently assumes that the biomolecular unit cell it is initially presented may contain some amount of solvent already, and according to the AMBER convention of listing macromolecular solute atoms first and solvent last assumes that the first -P atoms in the file are the protein (or biomolecule). *AddToBox* will then color a very fine grid "black" if the grid point is within a certain distance of a biomolecular atom (argument -RP, default 5.0Å) or other solvent atom (argument -RW, default 1.0Å); the grid is "white" otherwise (the grid is stored in binary for memory efficiency). *AddToBox* will the make a copy of the solvent residue and randomly rotate and translate it somewhere within the unit cell. If all atoms of the solvent residue land on "white" grid voxels, the solvent molecule will become part of the system and the grid around the newly added solvent will be blacked out accordingly. If the solvent molecule cannot be placed, this process will be repeated until a million consecutive failures are encountered, at which point the program will terminate. If *AddToBox* has not placed the requested number of solvent molecules by the time it terminates, the -V option can be used to order the program to recursively call itself with progressively smaller solvent buffer distances until all the requested solvent can be placed. The output of the *AddToBox* program is another PDB named by the -o option.

Successful operation of *AddToBox* may take practice. If multiple solvent species are required, as is the case with heterogeneous crystallization solutions, *AddToBox* may be called repeatedly with each input molecular cell being the previous call's output. When considering crystal solvation, the order of addition is important! It is recommended that rare species, such as trace buffer reagents, be added first, with large -RW argument to ensure that they are dispersed throughout the available crystal void zones. Large solvent species such as MPD (an isohexane diol commonly used in crystallization conditions) should be added second, and with a sufficiently large -RW argument that methyl groups and ring systems cannot become interlocked (which will likely lead to SHAKE / vlimit errors). Small and abundant species such as water should be added last, as they can go anywhere that space remains.

Below is an example of the usage for a hypothetical protein with 5431 atoms and a net charge of +6 that is to be neutralized with ammonium sulfate:

```
AddToBox -c ExpandedCell.pdb -a Sulfate.pdb -na 18 -RP 3.0 -P 5431 -o System.pdb
AddToBox -c System.pdb -a Ammonium.pdb -na 30 -RP 3.0 -P 5431 -o System.pdb
AddToBox -c System.pdb -a Water.pdb -na 1089 -RP 3.0 -P 5431 -o System.pdb
```

The use of the -V flag ensures that the desired amounts of each species are included. The protein clipping radius of 3Å is lower than the default, but safe (remember, this radius stipulates that no solvent atom, regardless of the size of the solvent molecule, come within 3Å of the protein). Note how the original protein PDB file serves as the base for system, but thereafter we work with the System.pdb to accumulate more solvent particles. Here, the ammonium sulfate serves both to neutralize the system and replicate a salty bath, perhaps from a crystallization mother liquor, hence the break from the usual 2:1 stoichiometry of ammonium sulfate ions.

It is likely that the unobservable "void" regions between biomolecules in most crystals *do not* contain solvent species in proportion to their abundance in the crystallization solution–the vast majority of these regions are within a few Ångstroms of some biomolecular surface, and different biomolecular functional groups will preferentially interact with some types of solvent over others. Also, in many crystals some solvent molecules *are* observed; in many of these, the amount of solvent observed is such that it would be impossible to pack other species into the unit cell in proportion to their abundances in the crystallization fluid. In these cases, we recommend estimating the amount of volume that must be filled with solvent *apart from solvent which has already been observed in the crystal*, and filling this void with solvent in proportion to the composition of the crystallization fluid. For example, if a crystal were grown in a 1:1 mole-to-mole water/ethanol mixture, and the crystal coordinates as deposited in the PDB contained 500 water molecules and 3 ethanol molecules, we would use *AddToBox* to add water and ethanol in a

1:1 ratio until the system contained enough solvent to maintain the correct volume during equilibrium dynamics at constant pressure.

Finally, it is difficult to estimate exactly how much solvent will be needed to maintain the correct equilibrium volume; the advisable approach is simply to make an initial guess and script the setup so that, over multiple runs and reconstructions, the correct system composition can be found. We recommend matching the equilibrium unit cell volume to within 0.3% to keep this simulation parameter within the error of most crystallographic measurements. While errors of 0.5-1% will show up quickly after constant pressure dynamics begin, a 10 to 20ns simulation may be needed to ensure that the correct equilibrium volume has been achieved.

10.2.4 ChBox

After the complex process of adding solvent, the LEaP program may be used to produce a topology and initial set of coordinates based on the PDB file produced by *AddToBox*. By using the *SetBox* command, LEaP will create a periodic system without adding any more solvent on its own. The only problem with using LEaP at this point is that the program will fail to realize that the system *does* tile in three dimensions if only the box dimensions are set properly. If visualized, the output of UnitCell / PropPDB will likely look jagged, but the output of *AddToBox*, containing lots of added water, will make it obvious how parts of biomolecules jutting out one face of the box fit neatly into open spaces on an opposite face. The topology produced by LEaP needs no editing; only the last line of the coordinates does. This can be done manually, but the *ChBox* program automates the process, taking the same coordinates supplied to AddToBox and grafting them into the input coordinates file.

The program is even unnecessary in the case of orthorhombic (rectangular) unit cells, as this the tleap command will substitute:

```
    set [unit] box { <x> <y> <z> }
```

For cells that do not have only 90-degree box angles, *ChBox* will do the trick.

10.3 X-ray functionality and diffraction-based restraints

The *msander* program includes a new module dedicated to biomolecular crystallography. It is envisioned that in future Amber can be used as a platform to address various crystallography-related problems, e.g. to refine crystallographic structures of proteins and nucleic acids (similar to the existing capability in the area of biomolecular NMR). This module is intended for use with an MD simulation of the crystal unit cell or a "supercell"[225]. For information on how to set up a crystal simulation, including periodic boundary conditions to emulate crystalline lattice, see Chapter 10.2. It is expected that the crystal is solvated using an explicit (or implicit) solvent. While a number of crystallographic concepts are implemented in the new Amber module, many others have not yet been implemented. For example, an MD model of a crystal unit cell can naturally accommodate different side-chain conformations; however, the concept of alternate side-chain conformations, as employed in protein crystallography, is currently unavailable in Amber.

Although it is not a part of Amber, it is worth noting that the *phenix* crystallographic package now allows for X-ray refinement using Amber (or other) force fields[226]. This was accomplished by using the python API to *sander*, and uses locally-enhanced sampling to handle alternate conformations. It supports all of the X-ray related options in *phenix.refine*, but has limited options for molecular dynamics, and no GPU acceleration.

10.3.1 Structure factor calculations

For the crystal simulation, the program can calculate crystallographic structure factors (SFs) for individual MD frames. The calculations are conducted using direct summation formula[227]; a mask is available to define the subset of atoms included in these calculations (*atom_selection_mask*). For example, this mask could select the macromolecules, but not the solvent or the neutralizing ions present in the simulation. The B-factors used in the direct summation formula are supplied through a designated PDB file. The set of Miller indices for SF calculations is supplied as a part of the *reflection_infile*.

In principle, explicit solvent and ions can also be accounted for via the direct summation formula. However, any single individual frame does not offer an adequate statistical sampling with regard to the positioning of water molecules (if desired, such statistical sampling can be obtained by modeling of a very large supercell or otherwise by means of time averaging). As a commonly accepted alternative, Amber 22 offers two mask-based models of bulk solvent. The first one (*bulk_solvent_model* = 'simple') is a simple variant of flat mask bulk solvent model[228], which calculates the contribution from interstitial solvent into SFs using two generic parameters (*k_sol* and *b_sol*), as reported by Fokine and Urzhumtsev[229]. The implementation, including the scheme to build solvent mask, is analogous to the one in cctbx library[230]. The more advanced version (*bulk_solvent_model* = 'afonine-2013') employs the varaible k_{mask}, which replaces *k_sol* and *b_sol*; this variable is automatically optimized over the individual resolution bins[231]. The iterative optimization procedure to determine k_{mask} also adjusts the overall scaling coefficient that is applied to calculated SFs. The implementation follows the one in cctbx library with several minor modifications.

It is worth noting that crystal MD simulations in Amber (as described in Chap. 10.2) do not maintain a perfect space group symmetry. Therefore, strictly speaking, the calculated SFs correspond to P(1) space group with the unit cell that is identical to the simulation box. During the course of the simulation, the calculated SFs can be collected frame-by-frame at a specified interval (*ntwsf*) and stored in a form of special trajectory file (*sf_outfile*).

10.3.2 Structure-factor-based potential

The X-ray energy term E_{xray} is added to the total potential energy with the user-specified weight w_{xray} (controlled by *xray_weight_initial* / *xray_weight_final* variables):

$$E_{total} = E_{force-field} + w_{xray} E_{xray} \tag{10.1}$$

E_{xray} uses the set of target (i.e. experimentally observed) SFs, which are supplied via the input file *reflection_infile*. This file must also contain flags to divide all reflections into a "working" set and a "free" (test) set. Currently *pmemd* offers two variants of the E_{xray} term. The first one is a very simple least squares objective function (*target* = 'ls') involving the aplitudes and of calculated and observed structure factors:

$$E_{xray} = \frac{\sum_{h,k,l} \left(F_{calc}(h,k,l) - F_{obs}(h,k,l)\right)^2}{\sum_{h,k,l} F_{obs}^2(h,k,l)} \tag{10.2}$$

The sum in this expression is over the working set of reflections.

The second option for E_{xray} is the Maximum Likelihood target function (*target* = 'ml')[232, 233]:

$$E_{xray} = \sum_{h,k,l} \left(-\ln\left(\frac{2F_{obs}(h,k,l)}{\varepsilon\beta}\right) + \frac{F_{obs}^2(h,k,l)}{\varepsilon\beta} + \frac{\alpha^2 F_{calc}^2(h,k,l)}{\varepsilon\beta} - \ln I_0\left(\frac{2\alpha F_{obs}(h,k,l) F_{calc}(h,k,l)}{\varepsilon\beta}\right)\right) \tag{10.3}$$

where α and β are (resolution-shell-dependent) ML likelihood distribution parameters, ε is the symmetry coefficient ($\varepsilon = 1$ for the space group P(1) at hand), $I_0(x)$ is the zeroth-order modified Bessel function of the first kind, and the summation is over the working set of reflections. For practical applications, we recommend using *target* = 'ml' along with the more advanced version of solvent, *bulk_solvent_model* = 'afonine-2013'.

The expression for E_{xray}, along with the direct-summation formula for F_{calc}, provides a basis to evaluate forces. These "restraint" forces act like those used in NMR refinement, discussed in Chap.9, and are generally used to drive minimization or MD simulations that minimize E_{total}. The value of E_{xray} is reported in the mdout file, together with R_{work} and R_{free}.

We envisage that SF-based restraints can be used for a number of purposes. For example, they can be viewed as an empirical addition to the force fields, which can potentially remedy certain existing biases[234, 235]. Another promising application is refinement of crystallographic structures. Such an Amber-based protocol has been reported by Mikhailovskii et al.[236] (the web interface is available at https://arx.bio-nmr.spbu.ru). Ultimately, the entire process of crystallographic structure determination can be incorporated into Amber. This approach may be particularly valuable for lower-quality diffraction data sets and incomplete structural models (e.g. in the case of weak or missing electron density for mobile side chains, loops or terminal regions in protein molecules). In this situation, the state-of-the-art force field provides a natural solution to model the poorly resolved or unresolved elements of the structure. This is accomplished in a highly realistic manner, by using the explicit representation of the crystal unit cell (supercell), taking into consideration the effect of solvent, crystal contacts, etc.

10.3.3 Inputs and file formats

System setup follows the general procedures outlined in Chap. 10.2. For users with access to the *phenix* package of crystallographic analysis tools, the XrayPrep tool can prepare the system: inputs are simply a PDB file (*xxxx.pdb,* where xxxx is a PDB id) and the corresponding structure factor file (*xxxx-sf.cif*).

For those who will prepare their own inputs, one needs a PDB file, expanded to the unit cell (see Chap. 10.2) that contains the B-factors. The structure factors have to be listed in the r*eflection_infile*, which is a human-readable ascii file containing the same information that can be normally found in .mtz files. The first line contains a total number of reflections followed by a zero. Subsequent lines list Miller indices *h, k* and *l*, followed by the respective SF values and their standard deviations, followed by an R-free flag (we adopt the convention that "1" indicates a member of the working set, and "0" a member of the test set). An example file is given below. Note that column spacing or number formatting is not critical, but each entry should be separated by at least one space.

```
41243 0
-19        -6        1        13.86329        9.685285        0
-19        -6        2        46.38137        3.528763        1
-19        -5        1        9.675193        21.28529        1
...
 19         6        1        13.86329        9.685285        0
 19         6        2        46.38137        3.528763        1
```

Input variables in the &xray namelist The X-ray functionalities are activated by adding the *&xray* namelist to the mdin file. User-assigned parameters that are not used by the algorithm are silently ignored(e.g.\ *k_sol* and *b_sol* in 'afonine-2013' bulk solvent model). The keywords in *&xray* namelist include the following:

File handling:

`pdb_infile`	name of the PDB input file containing B-factors
`pdb_read_coordinates`	if true, use coordinates from the PDB file, not inpcrd, as starting coordinates
`pdb_outfile`	name of PDB file to write the final atomic coordinates from the simulation. Currently writes back the input B-factors and occupancies as read from *pdb_infile*
`reflection_infile`	name of the input file containing experimental SFs and R-flags
`fmtz_outfile`	name of the file with calculated SFs, in "formatted MTZ" format

Bulk solvent parameters:

`bulk_solvent_model`	the type of bulk solvent to use. Possible values: 'none' for disabled bulk solvent contribution, 'simple' or 'afonine-2013' (default)
`k_sol`	solvent electron density (default 0.35 e Å^{-3})
`b_sol`	determines the blurring of the boundary between the solvent region and the macromolecule (default 46 Å^2)
`solvent_mask_adjustment`	increment to be added to atomic radii of the atoms selected by *atom_selection_mask* as a part of the algorithm to build bulk solvent mask (default 1.11 Å)
`solvent_mask_probe_radius`	the radius of solvent probe to apply as a part of the algorithm to build bulk solvent mask (default 0.9 Å)
`mask_update_period`	time interval for bulk solvent grid update (expressed as a multiple of integration step, default 100 steps)

Structure-factor-based potential:

`atom_selection_mask`	ambmask-format mask to specify the atoms that contribute to F_{calc} via direct summation formula (default '!@H=')
`scale_update_period`	time interval to re-scale F_{calc} to F_{obs} (expressed as a multiple of integration step, default 100 steps)
`target`	the type of crystallographic target function. Possible values: 'ls' for Least Squares, 'ml' for Maximum Likelihood (default)
`ml_update_period`	time interval to update ML parameters α and β (expressed as a multiple of integration step, default 100 steps)
`xray_weight`	Sets the value of w_{xray} (default 1.0) This value can be variable: see Section 2.9 for more information.
`xray_weight_final`	final value that defines linear scaling of w_{xray} weight along the trajectory (if unassigned, assumed to be equal to *xray_weight_initial*)

10.4 Auxiliary scripts for X-ray refinement

This section gives a bird's-eye view of capabilities of the some useful auxiliary scripts. The hope is to get more detailed examples and documentation as time permits. The organization is based on the compuational tools required, which almost certainly not the best approach.

10.4.1 ccp4_scripts

These scripts are based on the *CCP4* and *refmac5* programs.

conf_solvent.sh Takes an electron density for the solvent (say from MD simulations or from integral equation models), plus an atomic model for the "solute" (generally a macromolecule like a protein), and refines the model with *refmac5*. As a comparison, ignores the input solvent density and uses the standard *refmac5* bulk solvent contribution, as would be done in a conventional simulation. This offers one way to see if the input density is any "better" than the standard solvent contribution.

md2diffuse.sh Takes snapshots from an MD simulation, and works through the steps to compute both the Bragg intensities and diffuse intensities. The *sfall* program is the key engine to compute scattering amplitudes. Special attention is paid to reducing disk space requirements, so that scattering amplitudes from many snapshots can be stored and analyzed. The code optionally will constuct map coefficients and an electron density map arising from the simulation.

md2map.sh

mdv2map.sh The two scripts compute the average electron density, as a CCP4 map and as map coefficients, from an input MD simulation. Unlike the **md2diffuse.sh** script, which computes the average scattering amplitudes, then uses an FFT to obtain the final map, these two scripts directly add the maps together, then uses a Fourier transform to obtain the map coefficients. Since no diffuse scattering information is needed, there is much less need to optimize disk space; still the overall amount of computation is about the same, *i.e.* that required to run *sfall* on each frame of the input trajectory. The **mdv2map.sh** script assumes that the unit cell dimensions don't change throughout the trajectory, whereas **md2map.sh** does not make this assumption.

10.4.2 phenix_scripts

XrayPrep This script currently has two parts, which can be run together, or separately. The first part uses the *phenix.AmberPrep* script to prepare an input pdb-format file for use in refinement. It usually works well, but is limited in its ability to handle unusual situations, such as post-translational modifications, etc. See the *run_alt_tleap* script below for a more flexible alternative. The second part of the script takes an *xxx-sf-cif* file from the PDB, and converts the information into files needed for amber-based refinement. This uses *phenix.refine* to do conversions, reject outliers, possible convert intensities to structure factors, expand to P1 symmetry, and so on. (Going forward, I will probably remove the first part of this script.)

CryoPrep These two scripts take an atomic model (in the form of a PDB-format file), and either an electron density (as a CCP4 map) or structure factors (as an sf-cif file), and prepare all the files needed to carry out refinements that employ a molecular force field (rather than Engh-Huber like distance and angle restraints) to constrain the molecular geometry.

run_phenix.solvent This is a sample script to allow comparisons of the bulk-solvent model in *phenix* and a user-supplied solvent density. Unlike the **conf_solvent.sh** script based on *refmac5*, this only works for a single-point, and doesn't carry out refinements that include the user-supplied solvent density.

run_phenix_refine This is just an alternative way to run *phenix.refine* from the command line, with flexible ways to choose the input paramters. It's probably not for everyone.

run_fmodel This just runs *phenix.fmodel* and re-formats the output in a simple but occasionally useful way.

10.4.3 rism_scripts

run_rism A simple script to run 3D-rism calcualtions (using the *msander* program) on a single snapshot. Typical values for grid spacing, closure and solvent model are hard-wired here, but you should edit this if you wish to make changes.

run_metatwist_rho Takes the output from *run_rism* and computes an electron density map for the solvent. This can be used to replace the "flat" bulk solvent density model in programs like *msander* or *refmac5*.

10.4.4 ensemble_scripts

These are scripts that are used to run ensemble refinements, where many complete copies of the macromolecule are present, generally arranged in a supercell. These are mainly useful as part of a bigger workflow, whose documentation is not yet ready.

find_alts output a listing of alternate conformations found in an input pdb-file

select_alts this is the main script, that takes an input pdb file with alternate conformations (for example, as found in the PDB). For each chain, it randomly selects an alternate conformer for each residue, based on its input occupation. The output file sets all occupations to 1.0. This is commonly used to create a starting model for ensemble refinement.

modify_pdb is a utility perl script for making all manner of modifications to PDB files, at least that can be done on a line-by-line basis. The scripts reads an input PDB file, makes whatever changes are coded in an intermediate section, and writes out the result.

modify_frac is a variant of modify_pdb that allows modification based on fractional coordinates, useful for repacking coordinates into a single unit cell, and so on.

collapse_pdb takes an ensemble model and converts it into a more standard “alternate conformer” model

10.4.5 XtalAnalyze scripts

This is a set of scripts created by Pawel Janowski some years ago. The main current documentation is in the *README*, *XtalAnalyze.sh* and *XtalPlot.sh* files. Read these for now to see how to run things.

11 LEaP

11.1 Introduction

LEaP is the generic name given to the programs teLeap and xaLeap, which are generally run *via* the *tleap* and *xleap* shell scripts. These two programs share a common command language but the *xleap* program has been enhanced through the addition of an X-windows graphical user interface. The name LEaP is an acronym constructed from the names of the older AMBER software modules it replaces: link, edit, and parm. Thus, LEaP can be used to prepare input for the AMBER molecular mechanics programs.

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

```
Read AMBER PREP input files
Read Amber PARM format parameter sets
Read and write Object File Format files (OFF)
Read and write PDB files
Construct new residues and molecules using simple commands
Link together residues and create nonbonded complexes of molecules
Modify internal coordinates within a molecule
Generate files that contain topology and parameters for AMBER and NAB
```

```
usage: tleap [ -I<dir> ] [ -f <file>|- ]
```

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

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

11.2 Concepts

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

11.2.1 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 PARMSETs. 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:

```
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. (Whitespace is blanks, tabs, and commas; and as of Amber version 21 carriage returns are also treated as 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.

11.2.2 Variables

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

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

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

11.2.3 Objects

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

NUMBERs

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

STRINGs

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

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

LISTs

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

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

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

PARMSETs (Parameter Sets)

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

ATOMs

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

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

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

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

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

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

RESIDUEs

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

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

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

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

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

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

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

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

UNITs

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

head

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

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

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

Examples of setting the above properties are

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

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

Complex objects and accessing subobjects

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

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

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

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

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

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

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

Assuming that the ATOM with the name CA has a sequence number 3 within RESIDUE 1, then both of the above commands will print a description of the 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.

11.3 Running LEaP

```
tleap -h
```

will give a list of command-line arguments (which are very simple). Once you have started either program, typing "help" will display useful information about possible actions.

A file called *leaprc* is executed as a script file at the start of the LEaP session unless the user suppresses it with the *-s* command line option. Sample script files are in *$AMBERHOME/dat/leap/cmd*, and you may wish to copy one of these to become "your" default file. LEaP will look first for a *leaprc* file in the user's current directory, then in any directories included with *-I* flags.

The command line interface allows the user to specify a log file that is used to log all input and output within the command line environment. The log file is named using the *logFile* command. The file has two purposes: to allow the user to see a complete record of operations performed by LEaP, and to help recover from (and recreate) program crashes. Output from LEaP commands is written to the log file at a verbosity level of 2 regardless of the verbosity level set by the user using the *verbosity* command. Each line in the log file that was typed in by the user begins with the two characters "> " (a greater-than sign followed by a space). This allows the user to extract the commands typed into LEaP from the log file to create a script file that can be executed using the *source* command. This provides a type of insurance against program crashes by allowing the user to regenerate their interactive sessions. An example of a command that will create a script to reenact a LEaP session is:

```
cat LOGFILE | grep "^> " | sed "s/^> //" > SOURCEFILE.x
```

11.4 Basic instructions for using LEaP to build molecules

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

11.4.1 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.protein.ff14SB (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 `zMatrix` command to build a molecule from a Z-matrix. For small molecules, e.g., ligand like, `loadMol2` or `loadMol3` are available. See the Commands section below for descriptions of these options. If you do not have a starting structure (in the form of a PDB file), LEaP can be used to build the molecule; you will find, however, that this is not always a straightforward process. Many experienced Amber users turn to other (commercial and non-commercial) programs to create their initial structures.

2. Be very attentive to any errors produced in the `loadPdb` step; these generally mean that LEaP has misread the file. A general rule of thumb is to keep editing your input PDB file until LEaP stops complaining. It is often convenient to use the `addPdbAtomMap` or `addPdbResMap` commands to make systematic changes from the names in your PDB files to those in the Amber topology files; see the leaprc files in *$AMBERHOME/dat/leap/cmd* for examples of this. *Be sure to read Section* **??** *for information on how to "clean up" PDB files before loading them.*

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

11.4.2 Amino Acid Residues

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

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

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

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

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

11.4.3 Nucleic Acid Residues

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

11.5 Error Handling and Reporting

In Amber version 18 changes were made to LEaP's error processing. The first set of changes involve error handling. For input from a file (i.e., `tleap invoked with -f`) execution is now terminated at the first occurrence of these errors: file input/output errors, illegal command syntax, illegal command arguments, and some command parsing errors. The intent is to simplify error detection and to ease troubleshooting. For interactive input there is no change in handling: LEaP continues to be forgiving of these errors in the hope that the user can recover in real time.

The final set of changes involve error reporting. LEaP produces four kinds of messages: errors, warnings, notes, and processing messages. Messages beginning with "Fatal Error!" or "Error!" or "Error:" indicate a serious problem. Messages beginning with "Warning!" or "Warning:" indicate a potential problem that should be investigated. Messages beginning with "Note." or "Note:" provide information worth noting. Messages that are not designated by one of the above tags report processing status. Total counts of errors, warnings, and notes are outputted at the end of LEaP. The intent is to simplify error detection by emitting clear and consistent messages.

As with all computational software, LEaP's output should be carefully examined. Some error and warning messages mention likely causes or contain suggested workarounds, but all such messages provide clues. Apply common sense and the scientific method to troubleshoot. Typical first steps are to verify input files and to search the AMBER Mail Reflector for similar reported problems. Note that LEaP normally produces a log file that contains all messages and more detailed output that can be inspected.

11.5.1 Known Issues

This is an incomplete list, started with Amber version 24, focussing on incompatibilities with standards, significant bugs, and gotchas in *LEaP*:

- LEaP does not write Mol2 files that are compliant with Sybyl bond types in the @<TRIPOS>BOND record. As of Amber version 24, single, double, and triple bond types are correctly emitted; the number 4 is used to denote aromatic bonds. In previous versions, the number 1 was always emitted.

- A workaround exists for the num lock issue in xLEaP; contact the Amber mailing list for details.

Of course other issues should be reported via the Amber mailing list.

11.6 Commands

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

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

11.6.1 add

```
add a b
```

UNIT/RESIDUE/ATOM a,b

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

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

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

11.6.2 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 standard leaprc files. The STRINGs are most safely rendered using quotation marks. If atom types are not defined, confusing messages about hybridization can result when loading PDB files.

11.6.3 addC4Pairwise

```
addC4Pairwise  unit atom1 atom2 C4Value(kcal/mol*A^4)
```

Adds pairwise C4 interaction between two specific atoms. C4 interaction means the non-bonded force between `atom1` and `atom2` will follow 12-6-4 pattern. Note here this 12-6-4 pattern is atom-specific, not atom-type-specific. This is achieved by generating `LENNARD_JONES_DCOEF` and `LENNARD_JONES_DVALUE` flags in the prmtop file. Under `LENNARD_JONES_DCOEF` flag, each row will contain the ID of `atom1`, the ID of `atom2`, and a numeric index for debugging purpose. Under `LENNARD_JONES_DVALUE` flag, each row will contain the `C4Value`. Multiple `addC4Pairwise` commands are supported in a single tleap file.

11.6.4 addC4Type

```
addC4Type  unit type1 type2 C4Value(kcal/mol*A^4)
```

Similar to `add12_6_4` in `ParmEd`, this function will generate a `LENNARD_JONES_CCOEF` flag in the prmtop file. Under that flag, a C4 matrix will be presented with mostly zero values, except for the entries that represent non-zero `C4Value`s between `type1` and `type2`. Multiple `addC4Type` commands are supported in a single tleap file.

11.6.5 addIons and addIons2

```
addIons unit ion1 numIon1 [ion2 numIon2]
addIons2 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, *ion1* must be opposite in charge to *unit* and *ion2* must not be specified. Otherwise, the specified numbers of *ion1* [*ion2*] are added [in alternating order]. If solvent is present, it is ignored in the charge and steric calculations, and if an ion has a steric conflict with a solvent molecule, the ion is moved to the center of that solvent molecule, and the latter is deleted. (To avoid this behavior, either solvate _after_ `addions`, or use `addIons2`.) Ions must be monatomic. This procedure is not guaranteed to globally minimize the electrostatic energy. When neutralizing regular-backbone nucleic acids, the first cations will generally be placed between phosphates, leaving the final two ions to be placed somewhere around the middle of the molecule. The default grid resolution is 1 Å, extending from an inner radius of (*maxIonVdwRadius* + *maxSoluteAtomVdwRadius*) to an outer radius 4 Å beyond. A distance-dependent dielectric is used for speed. `addIons2` is the same as `addIons`, except solvent and solute are treated the same.

Algorithms for determining the number of ions to add, based on a desired salt concentration, are given in Refs. [237, 238].

11.6.6 addIonsRand

```
addIonsRand unit ion1 #ion1 [ion2 #ion2] [separation]
```

Adds counterions in a shell around *unit* by replacing random solvent molecules. If *#ion1* is 0, the unit is neutralized (*ion1* must be opposite in charge to *unit*, and *ion2* cannot be specified). Otherwise, the specified numbers of *ion1* [*ion2*] are added [in alternating order]. If *separation* is specified, ions will be guaranteed to be more than that distance apart in Angstroms.

Ions must be monoatomic. This procedure is much faster than `addIons`, as it does not calculate charges. Solvent must be present. It must be possible to position the requested number of ions with the given separation in the solvent. Algorithms for determining the number of ions to add, based on a desired salt concentration, are given in Refs. [237, 238].

11.6.7 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.

11.6.8 addPdbAtomMap

```
addPdbAtomMap list
```

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

```
{ string string }
```

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

11.6.9 addPdbResMap

```
addPdbResMap list
```

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

```
{ double string1 string2 }
```

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

```
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 } : :
> }
```

11.6.10 alias

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

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

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

11.6.11 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
```

11.6.12 bondByDistance

```
bondByDistance container [ maxBond ]
```

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

```
bondByDistance alkylChain
```

11.6.13 check

```
check unit [ parms ]
```

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

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

11.6.14 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
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>
```

11.6.15 copy

```
newvariable = copy variable
```

In most cases, 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 TIP3PBOX 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 TIP3PBOX 8 2
```

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

Note that in a few instances, the copy command does not produce an exact copy. This is particularly relevant when making copies of oligosaccharide residues. In these, the copy command invariably inverts chirality at the anomeric carbon. The workaround for this is to use the copy command twice, where the second call inverts the chirality back.

11.6.16 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.)

11.6.17 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.)

11.6.18 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.)

11.6.19 deleteBond

```
deleteBond atom1 atom2
```

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

11.6.20 desc

```
desc variable
```

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

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

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

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

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

```
> desc ALA.1.N
ATOM Name: N
Type: N
Charge: -0.463
Element: N
Atom flags: 20000|posfxd- posblt- posdrn- sel- pert- notdisp- tchd-
```

```
      posknwn+ int - nmin- nbld-
Atom position: 3.325770, 1.547909, -0.000002
Atom velocity: 0.000000, 0.000000, 0.000000
Bonded to .R<ALA 1>.A<HN 2> by a single bond.
Bonded to .R<ALA 1>.A<CA 3> by a single bond.
```

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
```

11.6.21 groupSelectedAtoms

```
groupSelectedAtoms unit name
```

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

```
groupSelectedAtoms TRP sideChain
```

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

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

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

11.6.22 help

```
help [string]
```

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

11.6.23 impose

```
impose unit seqlist internals
```

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

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

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

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

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

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

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

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

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

α-D-Glc$p$-(1-6)-β-D-Man$p$-(1-6)-β-D-Gal$p$-OH

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

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

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

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

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

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

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

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

will not affect any inter-residue linkage, but instead will set the C5-C6 torsion in the glucopyranoside (0GA) at the non-reducing end of the oligosaccharide.

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

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

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

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

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

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

will set the torsion between residues 3 and 2.

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

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

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

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

because HB1 is not bonded to N.

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

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

11.6.24 list

List all of the variables currently defined. To illustrate, the following (edited) output shows the variables defined when LEaP is started with a standard leaprc file:

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

11.6.25 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
```

11.6.26 loadAmberPrep

```
loadAmberPrep filename [ prefix ]
```

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

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

11.6.27 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
```

11.6.28 loadMol2

```
variable = loadMol2 filename
```

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

11.6.29 loadPdb

```
variable = loadPdb filename
```

Load a Protein Data Bank (PDB) format file with the file name *filename* into *variable*, a UNIT. The sequence numbers of the RESIDUEs will be determined from the order of residues within the PDB file ATOM records. This function will search the variables currently defined within LEaP for variable names that map to residue names within the ATOM records of the PDB file. If a matching variable name is found then the contents of the variable are added to the UNIT that will contain the structure being loaded from the PDB file. Adding the contents of the matching UNIT into the UNIT being constructed means that the contents of the matching UNIT are copied into the UNIT being built and that a bond is created between the connect0 ATOM of the matching UNIT and the connect1 ATOM of the UNIT being built. (This bond creation does not occur if a PDB 'TER' card separates the atoms. As of AmberTools21 a PDB TER record is also used to detect a new residue in the case of contiguous residues with identical residue sequence numbers.) 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. Note that the standard leaprc files include commands to establish automatic N- and C-termination of amino acid sequences and 5′ and 3′ termination of nucleic acid sequences.

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

11.6.30 loadPdbUsingSeq

```
loadPdbUsingSeq filename unitlist
```

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

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

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

11.6.31 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
```

11.6.32 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 (determining simple angles and dihedral angles are straightforward extensions). 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 ....
> measureGeom ALA.ALA.3 ALA.ALA.CB
Distance: 1.52 angstroms
```

11.6.33 quit

Quit the LEaP program.

11.6.34 remove

```
remove container item
```

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

```
> dipeptide = combine { ALA GLY }
Sequence: ALA
Sequence: GLY
> desc dipeptide
UNIT name: ALA
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
Head atom: .R<ALA 1>.A<N 1>
Tail atom: null
Contents: R<ALA 1>
```

11.6.35 saveAmberParm

```
saveAmberParm unit topologyfilename coordinatefilename
```

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

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

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

11.6.36 saveMol2

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

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

11.6.37 saveOff

```
saveOff object filename
```

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

11.6.38 savePdb

```
savePdb unit filename
```

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

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

Warning: The PDB-like file created with this command is primarily useful for reading back into *tleap*, or for other Amber-related uses. It is consistent with Amber, but not with other aspects of the PDB standard (e.g. in atom and residue names, etc.) Use the *ambpdb* program (see Section **??**) if you need a file that more fully complies with the PDB standard.

11.6.39 sequence

```
variable = sequence list
```

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

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

11.6.40 set

This command operates in two modes. In the first, it sets default values for some parameters. In the second, it sets specific properties to containers (for example, UNITs).

Defaults can be set in LEaP for the global parameters below with this usage:

```
set default parameter value
```

For example:

```
set default PBRadii mbondi
```

OldPrmtopFormat If set to "on", the `saveAmberParm` command will write a prmtop file in the format used in Amber 6 and earlier versions; if set to "off" (the default), it will use the new format. This is discouraged for general use and is available mainly for backwards compatibility with programs that expect old-style topology files or for testing.

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

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

PBRadii Used to choose various sets of atomic radii for generalized Born or Poisson-Boltzmann calculations. Options are: "bondi", which gives values from Ref. [127], which should be used with *igb* = 7; "mbondi", which is the default, and the recommended parameter set for *igb* = 1 [62]; "mbondi2", which is a second modification of the Bondi radii set [45], and should be used with *igb* = 2 or 5; "mbondi3", which is a third modification of the Bondi radii set [56] recommended for use with igb = 8; and "amber6", which is only to be used for reproducing very early calculations that used *igb* = 1 [43].

nocenter If set to "on", LEaP will not center the coordinates inside the box for a periodic simulation; it will leave them unchanged as it does for a non-periodic simulation (note that the various solvate commands can still rigidly translate a solute). If set to "off" (the default), centering of coordinates will occur (as it always has, in previous versions of LEaP). Avoiding coordinate translations can be useful to avoid changing reference (perhaps experimental) coordinates. This option may be especially helpful for crystal simulations.

reorder_residues If set to "off", residues in the output will be left in the same order they were found in the input file. The default behavior ("on") is to place non-solvent residues first, followed by solvent residues, followed by solvent cap residues (if cap exists). "off" can, for example, be useful in crystal simulations (keep residues belonging to each asymmetric unit separate), but note that turning residue ordering off is untested and may lead to unforeseen behavior. Only set to "off" if you know what you are doing!

The parameters listed below can be set for the specified *containers* within LEaP using the following syntax:

```
set container parameter object
```

Some examples:

```
set ATOM name "name"
set RESIDUE connect0 ATOM
my_system = loadPDB file.pdb
set my_system box {25 30 32}
```

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 This STRING is a unique identifier for an ATOM in its final state during a Free Energy Perturbation calculation. This functionality is no longer implemented in Amber.

pertType This STRING is the AMBER force field atom type of a perturbed ATOM. This functionality is no longer implemented in Amber.

pertCharge This NUMBER represents the final electrostatic point charge on an ATOM during a Free Energy Perturbation. This function is no longer implemented in Amber.

For RESIDUEs:

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

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

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

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

name This STRING property is the RESIDUE name.

For UNITs:

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

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

box This property defines the bounding box of the UNIT (*container*). If *object* is set to null then no bounding box is defined. If it is a single NUMBER, the bounding box will be defined to be a cube with each side being NUMBER Å across. If it is a LIST, it must contain three NUMBERs, the lengths (in Å) of the three sides of the bounding box. Note that this command does not allow one to set the angles for the periodic system. See the `ChBox` command to do that.

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

11.6.41 setBox

```
setBox solute enclosure [ distance ]
```

This command creates a periodic box around *solute*, which should be a UNIT. It does not add any solvent to the system. `setBox` creates a cuboid box. The *enclosure* parameter determines whether the box encloses entire atoms or just atom centers. The former case is specified by the STRING value "vdw" for *enclosure* and the latter case by the STRING "centers". Use "centers" if the system has been previously equilibrated as a periodic box. The minimum distance between any atom in *solute* and the edge of the periodic box is given by the *distance* parameter; see the solvateBox command for more details.

```
> mol = loadpdb my.pdb
> setBox mol "vdw"
```

11.6.42 solvateBox and solvateOct

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

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

If the distance parameter is a single NUMBER then the minimum distance is the same for the x, y, and z directions, unless the STRING "iso" parameter is specified to make the box or truncated octahedron isometric. For solvateBox if "iso" is used, the solute is rotated to orient the principal axes, otherwise it is just centered on the origin. For solvateOct if the "iso" option is used, the isometric truncated octahedron is rotated to an orientation used by the PME code, and the box and angle dimensions output by the saveAmberParm* commands are adjusted for PME code imaging.In solvateBox, if the distance parameter is a LIST of three NUMBERS then the NUMBERs are applied to the x, y, and z axes respectively. As the larger box is created and superimposed on the solute, solvent molecules overlapping the solute are removed.In solvateOct, when a LIST is given for the distance parameter, four numbers are given instead of three, where the fourth is the diagonal clearance. If 0.0 is given as the fourth number, the diagonal clearance resulting from the application of the x,y,z clearances is reported. If a non-0 value is given, this may require scaling up the other clearances, which is also reported. Similarly, if a single NUMBER is given, any scaleup of the x,y,z buffer to accommodate the diagonal clip is reported.

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

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

11.6.43 solvateCap

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

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

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

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

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

```
> mol = loadpdb my.pdb
> solvateCap mol TIP3PBOX mol.2.CA 12.0 0.75
```

11.6.44 solvateShell

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

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

```
> mol = loadpdb my.pdb
> solvateShell mol TIP3PBOX 12.0 0.8
```

11.6.45 source

```
source filename
```

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

11.6.46 transform

```
transform atoms, matrix
```

Transform all of the ATOMs within *atoms* by a symmetry operation. The symmetry operation is represented as a (3 × 3) or (4 × 4) matrix, and given as nine or sixteen NUMBERs in *matrix*, a LIST of LISTs. The general matrix looks like:
r11 r12 r13 -tx r21 r22 r23 -ty r31 r32 r33 -tz 0 0 0 1
The matrix elements represent the intended symmetry operation. For example, a reflection in the (x,y) plane would be produced by the matrix:

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

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

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

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

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

11.6.47 translate

```
translate atoms direction
```

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

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

11.6.48 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
```

11.6.49 zMatrix

```
zMatrix object zmatrix
```

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

```
{ a1 a2 bond12 }
```

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

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

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

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

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

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

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

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

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

12 Antechamber and GAFF

These are a set of tools to generate files for organic molecules and for some metal centers in proteins, which can then be read into LEaP. The Antechamber suite was written by Junmei Wang, and is designed to be used in conjunction with the general AMBER force field (GAFF) (gaff.dat).[239] See Ref. [240] for an explanation of the algorithms used to classify atom and bond types, to assign charges, and to estimate force field parameters that may be missing in gaff.dat. The python Metal Site Modeling Toolbox (pyMSMT) software package was developed by Pengfei Li, and is described in Section**??**.

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 can be HF/6-31G* RESP, AM1-BCC,[241, 242] or ABCG2.[243, 244] The force field parametrization was performed entirely with HF/6-31G* RESP charges. (Note that in AM1-BCC, the QM electrostatic potentials that were used as fitting targets were created in a very slightly different manner and then compared to RESP charges, using different scaling factors (i.e. 0.001/0.01 [242] versus 0.0005/0.001 [245].)

To maintain force field integrity and to achieve the best performance, the following is the overall guidance on choosing force fields and charge models. Both gaff and gaff2 were developed using HF/6-31G* RESP charge, therefore, RESP charge is compatible with both general AMBER force fields. It was found that gaff/bcc combination achieves good performance in many scenarios including solvation free energy calculations. For the efficient charge models, bcc or abcg2, gaff/bcc and gaff2/abcg2 combinations are recommended, while gaff2/bcc or gaff/abcg2 are not.

The van der Waals parameters are the same as those used by the traditional AMBER force fields. The equilibrium bond lengths and bond angles came from *ab initio* calculations at the MP2/6-31G* level and statistics derived from the Cambridge Structural Database. The force constants for bonds and angles were estimated using empirical models, and the parameters in these models were trained using the force field parameters in the traditional AMBER force fields. General torsional angle parameters were extensively applied in order to reduce the huge number of torsional angle parameters to be derived. The force constants and phase angles in the torsional angle parameters were optimized using our PARMSCAN package,[246] with an aim to reproduce the rotational profiles depicted by high-level *ab initio* calculations (geometry optimizations at the MP2/6-31G* level, followed by single point calculations at MP4/6-311G(d,p)).

By design, GAFF is a complete force field (so that missing parameters rarely occur); it covers almost all the organic chemical space that is made up of C, N, O, S, P, H, F, Cl, Br and I. Moreover, GAFF is totally compatible with the AMBER macromolecular force fields. It should be noted that GAFF atom types, except metal types, are in lower case, while AMBER atom types are always in upper case. This feature makes it possible to load both AMBER protein/nucleic acid force fields and GAFF without any conflict. One can even merge the two kinds of force fields into one file. The combined force fields are capable of studying complicated systems that include both proteins/nucleic acids and organic molecules. We believe that the combination of GAFF with AMBER macromolecular force fields will provide a useful molecular mechanical tool for rational drug design, especially in binding free energy calculations and molecular docking studies. Since its introduction, GAFF has been used for a wide range of applications, including ligand docking,[247] bilayer simulations,[248, 249] and the study of pure organic liquids [250]; see also a recent overview of general force fields for small molecules.[251]

12.1 Principal programs

The *antechamber* program itself is the main program of Antechamber. If your molecule falls into any of several fairly broad categories, *antechamber* should be able to process your PDB file directly, generating output files suitable for LEaP. Otherwise, you may provide an input file with connectivity information, i.e., in a format such as Mol2 or SDF. If there are missing parameters after *antechamber* is finished, you may want to run *parmchk2* to generate a frcmod template that will assist you in generating the needed parameters.

12.1.1 antechamber

This is the most important program in the package. It can perform many file conversions, and can also assign atomic charges and atom types. As required by the input, antechamber executes the following programs: *sqm* (or, alternatively, *mopac* or *divcon*), *atomtype*, *am1bcc*, *bondtype*, *espgen*, *respgen* and *prepgen*. It typically produces many intermediate files; these may be recognized by their names, in which all letters are upper-case. If you experience problems while running *antechamber*, you may want to run the individual programs that are described below (to facilitate this run antechamber with the option ’-s 2’).

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
       radius: only read in radius
       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, overrides input file, default is MOL
-rf    residue toplogy file name in prep input file,
       default is molecule.res
-ch    check file name for gaussian, default is 'molecule'
-ek    mopac or sqm keyword, inside quotes; overwrites previous ones
-gk    gaussian job keyword, inside quotes, is ignored when both -gopt and -gsp are used
-gopt  gaussian job keyword for optimization, inside quotes
-gsp   gaussian job keyword for single point calculation, inside quotes
-gm    gaussian memory keyword, inside quotes, such as "%mem=1000MB"
-gn    gaussian number of processors keyword, inside quotes, such as "%nproc=8"
-gdsk  gaussian maximum disk usage keyword, inside quotes, such as "%maxdisk=50GB"
-gv    add keyword to generate gesp file (for Gaussian 09 only)
       1    : yes
       0    : no, the default
-ge    gaussian esp file generated by iop(6/50=1), default is g09.gesp
-tor   torsional angle list, inside a pair of quotes, such as "1-2-3-4:0,5-6-7-8"
       ':1' or ':0' indicates the torsional angle is frozen or not
```

```
-df    am1-bcc precharge flag, 2 - use sqm(default); 0 - use mopac
-at    atom type
       gaff : the default
       gaff2: for gaff2 (beta-version)
       amber: for PARM94/99/99SB
       bcc  : bcc
       abcg2: abcg2
       sybyl: sybyl
-du    fix duplicate atom names: yes(y)[default] or no(n)
-bk    component/block Id, for ccif
-an    adjust atom names: yes(y) or no(n)
       the default is 'y' for 'mol2' and 'ac' and 'n' for the other formats
-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: 0(brief), 1(default) or 2(verbose)
-eq    equalizing atomic charge, default is 1 for '-c resp' and '-c bcc' and 0 for the other charg
       0    : no use
       1    : by atomic paths
       2    : by atomic paths and structural information, i.e. E/Z configurations
-pf    remove intermediate files: yes(y) or no(n)[default]
-pl    maximum path length to determin equivalence of atomic charges for resp and bcc,
       the smaller the value, the faster the algorithm, default is -1 (use full length),
       set this parameter to 10 to 30 if your molecule is big (# atoms >= 100)
-seq   atomic sequence order changable: yes(y)[default] or no(n)
-dr    acdoctor mode: yes(y)[default] or no(n)

-i -o -fi and -fo must appear in command lines and the others are optional

Use 'antechamber -L' to list the supported file formats and charge methods
```

List of the File Formats:

```
file format type  abbre. index | file format type abbre. index
--------------------------------------------------------------
Antechamber        ac       1  | Sybyl Mol2         mol2    2
PDB                pdb      3  | Modified PDB       mpdb    4
AMBER PREP (int)   prepi    5  | AMBER PREP (car)   prepc   6
Gaussian Z-Matrix  gzmat    7  | Gaussian Cartesian gcrt    8
Mopac Internal     mopint   9  | Mopac Cartesian    mopcrt 10
Gaussian Output    gout    11  | Mopac Output       mopout 12
Alchemy            alc     13  | CSD                csd    14
MDL                mdl     15  | Hyper              hin    16
AMBER Restart      rst     17  | Jaguar Cartesian   jcrt   18
Jaguar Z-Matrix    jzmat   19  | Jaguar Output      jout   20
Divcon Input       divcrt  21  | Divcon Output      divout 22
SQM Input          sqmcrt  23  | SQM Output         sqmout 24
Charmm             charmm  25  | Gaussian ESP       gesp   26
Component cif      ccif    27  | GAMESS dat         gamess 28
Orca input         orcinp  29  | Orca output        orcout 30
pdbqt              pdbqt   31  |
--------------------------------------------------------------
```

AMBER restart file can only be read in as additional file

List of the Charge Methods:

```
charge method     abbre.  index | charge method      abbre.
----------------------------------------------------------------
RESP              resp      1   |  AM1-BCC           bcc     2
CM1               cm1       3   |  CM2               cm2     4
ESP (Kollman)     esp       5   |  Mulliken          mul     6
Gasteiger         gas       7   |  ABCG2             abcg2   8
Read in charge    rc        9   |  Write out charge  wc     10
Delete Charge     dc       11   |
----------------------------------------------------------------
```

Examples:

The basic use of *antechamber* is to pick input and output files and formats (via the `-i, -fi, -o, -fo` flags) , and choose various options for charge models, atom types, etc. A typical use would be:

```
antechamber -i my.pdb -fi pdb -o my.mol2 -fo mol2 -c bcc -nc 1
```

The only "tricky" part is in generating resp charges, which requires interacting with the Gaussian program, and which varies depending on the version:

```
Using Gaussian 98 files as input:
(1) antechamber -i g98.out -fi gout -o sustiva_resp.mol2 -fo mol2 -c resp -eq 2
(2) antechamber -i g98.out -fi gout -o sustiva_cm2.mol2 -fo mol2 -c cm2

Using Gaussian03 files as input:
(11)antechamber -i g03.out -fi gout -o mtx.mol2 -fo mol2 -c resp
      -a mtx.pdb -fa pdb -ao name

Using Gaussian09 (version b1 and beyond):
(12)antechamber -i ch3I.mol2 -fi mol2 -o gcrt.com -fo gcrt -gv 1 -ge ch3I.gesp
    run Gaussian09 with gcrt.com as input
    antechamber -i ch3I.gesp -fi gesp -o ch3I_resp.mol2 -fo mol2 -c resp -eq 2
```

The following is the detailed explanations of some flags

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

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

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

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

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

-at This flag is used to specify whether atom types are to be created for the GAFF force field or for atom types consistent with parm94.dat and parm99.dat (i.e., the AMBER force fields). If you are using *antechamber* to create a modified residue for use with the standard AMBER parm94/parm99 force fields, you should set this flag to "amber"; if you are looking at a more arbitrary molecule, set it to "gaff", even if the molecule is intended for use as a ligand bound to a macromolecule described by the AMBER force fields.

-j This flag instructs the program how to run 'bondtype' and 'atom type'. '-j 1' assumes the bond types already exists; '-j 4' first predicts the connectivity table, then assigns bond and atom types sequentially; '-j 5' reads in connectivity table from the input and then run 'bondtype' and 'atomtype' sequentially. In most situations, '-j 4', the default option, is recommended. However, '-j 5' should be used if the input structure is not good enough and it includes the bond connectivity information (such as mol2, mdl, gzmat, etc.)

-eq This flag specifies how to do charge equilibration. With '-eq 1', atomic charge equilibration is predicted only by atom paths, in another word, if two or more atoms have exactly same sets of atom paths, they are equivalent and their charges are forced to be same. While '-eq 2' predicts charge equilibration using both atom paths and some geometrical information (E/Z configuration). With the '-eq 2' option, the charges of two hydrogen atoms bonded to the No 2 carbon of chloroethene are different as they adopt different configurations to chlorine (one is cis and the other is trans). Similarly, the two amide hydrogen atoms of acetamide do not share the same partial charge as the amide bond cannot rotate freely. To back-compatible to the older versions, the default is set to '1'

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

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

12.1.2 parmchk2

parmchk2 reads in an ac/mol2/prepi/prepc file, an atomtype similarity index file (the default is $AMBERHOME/dat/antechamber/PARMCHK.DAT) as well as a force field file (the default is *$AMBERHOME/dat/leap/parm/gaff.dat*). It writes out a force field modification (frcmod) file containing any force field parameters that are needed for the molecule but not supplied by the force field (*.dat) file. Problematic parameters, if any, are indicated in the frcmod file with the note, "ATTN, need revision", and are typically given values of zero. This can cause fatal terminations of programs that later use a resulting prmtop file; for example, a zero value for the periodicity of the torsional barrier of a dihedral parameter will be fatal in many cases. For each atom type, an atom type corresponding file (ATCOR.DAT) lists its replaceable general atom types. By default, only the missing parameters are written to the frcmod file. When the "-a" switch is given the value "Y", *parmchk2* prints out all force field parameters used by the input molecule, whether they are already in the parm file or not. This file can be used to prepare the frcmod file used by thermodynamic integration calculations using sander.

Unlike *parmchk* which only checks several substitutions for a missing force field parameter, *parmchk2* enumerates all the possible substitutions and select the one with the best similarity score as the final substitute. Moreover, a penalty score, which measures the similarity between the missing force field parameter and the substitute is provided. The similarity socres are calculated using the similarity indexes defined in the atom type similarity index file (PARMCHK.DAT). A similarity index of a pair of atom types ('A/B') for a specific force field parameter type was generated by calculating the average percent absolute error of two set of force field parameters in gaff. The two set of force field parameters are identical except that one set has atom type 'A' and the other has 'B'. Each atom type pair ('A/B') has nine similarity indexes for nine different types of force field parameters, which are bond equilibrium length, bond stretching force constant, bond equilibrium angle ('A' and 'B' are central atoms), bond angle bending force constant ('A' and 'B' are central atoms), bond equilibrium angle ('A' and 'B' are non-central atoms), bond angle bending force constant ('A' and 'B' are non-central atoms), torsional angle twisting force constant ('A' and 'B' are inner side atoms), torsional angle twisting force constant ('A' and 'B' are outter side atoms), and improper dihedral angle.

```
parmchk2 -i    input file name
         -o    frcmod file name
         -f    input file format (prepi, prepc, ac, mol2, frcmod, leaplog)
         -s    ff parm set, it is suppressed by "-p" option
               1 or gaff:    gaff (the default)
               2 or gaff2:   gaff2
               3 or parm99:  parm99
               4 or parm10:  parm10
               5 or lipid14: lipid14
         -frc  frcmod files to be loaded, the supported frcmods include
               ff99SB, ff14SB, ff03 for proteins , bsc1, ol15, ol3 for DNA and yil for
               eg. ff14SB+bsc1+yil, ff99SB+bsc1
         -p    parmfile, suppress '-s' flag, optional
         -pf   parmfile format
               1: for amber FF data file (the default)
               2: for additional force field parameter file
         -afrc additional frcmod file, no matter using -p or not, optional
         -c    atom type corresponding score file, default is PARMCHK.DAT
         -atc  additional atom type corresponding score file, optional
               type 'parmchk2 -l' to learn details
         -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'
         -fc   option of force constant calculation for '-f frcmod' or '-f leaplog'
               1: default behavior (the default option)
               2: do empirical calculation before using corresponding atom types
         -att  for the frcmod input format, option of performing parmchk
               1: for all parameters (the default)
               2: only for those with ATTN
```

Example:

```
parmchk2 -i sustiva.prep -f prepi -o frcmod
```

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

12.2 A simple example for antechamber

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

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

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

```
ATOM      1  CG  TP      1      -1.959   0.102   0.795
ATOM      2  CD1 TP      1      -1.249   0.602  -0.303
ATOM      3  CD2 TP      1      -2.071   0.865   1.963
ATOM      4  CE1 TP      1      -0.646   1.863  -0.234
ATOM      5  C6  TP      1      -1.472   2.129   2.031
ATOM      6  CZ  TP      1      -0.759   2.627   0.934
ATOM      7  HE2 TP      1      -1.558   2.719   2.931
ATOM      8  S15 TP      1      -2.782   0.365   3.060
ATOM      9  H19 TP      1      -3.541   0.979   3.274
ATOM     10  H29 TP      1      -0.787  -0.043  -0.938
ATOM     11  H30 TP      1       0.373   2.045  -0.784
ATOM     12  H31 TP      1      -0.092   3.578   0.781
ATOM     13  H32 TP      1      -2.379  -0.916   0.901
```

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

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

The output file will look like this:

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

@<TRIPOS>BOND
     1    1    2 ar
     2    1    3 ar
     3    1   13 1
     4    2    4 ar
     5    2   10 1
     6    3    5 ar
     7    3    8 1
     8    4    6 ar
     9    4   11 1
    10    5    6 ar
    11    5    7 1
    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 parmchk2 to see if all of the needed force field parameters are available:

```
parmchk2 -i tp.mol2 -f mol2 -o frcmod
```

This yields the frcmod file:

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

In this case, there were two missing dihedral parameters from the gaff.dat file, which were assigned a default value. (As gaff.dat continues to be developed, there should be fewer and fewer missing parameters to be estimated by parmchk2.) In rare cases, parmchk2 may be unable to make a good estimate; it will then insert a placeholder (with zeros everywhere) into the frcmod file, with the comment "ATTN: needs revision". After manually editing this to take care of the elements that "need revision", you are ready to read this residue into LEaP, either as a residue on its own, or as part of a larger system. The following LEaP input file (leap.in) will just create a system with thiophenol in it:

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

You can read this into LEaP as follows:

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

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

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

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

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

1. For the input molecules, make sure there are no open valences and the structures are reasonable. All hydrogen atoms must be present. Antechamber doesn't know what to do with metal ions (see the MCPB.py program for that), or for other non-organic elements such as Boron. Look at the *$AMBERHOME/dat/leap/parm/gaff.dat* file to see what sorts of atomic environments are supported.

2. The Antechamber package produces two kinds of messages: error messages and informative messages. Informative messages begin with "Info:" and may be safely ignored, but they may be helpful for understanding and troubleshooting antechamber. For example: "Info: Bond types are assigned for valence state 1 with penalty of 1". Messages beginning with "Fatal Error!" or "Error:" indicate a problem. Some such messages may mention likely causes or contain suggested workarounds, but all such messages provide clues. Apply common sense and the scientific method to troubleshoot. Typical first steps are to verify input files and to search the AMBER Mail Reflector for similar reported problems. Additional steps are described below.

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. *acdoctor* can diagnose many possible problems with input molecules. If you encounter failures when running antechamber programs, it is highly recommended to let *acdoctor* perform a diagnosis. Run the acdoctor program or use the acdoctor mode in program antechamber; the latter is controlled by option '-dr' and is on by default.

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=0.0005, 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 increase the value of *grms_tol*. If you see failures in scf convergence that are not fixed by changing scfconv, try adding setting *ndiis_attempts*=700. You can use the `-ek` flag to antechamber to change these: for example

   ```
   antechamber ....  -ek "qm_theory='AM1', grms_tol=0.0005, scfconv=1.d-8, ndiis_attempts=700,"
   ....
   ```

   But be aware that there may be something "wrong" with your molecule if these problems arise; *acdoctor* may help (see the previous tip).

7. The standard procedure for obtaining AM1-BCC charges calls for a geometry optimization first. [241, 242] For some molecules (especially anions like phosphates) such a vacuum minimization may be inappropriate, since it can lead to formation of intramolecular hydrogen bonds that are not representative of the expected conformations in solution. If you trust your initial geometries, you can add *maxcyc*=0 to the `-ek` flag to skip the geometry minimization. You might also want to turn off geometry optimization in order try out several conformations in order to assess the sensitivity of the AM1-BCC charges to input geometry.

12.3 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.

12.3.1 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 atom type set, suppressed by "-d" option
            gaff  : the default
            amber : for PARM94/99/99SB
            bcc   : for AM1-BCC
            abcg2 : for ABCG2
            gas   : for Gasteiger charge
            sybyl : for atom types used in sybyl
         -d atom type definition file, optional
         -a do post atom type adjustment (it is applied with "-d" option)
            1: yes, 0: no (the default)
```

Example:

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

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

12.3.2 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, BCCPARM.DAT is in $AMBERHOME/dat/antechamber) is read in. An ac file with AM1-BCC charges [241, 242] is written out. Be sure the charges in the input ac file are AM1-Mulliken charges.

```
am1bcc -i input file name in ac format
       -o output file name
       -f output file format(pdb or ac, optional, default is ac)
```

```
        -t am1bcc type, default is 'bcc', can also be 'abcg2'
        -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:

```
    am1bcc -i comp1.ac -o comp1_bcc.ac -f ac -j 4
```

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

12.3.3 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 [241, 242] 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
```

Examples can be found in *$AMBERHOME/test/antechamber/bondtype* and *$AMBERHOME/test/antechamber/chemokine*.

12.3.4 prepgen

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

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

Examples:

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

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

12.3.5 espgen

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

```
espgen -i   input file name
       -o   output file name
       -f   input format:
            1  Gaussian log file(default)
            2  Gaussian ESP file
            3  Gamess ESP file
       -p   generate esp for pGM:
            0  no, the default)
            1  yes
       -dq  print out dipole and quadrupole moments:
            0  no, the default)
            1  yes
       -re  print out remark line
            0  no, the default)
            1  yes
```

Example:

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

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

12.3.6 respgen

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

```
-i input file name(ac)
-o output file name
-l maximum path length (default is -1, i.e. the path can be any long)
-f output file format
   resp1 - first stage resp fitting
   resp2 - second stage resp fitting
   iresp1 - first stage i_resp fitting
   iresp2 - second stage i_resp fitting
   resp3 - one-stage resp fitting
   resp4 - calculating ESP from point charges
   resp5 - no-equalization
-e equalizing atomic charge (default is 1)
   0 not use
   1 by atomic paths
   2 by atomic paths and geometry (such as E/Z configuration)
-a additional input data (predefined charges, atom groups etc)
-n number of conformations (default is 1)
-w weight of charge constraint
   the default values are 0.0005 for resp1/iresp1 and 0.001 for
   resp2/iresp2
```

The following is a sample of additional respgen input file

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

Example:

```
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
respgen -i acetamide.ac -o acetamide.respin1 -f resp1 -e 2
respgen -i acetamide.ac -o acetamide.respin2 -f resp2 -e 2
```

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

12.4 Miscellaneous programs

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

12.4.1 acdoctor

acdoctor reads the same input file formats used by the*antechamber* program and ’diagnoses’ potential issues that can cause antechamber to fail. In AmberTools version 17 the acdoctor functionality was added to program antechamber; it is controlled by option ’-dr’ and is on by default. The first step is to validate some commonly-used molecular formats, such as pdb, mol2, mdl (sdf), etc. Then the presence of any unusual elements (elements other than C, O, N, S, P, H, F, Cl, Br and I) is reported; in AmberTools version 19 the unusual elements check was changed from a warning to a fatal error; please contact the Amber Mail Reflector specifying the unusual element(s) to register your interest in using antechamber on those element(s). Unfilled valences are reported and additional checks are performed when atom types and/or bond types are read for file formats ac, mol2, sdf, prepi, prepc, mdl, alc and hin. The geometry is quantified by a distance matrix and atomic clashes are reported. *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 for those bonds that fail to meet the standard as well as for weird bonds. Next*acdoctor* determines whether all atoms are linked together through atomic paths. If not, an error message is printed. This kind of error typically implies 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 program suite failures.

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

Example:

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

The program reads test.mol2 and checks for potential problems when running the Antechamber programs. Errors and warning messages are printed. (Possible file formats are listed above in Section 12.1.1.

12.4.2 parmcal

parmcal is an interactive program to calculate the bond length and bond angle parameters, according to the rules outlined in Ref. [239].

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

12.4.3 residuegen

It can be painful to prepare a modified amino acid or nucleotide; the complication is that a residue is not a free standing molecule, and needs to be capped with extra atoms, usually at both termini. For "simple" systems, where a single conformation can be used to estimate partial charges, the *prepgen* program described above with the "-m" flag to specify which atoms to keep in the final residue. For more complex circumstances, the *residuegen* facilitates residue topology generation. *residuegen* reads in an input file and applies a set of antechamber programs to generate residue topologies in prepi 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 no more than 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
```

12.4.4 match

The match program was developed to conduct least-square fittings for two molecules (one input and one reference) which are not necessarily the same in structure. Users can specify which atom or residue in the input corresponds to which in the reference in the definition file (-df). The users can also specify which atoms participating the fitting (-ds). The match matrix can be saved for translating and roating those atoms not participating the fitting procedure in separate step using '-j 2'.

```
Usage: match  -i  input file name
              -r  reference file name
              -f  format: 1-pdb (the default), 2-ac, 3-mol2, 4-sdf, 5-crd/rst
              -o  output file name
              -l  run log file name, default is "match.log"
              -s  selection mode
                  0: use all atoms (the default)
                  1: specify atom names
                  2: use atom definition file
                  3: use residue definition file - original residue IDs
                  4: use residue definition file - renumbered residue IDs
              -ds definition string if selection modes of '1' or '3' or '4'
                  e.g. 'C,N,O,CA', or 'HET' which stands for heavy atoms for '-ds 1')
              -df definition file if selection mode of '2' or '3' or '4'
                  records take a form of 'ATOM atom_id_input atom_id_reference'
                  or 'RES res_id_input res_id_reference'
              -n  number of atoms participating ls-fitting,
                  default is -1, which implies to use all the selected atoms
              -m  matrix file, default is "match.matrix"
              -t  job type:
                  0: calculate rms only, need -i and -r
                  1: lsfit, need -i, -r and -o the default
                  2: translation/rotation, need -i, -o and -m
```

Example:

```
match -f pdb -r 1be9.pdb -i 3pdz.pdb -o 3pdz_aligned.pdb -s 4 -ds "CA,C,N,O" -df 3pdz_
```

The program runs least-square fitting for the non-hydrogen main chain atoms of residues defined in the 3pdz_1be9.corr. A part of the 3pdz_1be9.corr is shown below:

```
RES 34 35 G G
RES 35 36 I I
RES 36 37 Y F
...
RES 87 88 L I
RES 88 89 L I
```

12.4.5 match_atomname

One limitation of the Antechamber package is that the atom name information is lost after running Gaussian calculations. And a residue topology file in prepi or prepc or a mol2 file generated from the Gaussian output has atom names not matching those from the original file (usually a pdb file). Because of this glitch, one can not simply load the residue topology file to tleap, read in the pdb file and then to save the topolgoy. We developed match_atomname to address this problem. The match_atomname program takes an input file and a reference file in pdb, ac, prepi, prepc and mol2 format, automatically detects the corresponding atom name in the reference for each atom name in the input. An output file in the same format as that of the input is generated using the matched atom names.

```
Usage: match_atomname -i  input file name
                      -fi input format (pdb, ac, prepi, prepc, mol2
                      -r  ref file name
                      -fr ref format (pdb, ac, prepi, prepc, mol2)
                      -o  output file name
                      -h  include hydrogen atoms or not
                          0 not, the default
                          1 yes
                      -g  geometric info (such as E/Z configuration) is considered t
                          0 no, the default
                          1 yes
                      -l  maximum path length, default is -1 (full length)
                          if it takes very long time and/or core dump occur, a value
```

Example:

```
match_atomname -i SAH.prepi -fi prepi -o SAH_matched.prepi -r SAH_XRAY.pdb -fr pdb
```

The output, SAH_matched.prepi and SAH_XRAY.pdb can be loaded to tleap directly to generate a topology for minimization or MD simulations.

13 paramfit

Robin Betz

The *paramfit* program allows specific forcefield parameters to be optimized or created by fitting to quantum energy data. *Paramfit* can be used when parameters are missing in the default force fields and *antechamber* cannot find a replacement, or when existing parameters do not describe the system to the desired level of accuracy, such as for dihedral constants on protein backbones.

Paramfit attempts to make the following statement true: **With the correct AMBER parameters, calculations performed at a quantum level over many conformations of a structures should match those calculated by AMBER.**

Paramfit can calculate the energy of each conformation and/or the force on each atom, and adjust the force field parameters so that these values correspond to input quantum data.

For energies, *Paramfit* attempts to fit the AMBER energy to the quantum energy for a variety of conformations of the input structure, minimizing the equation

$$\sum_{n=1}^{N} w_i \left[\left(E_{MM}(n) - E_{QM}(n) \right)^2 + K \right] = 0$$

where K is a constant that adjusts for different origins in the QM and MM calculations so that minimization may be done to zero and N is the number of molecular conformations that are considered.

For forces, the equation that is optimized is

$$\sum_{n=1}^{N} \sum_{atom=1}^{N_{atoms}} w_i \left| F(n, atom)_{MM} - F(n, atom)_{QM} \right|^2 = 0$$

where the sum of the differences in the forces on each atom should match given the correct set of parameters. Individual structures can be assigned weights $w_i$to give them more or less relative importance in the fit. By default, all weights are set to 1.

The program works by altering the parameters that AMBER uses to describe the molecule, which alter the elements in the AMBER sum that is used to calculate the energy or forces. It is necessary to evaluate over many conformations of the molecule because the parameters should predict how the molecule will behave dynamically rather than statically. To get a good idea of the forces on a dihedral, for example, the energy needs to be evaluated for multiple conformations of the dihedral to see how it changes each time. *Paramfit* will fit so that the energy changes that AMBER predicts will happen when the dihedral twists match the changes predicted with quantum methods.

In order to facilitate force field development, *Paramfit* supports fitting parameters across multiple molecules (for example, fitting a single dihedral backbone term across a variety of input amino acids). Single molecule fits can also be done to generate parameters that are missing or inadequate to describe small molecules or ligands.

Paramfit provides functionality for the majority of steps in the fitting process, including writing input files for quantum packages, specifying which parameters are to be fit, determining the value of K for the system, and finally conducting the fit and saving it in a force field modification file that can be used by other programs. An external quantum program is needed to generate the energies needed for *paramfit* to conduct a fitting. Currently, the program is capable of writing input files for ADF, GAMESS, and Gaussian, although if you write your own input files instead of using *paramfit's* functionality, any quantum package will work.

Paramfit has OpenMP support for parallelization of the AMBER function evaluation over the input conformations, where each core will evaluate the energy for a subset of the conformations. Enable this by adding the *-openmp* option to configure and rebuilding *paramfit*. By default all available cores will be used. To change this, set the `OMP_NUM_THREADS` environment variable to the number of threads to be executed. You will see a speedup directly proportional to the number of cores you are running.

Paramfit now includes several ways fitting functions to aid in parameter generation. It can fit such that the energy of each input structure matches the single-point quantum energies inputted, or can now do the same fitting only with the forces on each atom, which may produce a more accurate fit that is less sensitive to problems with the input structure, and can also fit all dihedral force constants and phases simultaneously to a small set of quantum energies using a method developed by Chad Hopkins and Adrian Roitberg. This method fits every term and requires fewer function evaluations than running the full minimization algorithm, but requires especially good sampling of each torsion angle of interest.

Fitting forces requires several additional options to specify the location of the output forces files in the job control file. The easiest way to create a job control file for any of these options is to use the wizard, which runs automatically when no job control file is specified. This will walk you through the creation of a job control file and write it for you while prompting for all necessary options for the selected fitting function.

It is highly recommended that you fit to single-point quantum energies, as fitting to forces is considerably more expensive in terms of required calculation and still somewhat experimental. The implementation of the dihedral fitting method is requires a varied set of input structures, and does not allow specifying individual dihedrals to be fit. No matter which method is used, please take care to carefully validate all parameters for reasonableness– *paramfit's* fit is dependent on the variation and quality of the input structures and the resulting parameters are not guaranteed in ill-defined areas of the input conformation set. For example, if you fit a dihedral torsion term with input structures sampling the 0-30 degree range of that dihedral, the resulting parameters cannot be expected to give a valid energy of a structure with the dihedral at 90 degrees, as the algorithm merely fits to the available data and cannot make other predictions.

13.1 Usage

Paramfit is called from the command line as follows for a single molecule fit:

```
paramfit -i Job_Control.in -p prmtop -c mdcrd -q QM_data.dat \
-v MEDIUM --random-seed seed
```

Running *paramfit* without any options will run a wizard that assists in the creation of a job control file. It is highly recommended that you use the wizard to assist you in setting run options.

The following switches apply to single molecule fits only:

-p prmtop The molecular topology file for the structure.

-c mdcrd A coordinate file containing many conformations of the input structure. These may be generated by running a short simulation in solution, or by manually specifying coordinates for each atom. It is important that there be a good representation of the solution space for any parameters that are to be optimized– for example, if you want a bond force constant it would be a good idea to have input structures with a good range of values for the length of the that bond type. See Subsection 13.2.6

-q QM_data.dat A file containing the quantum energies of the structures in the coordinate file, in order, one per line. You will have to extract the energies from the output files that the quantum package produces. An example script to do this for Gaussian formatted output files can be found in $AMBERHOME/AmberTools/src/paramfit/scripts.

To fit multiple molecules, the following switches are used:

```
paramfit -i Job_Control.in -pf prmtop_list -cf mdcrd_list -v MEDIUM --random-seed se
```

Here is a very brief description of the command-line arguments for a multiple molecule fit. For more information on conducting these, fits, please see 13.3.

-pf prmtop_list A file containing a plain-text list of input topology files and the adjustment constant K for each file separated by a space, one per line.

-cf mdcrd_list A file containing a plain text list of input coordinate files, number of structures to read from each file, and directory containing quantum output from each file, separated by a space. These should be specified in the same order as the topologies in the prmtop_list.

The following switches apply to either type of fit:

-i Job_Control.in The job control file for the program. See Section 13.2 for a description of the options and format for this file. If no job control file is specified, a wizard will be initiated that will prompt you for options and help create the file. Use of the wizard is highly recommended when running *Paramfit* for the first time.

-v MEDIUM The verbosity level to run the program at, either LOW, MEDIUM, or HIGH.

–random-seed seed The integer seed for the random number generator. Only specify this parameter when exactly reproducible results are needed for debugging.

13.2 The Job Control File

Similarly to *sander* and other programs, *paramfit* requires a job control file that specifies individual options for each run. The options that apply to your run vary depending on the runtype and the other settings, and they are quite numerous. To aid you in creating a job control file, a wizard has been included that will prompt you about applicable settings and create the job control file for you. Using the wizard is highly recommended, especially when running a fit for the first time. To use the wizard, simply run *paramfit* without any options. **It is highly recommended that you use the wizard to create job control files,** as it prompts for all options relevant to your run and the resulting file can then be easily edited by hand.

The format consists of variable assignments, in the format variable=value, with one assignment per line. Pound signs (#) will comment out lines. See the following sections for a description of what to put in the job control file for various tasks:

13.2.1 General options

paramfit requires several options be set for every run. These variables should usually appear in your job control file.

RUNTYPE Specifies whether this run will be creating quantum input files, setting parameters, or conducting a fit.

- **= CREATE_INPUT** The structures in the coordinate file will be written out as individual input files for a quantum package. See 13.2.2.
- **= SET_PARAMS** Provides an interactive prompt allowing you to specify which parameters will be fit for this molecule. See 13.2.3.
- **= FIT** Conducts a fitting using one of the two minimization algorithms. See 13.2.4for other options that need to be specified.

NSTRUCTURES Specifies how many structures are in the input coordinate file. If this value is less than the total number of structures in the file, only the first **n** will be read. Only applies to single molecule fits! If you are fitting multiple molecules at once, the number of structures for each molecule should be specified in the mdcrd_list file as described in 13.3.

13.2.2 Creating quantum input files

Given a trajectory, *Paramfit* can write input files for a variety of quantum packages. This is necessary to generate the energy values for each input conformation that *Paramfit* will fit to. You do not necessarily need to do this step and can write your own input files if desired. Currently Gaussian, ADF, and GAMESS formats are supported.

Job files will be named sequentially with filename prefix and suffix specified in the job control file. Once all the input files are written, you must run the quantum package yourself*.* *Paramfit* can read Gaussian output files directly, but for other packages you must extract the energies yourself into a file with one energy per line in the same order as the input structures.

Currently *Paramfit* only supports Gaussian if you are fitting forces, and will read the output files and extract the force information for you. See 13.4 for more information on fitting these.

To enter this mode, set RUNTYPE=CREATE_INPUT and specify the following options in your job control file:

QMHEADER File that will be prepended to all created input files for the quantum program. This specifies things on a per-system basis, such as choice of basis set, amount of memory to use, etc. These parameters will vary depending on which quantum package you are using. Sample header files for all supported quantum packages are included in example_config_files in *paramfit's* source directory.

QMFILEFORMAT Specifies which quantum package the created input files should be formatted for.

- **= ADF** Use the Amsterdam Density Functional Theory package.
- **= GAMESS** Use the General Atomic and Molecular Electronic Structure System (GAMESS).
- **= GAUSSIAN** Use Gaussian.

QM_SYSTEM_CHARGE The integral charge of the system. Defaults to 0. Note that some quantum packages may require this to also be specified in your header file.

QM_SYSTEM_MULTIPLICITY The integral multiplicity of the system. Defaults to 1 (singlet).

QMFILEOUTSTART The prefix for each of the created input files. Defaults to 'Job.' The structure number and then the suffix will be appended to this value.

QMFILEOUTEND The suffix for each of the created input files. Defaults to '.in'. With both default options, the file will be named Job.**n**.in.

13.2.3 Specifying parameters

In order to facilitate batch runs as well as simplify the process of running *paramfit* on larger systems, the parameters to be fit are saved and then loaded in during actual fitting so that they do not have to be specified every time. The parameter setting runtype accomplishes this by prompting whether you would like to fit bond, angle and/or dihedral parameters and then displaying a list of the specific atom types for each so that you can pick exactly what *paramfit* should optimize. This saved file does not specify a value for any of the parameters, but simply indicates which ones are to be changed during fitting.

If you do not wish to save a parameter file, you may instead fit a default set of parameters or be prompted every time. See Subsection 13.2.4.

To enter this mode, set `RUNTYPE=SET_PARAMS` and the following options:

PARAMETER_FILE_NAME Specifies the name of a file in which to store the parameters. When loading these parameters in during a fitting, this line will stay the same. Do not modify this file by hand: *paramfit* numbers each bond, angle, and dihedral in a manner that is consistent but not human-readable.

13.2.4 Fitting options

The fitting function accomplishes the actual parameter modification. It does this by minimizing the least squares difference between the quantum energy and the energy calculated with the AMBER equation over all of the input conformations. For a perfect fit, this means that over all structures, $E_{MD} - E_{QM} + K = 0$.

K is the intrinsic difference between the quantum and the classical energies, which is represented as a parameter that is also fit. The value of K depends on the system, and should be fit once as the only parameter before fitting any other parameters.

To enter this mode, set `RUNTYPE=FIT` and set the following additional variables:

ALGORITHM The minimization algorithm to use. *paramfit* currently implements a genetic algorithm and a simplex algorithm for conduction minimization. Each algorithm requires several parameters and is suited to different problems. Please see 13.2.5 for descriptions of these options and a guide on choosing the appropriate algorithm.

- **= GENETIC**
- **= SIMPLEX**
- **= BOTH** Runs the hybrid genetic algorithm followed by the simplex algorithm to fine tune results
- **= NONE** No fit is performed- useful for calculating energy of each structure with the initial parameters to see their quality

FUNC_TO_FIT The fitting function to use in the calculation.

- **= SUM_SQUARES_AMBER_STANDARD** Standard fit to single-point energies. Recommended selection.
- **= AMBER_FORCES** Fit to the forces on atoms involved in fitted parameters. Currently only supports Gaussian output. See Section 13.4 for details.
- **= DIHEDRAL_LEAST_SQUARES** Use Chad Hopkins and Adrian Roitberg's method to fit all dihedral terms at once. This method will fit all dihedral torsion terms simultaneously with a minimal number of function evaluations, but requires very good sampling of the relevant torsion angles.

K The intrinsic difference between the quantum and classical energies. This value needs to be determined once for each system so that the algorithm can minimize to zero instead of to a constant. See Subsection 13.5.2 for an example.

PARAMETERS_TO_FIT Sets how *paramfit* determines which parameters are to be fit. *paramfit* does not fit electrostatics, but is capable of fitting every other element of the AMBER sum, which include bond harmonic force constant and equilibrium length, angle harmonic force constant and equilibrium angle, and proper and improper dihedral barrier height, phase shift, and periodicity. As a general rule, the fewer parameters there are to fit, the faster and more accurate the results will be. Avoid fitting more parameters than necessary.

- **= DEFAULT** Fit all bond force constants and lengths, angle force constants and sizes, and dihedral force constants. This option will usually fit a very large number of parameters, and is rarely necessary. For most cases, only a few parameters are desired, and they should be fit individually.
- **= K_ONLY** Do not fit any force field parameters. Only fit the value of K (the difference between quantum and classical energies for the system). This needs to be done once per system in order to determine K before any other parameters are fit, as attempting to fit it at the same time results in inaccurate results. Since small changes in K produce a great change in the overall least squares sum, the algorithm will tend to focus on changing the value of K and will neglect the parameters.

= LOAD The list of parameters to be fit is contained in a file that was previously created with the parameter setting runtype. Set PARAMETER_FILE_NAME to the location of this file. To create this file, run *paramfit* with `RUNTYPE=SET_PARAMS`.

SCEE The value by which to scale 1-4 electrostatics for the AMBER sum. Defaults to 1.2

SCNB The value by which to scale 1-4 van der Waals for the AMBER sum. Defaults to 2.0.

QM_ENERGY_UNITS The unit of energy in the quantum data file if you are fitting to energies. This will depend on your quantum package and settings used for the single point calculations.

= HARTREE Default

= KCALMOL

= KJMOL

QM_FORCE_UNITS The unit of force in the quantum data files if you are fitting to forces. This will depend on your quantum package and settings used for the force calculations.

= HARTREE_BOHR Default

= KCALMOL_ANGSTROM

WRITE_ENERGY Saves the final AMBER energy and the quantum data for each structure to the specified file. Plotting these data is useful in verifying the results of the fitting and identifying any problem structures. See Subsection 13.5.3 for more on how to verify the accuracy of results.

WRITE_FRCMOD When the fitting is complete, the parameters will be saved in a force field modification file at this location in addition to displaying them in standard output. This file may be used with LEaP to create a new *prmtop.* If no value is specified the file will not be created.

SCATTERPLOTS Creates graphs of the bond, angles, and dihedrals found in the input files for each parameter that is being fit. These plots can be visualized using *scripts/scatterplots.sh* found in *paramfit's* source directory. This can be helpful in assessing the quality of the input conformations. No need to specify anything after the = sign for this parameter.

SORT_MDCRDS = YES Sorts the input structures in order of increasing energy before conducting the fit. This can aid in identification of problem regions for the initial or fitted parameters, as they may be generally worse on structures in certain energy ranges.

= NO Default

COORDINATE_FORMAT The format of the input coordinate set. *Paramfit* will return an error if the file is in an unexpected format.

= TRAJECTORY Default

= RESTART

13.2.5 Algorithm options

Paramfit implements two minimization algorithms: a simplex and a hybrid simplex-genetic algorithm (GA). The current version of *paramfit* incorporates numerous refinements to the genetic algorithm that require much less input from the user– it is no longer necessary to choose between the simplex or GA. This improved algorithm means that iterative fits are no longer necessary, and the algorithm will converge very close to or at the global minimum on a single run.

The genetic algorithm starts with a randomly generated solution set, which it recombines and alters in ways similar to evolution. The GA will start with many initial randomly generated sets of parameters. It will then determine which are the best by evaluating the AMBER sum, select them for recombination to produce a new set of parameters, randomly alter a few parameters slightly to prevent premature convergence, and iterate. Once several "generations" have passed without improvement, a loosely converging simplex algorithm is run on a random subset of the population, which is then allowed to recover for several generations before further simplex iterations are conducted. This hybrid approach dramatically speeds convergence to the global minimum, while maintaining the strengths of the genetic algorithm in searching a large, complex solution space with low sampling.

The following options in the job control file will control the behavior of the genetic algorithm. In general the default values for these options is sufficient to produce good results, and alterations to them will speed convergence. Options marked *internal algorithm parameter* should not need to be altered by the vast majority of users, as they are already set to their optimum. The algorithm's results should be independent of these values if they are within reasonable ranges (run the wizard for suggestions).

OPTIMIZATIONS The integer number of possible optimizations the algorithm will use. Analogous to the population size in evolution; larger values require more function evaluations and are slower but produce better initial sampling, and smaller ones will delay convergence. Defaults to 50.

SEARCH_SPACE If positive, the algorithm will search for new parameters for everything except dihedral phases within this percentage of the original value, where 1.0 will search within $\pm$100% of the value found in the input *prmtop*. Defaults to searching over the entire range of valid values and ignoring the original value in the topology file. You may wish to alter this value if you know that the original parameters are good and you wish to search in their neighborhood.

MAX_GENERATIONS The maximum number of iterations the algorithm is allowed to run before it returns the best non-converged optimization. Defaults to 50,000. If you find that you repeatedly need to increase this value compared to the default, there are likely significant problems with your system or insufficient input structures.

GENERATIONS_TO_SIMPLEX The number of iterations in a row that must pass without improvement in the best parameter set for simplex refinements to be run on a random 5% of the populations. Set to 0 for a pure genetic algorithm. Smaller values will speed convergence but may result in retrieval of local minima. Defaults to 10.

GENERATIONS_WITHOUT_SIMPLEX The number of generations that must pass between runs of simplex refinement, regardless of improvement in the best parameter set. These iterations serve as a recovery period for the population of the genetic algorithm, and allows time for the simplex results to be incorporated. If set to small or zero values, simplex refinement may run too often, resulting in convergence to a local minima and eliminating the global search properties of the genetic algorithm. Defaults to 10.

GENERATIONS_TO_CONV The number of iterations in a row that must pass without improvement in the best parameter set for the algorithm to be considered converged. Set to a larger value for a longer but potentially more accurate run. Defaults to 50, which is too large for most systems. This counter increments along with the counter to trigger simplex refinement, and at the global minimum simplex refinement will produce no improvement on the population, allowing convergence.

MUTATION_RATE *Internal algorithm parameter* The chance an allele (potential parameter) in the genetic algorithm population has to be randomly set to a new value each generation. Defaults to 0.05.

PARENT_PERCENT *Internal algorithm parameter* The percentage of each generation that is allowed to pass on alleles to the next generation. Defaults to 0.25.

The simplex algorithm is excellent at refining a good set of input parameters, but can converge on physically unreasonable values (such as negative bond force constants) if given a naive guess. For this reason, the genetic algorithm is recommended for finding the global minimum or a close approximation thereof, and the simplex algorithm may be run on the resulting parameters to confirm the results, if desired. The simplex algorithm starts at an initial set of parameters and moves "downhill" iteratively while sampling neighboring areas (much like an amoeba crawling along the function landscape), and converges when the improvement from one step to another becomes negligible. The simplex algorithm is generally faster than the GA, and excels at well-defined systems with a small number of dimensions. This algorithm requires a very well-defined sample space, and the input structures should contain a good range over all the bonds, angles, and dihedrals that are to be optimized. Otherwise, the algorithm tends to wander and will converge in badly defined areas of the sample set. In smaller, well-defined systems with only a few parameters, this algorithm will outperform the genetic algorithm.

Choose the simplex algorithm if you wish to fit only a few parameters and have a large number of input conformations. You may specify the following options to fine-tune the step sizes taken, but for the vast majority of cases the defaults should suffice:

BONDFC_dx Intrinsic length of parameter space for minimization. Used to determine the size of the steps to construct the initial simplex. Should be large enough that the steps sample a sufficiently large area but small enough to not move outside of normal parameter range. Bond force constant step size defaults to 5.0.

BONDEQ_dx Bond equilibrium length step size. Defaults to 0.02.

ANGLEFC_dx Angle force constant step size. Defaults to 1.0.

ANGLEEQ_dx Angle equilibrium step size. Defaults to 0.05.

DIHEDRALBH_dx Dihedral force constant step size. Defaults to 0.2.

DIHEDRALN_dx Dihedral periodicity step size. Defaults to 0.01.

DIHEDRALG_dx Dihedral phase step size. Defaults to 0.05.

K_dx Step size for intrinsic difference constant. Defaults to 10.0.

CONV_LIMIT Floating point number that details the convergence limit for the minimization. The smaller the number, the longer the algorithm will take to converge but the results may be more accurate. Defaults to 1.0E-15, which is very strict.

13.2.6 Bounds Checking

In order to ensure that the algorithms can return meaningful results, bounds checking routines are included in *paramfit*. The bounds checking functionality ensures that the algorithm's results are reasonable given the initial sample set, and also makes sure that the sample set is well-defined.

Since bonds and angles are approximately harmonic, the algorithm's result is reasonable if it lies within a well-defined area of the sample set. Bonds and angle values are therefore checked after the algorithm has finished running. In order to properly fit dihedrals, sample structures should span the entire range of phases for each dihedral that is to be fit. Dihedral checking is therefore accomplished before the algorithm begins to conduct the fit.

Bounds checking defaults to halting execution of the program upon reaching a failing condition. It is not recommended that this behavior be disabled, since the results of the fit are most likely inaccurate. Using the fitted parameters anyway will probably result in an inaccurate depiction of the molecule. Properly represented parameters in the input structures are crucial for a valid fit. Instead of using the parameters, fix the input structures so that data are provided in the missing ranges, which will be stated in the error message, and rerun the program twice: first in CREATE_INPUT mode to obtain quantum energies for the added structures and then in FIT mode to redo the fit.

If you **know** that your input structures describe the parameters to be fit quite well, the selectivity of the bounds checking can be altered by the specifying the following options in the job control file. Use these options with caution, and verify the generated parameters carefully.

CHECK_BOUNDS = ON The recommended and default option. This will halt execution when the bounds check fails.

= WARN Continue upon reaching a bounds failure condition, but output a warning. Do not use the parameters generated by this fit without careful verification! Use the error message and other results to determine if they are reasonable.

BOND_LIMIT Fitting results for bond lengths that are this many Angstroms away from the closest approximation in the input structures will result in a failing condition. Defaults to 0.1.

ANGLE_LIMIT Fitting results for angles that are more than this many radians away from the closes approximation in the input structures will result in a failing condition. Defaults to 0.05π.

DIHEDRAL_SPAN The entire range of valid dihedral angles, 0 to π, for each dihedral that is to be fit should be spanned by this many input structure values, otherwise a failing condition will result. Defaults to 12, meaning that there needs to be a dihedral in every $\frac{\pi}{12}$ radian interval of the valid range.

13.3 Multiple molecule fits

Paramfit supports fitting one or more parameters across multiple molecules, and contains several features to aid in force field development. The program is invoked differently, using a prmtop list and mdcrd list that specify topology and structures for each molecule to fit. Since the value of K is also system-dependent, you will need to fit K for each molecule individually.

Input topologies are specified in a *prmtop list,* which contains the filename of each topology and the value of K for that system, separated by a space. There are no comments permitted in this file. For example:

```
molecule1.prmtop 50.0
molecule2.prmtop 100.0
```

To obtain the value of K for each topology file, conduct a single-molecule fit using all the structures corresponding to that topology and put the resulting value in this file. This enables fitting to zero over multiple molecules.

Input coordinate files are stated in the *coordinate list,* which contains the filename of each coordinate set, the number of structures contained in it, and the filename containing the energy of each structure, separated by a space. Each energy file is exactly the same as single-molecule fits, containing the energy of each structure, one per line, in the same order as the corresponding coordinate file. If there are more structures available in the coordinate file than the number N specified, the first N structures will be used in the fit. An example coordinate list would be:

```
molecule1.mdcrd 200 energy1.dat
molecule2.mdcrd 100 energy2.dat
```

Parameters to fit must be present in all of the available topologies, and the parameter specification file (PARAMETER_FILE_NAME) should be created using a single-molecule invocation of *paramfit.* Saved output files such as energy profile will be named according to the input file name, and a single frcmod will be written if specified. A multiple molecule invocation of *paramfit* uses the following command line options:

```
paramfit -i Job_Control.in -pf prmtop_list -cf mdcrd_list [-v MEDIUM] [--random-seed
```

The only alteration to the job control file necessary for multiple molecule fits is the deletion of the NSTRUCTURES parameter. NSTRUCTURES should not be specified as it is now ambiguous and will result in a program error.

13.4 Fitting Forces

Paramfit can fit to the forces on each atom within an input structure rather than to single point energies. In theory, this provides more data to the fitting algorithm and reduces noise by considering only the forces on atoms involved in a fitted parameter in the function evaluation. This section will walk you through the process of fitting forces using *paramfit.*

Currently, force fitting can only read in Gaussian output files, so input files will be created in the format accepted by that program. Specify in the QMHEADER file the “force” keyword, so Gaussian will print out the forces on each atom, and run *paramfit* in the CREATE_INPUT mode as normal. Then run Gaussian on those input files, keeping the resulting output with the same naming scheme, for example appending “.out” to the name of an input file to indicate its input. For example, in bash:

```
for i in `ls output/Job.*.gjf`; do g09 < $i > $i.out; done
```

To run a fit with forces, you must specify the following options in the job control file, or use the wizard. *Paramfit* will read in the output files from the Gaussian job using the same order and naming scheme, so alter the QM filename parameters so that they match the suffix you appended to Gaussian output files.

```
# Enable force fitting function
FUNC_TO_FIT=AMBER_FORCES
# K irrelevant for force fitting
K=0.0
# Force units used by Gaussian
QM_FORCE_UNITS=HARTREE_BOHR
# Naming scheme of gaussian output files
QMFILEOUTSTART=output/Job.
QMFILEOUTEND=.gjf.out
```

Specify parameters to fit, algorithm and output options as described previously for fits to energy.

As forces fitting is still experimental, take care to evaluate the resulting parameters.

13.5 Examples

13.5.1 Setting up to fit

The fitting process with *paramfit* follows a specific order. Example job control files for each step and a description of the step follow.

First, write a job control file to create the input structures and run *paramfit:*

```
RUNTYPE=CREATE_INPUT
# Trajectory has 50 structures
NSTRUCTURES=50
# Write in Gaussian format
QMFILEFORMAT=GAUSSIAN
# Prepend this file to QM inputs
QMHEADER=Gaussian.header
$AMBERHOME/bin/paramfit -i Job_Control.in -p prmtop -c mdcrd
```

After all 50 input files have been created, run the quantum program on them. Once it’s finished, extract the quantum energies from the output files using the provided script, or write your own. Since the example used Gaussian:

```
$AMBERHOME/AmberTools/src/paramfit/scripts/process_gaussian.x \
output_directory energies.dat
```

Now, or while the quantum jobs are running, since neither the energies nor the structures are needed yet, determine which parameters are to be fit and save them.

```
RUNTYPE=SET_PARAMS
# File to be created
PARAMETER_FILE_NAME=saved_params
$AMBERHOME/bin/paramfit -i Job_Control.in -p prmtop
```

Now the quantum energies to fit have been obtained and the parameters to fit have been set, and the fitting process may begin.

13.5.2 Fitting K

The first step in fitting is determining the value of K for a system. A job control file that will only fit K follows:

```
RUNTYPE=FIT
PARAMETERS_TO_FIT=K_ONLY
# Use the simplex function
FITTING_FUNCTION=SIMPLEX
```

Then,

```
$AMBERHOME/bin/paramfit -i Job_Control.in -p prmtop -c mdcrd -q energies.dat
```

Take this value of K and put it back in the job control file when conducting the actual fit.

```
RUNTYPE=FIT
# Use the parameters specified earlier
PARAMETERS_TO_FIT=LOAD
PARAMETER_FILE_NAME=saved_params
# Genetic algorithm options
FITTING_FUNCTION=GENETIC
OPTIMIZATIONS=500
GENERATIONS_TO_CONV=10
GENERATIONS_TO_SIMPLEX=2
GENERATIONS_WITHOUT_SIMPLEX=5
# Save parameters so they can be read into leap
WRITE_FRCMOD=fitted_params.frcmod
```

And call *paramfit* just as before. This example fit will create a force field modification file that can later be read into *LEaP* to create a new *prmtop* with the modified parameters for the molecule.

13.5.3 Evaluating Results

When using *paramfit*, it is important to verify the accuracy of the fitted parameters for your input structures. The WRITE_ENERGY option in the Job Control file is useful for this. Set it to a filename and *paramfit* will write the final AMBER energy of each structure next to the quantum energy for the same structure in a file that can be easily graphed.

If you have gnuplot, a script has been provided to quickly show each structure's energies. Assuming your energy file is named energy.dat:

```
$AMBERHOME/AmberTools/src/paramfit/scripts/plot_energy.x energy.dat
```

The resulting graph makes the identification of problem structures much easier, and gives a good visualization of the fit. In general, carefully validate parameters generated by *paramfit* against other data before conducting large simulations.

The SCATTERPLOT option in the job control file can also be useful in assessing the quality of the input structures. If this option is set, *paramfit* will dump a variety of data files indicating the value for all fitted bonds, angles, and dihedrals in the input conformations. These data may be visualized if you have the program gnuplot by running the following command in the directory where *paramfit* was run:

```
$AMBERHOME/AmberTools/src/paramfit/scripts/scatterplots.sh
```

The resulting graphs feature different colored points for each bond, angle, and dihedral type that is being fit for each of the input structures. This is useful in evaluating if the results of the fit are reasonable– for example, if the algorithm converges with an equilibrium bond length that is not similar to any of the structures, that parameter may not be accurate.

14 nabc and libsff

14.1 Introduction

The NAB language compiler *nab2c* (which converts NAB source code to C, for subsequent compilation) was written in the 1990's by Tom Macke. The original design idea was to create a "molecular awk": a scripting language for manipulation of (macro-)molecules that would be primarily used to create short scripts to carry out molecular manipulations. It was quickly realized that manipulations like force field minimization would be useful, and the Amber-compatible molecular mechanics routines were added by David Case as *sff*, a "simple force field".

Over the years, *sff* evolved to keep pace with (and in many cases drive) Amber developments involving implicit force fields, including generalized Born, Poisson-Boltzmann and RISM approaches. In keeping with its original motivation, *sff* concentrated on implicit solvation, leaving explicit solvent and periodic simulations to the main Amber programs *sander* and *pmemd*. The *sff* routines were parallelized using both openmp and MPI, and second derivatives of the generalized Born model were added by Russ Brown.[252] Apart from the lack of a GPU implementation, the routines in *sff* are the most general and efficient ones in the Amber package. In particular, *sff* excels at generalized Born simulations on large systems, benefitting from an advanced nonbonded list builder, and from the hierarchical charge partition model described in Section 14.6.

As a first step, we have prepared sample files in `$AMBERHOME/AmberTools/test/nabc`, which illustrate how to use most of the *sff* functionality directly from a stand-alone C driver. The *Makefile* in this directory can guide you through running several sample calculations. Looking at the code, and its comments, along with the header file (`$AMBERHOME/include/sff.h`) should go a long way towards allowing direct integration into C codes, without any reference to the NAB compiler. The rest of this chapter has documentation for *libsff*.

14.2 Basic molecular mechanics routines

```
int readparm( molecule m, string parmfile );
int mme_init( molecule mol, string aexp, string aexp2, point xyz_ref[], string filename );
int mm_options( string opts );
float mme( point xyz[], point grad[], int iter );
float mme_rattle( point xyz[], point grad[], int iter );
int conjgrad( float x[], int n, float fret, float func(), float rmsgrad,
        float dfpred, int maxiter );
int md( int n, int maxstep, point xyz[], point f[], float v[], float func );
int getxv( string filename, int natom, float start_time, float x[], float v[] );
int putxv( string filename, string title, int natom, float start_time,
        float x[], float v[] );
void mm_set_checkpoint( string filename );
```

readparm reads an AMBER parameter-topology file, created by tleap or with other AMBER programs, and sets up a data structure which we call a "parmstruct". This is part of the molecule, but is not directly accessible (yet) to nab programs. You would use this command as an alternative to getpdb_prm(). You need to be sure that the molecule used in the readparm() call has been created by calling getpdb() with a PDB file that has been created by tleap itself (i.e., that has exactly the Amber atoms in the correct order).

As noted above, the readparm() routine is primarily intended for cases where getpdb_prm() fails (i.e., when you need to run *tleap* by hand).

setxyz_from_mol() copies the atomic coordinates of mol to the array xyz. setmol_from_xyz() replaces the atomic coordinates of mol with the contents of xyz. Both return the number of atoms copied with a 0 indicating an error occurred.

The getxv() and putxv() routines read and write non-periodic Amber-style restart files. Velocities are read if present.

The getxyz() and putxyz() routines are used in conjunction with the mm_set_checkpoint() routine to write checkpoint or restart files. The coordinates are written at higher precision than to an AMBER restart file, i.e., with sufficiently high precision to restart even a Newton-Raphson minimization where the error in coordinates may be on the order of10^{-12} . The checkpoint files are written at iteration intervals that are specified by the *nchk* or *nchk2* parameters to the mm_options() routine (see below). The checkpoint file names are determined by the filename string that is passed to mm_set_checkpoint(). If filename contains one or more %d format specifiers, then the file name will be a modification of filename wherein the leftmost %d of filename is replaced by the iteration count. If filename contains no %d format specifier, then the file name will be filename with the iteration count appended on the right.

The mme_init() function must be called after mm_options() and before calls to mme(). It sets up parameters for future force field evaluations, and takes as input an nab molecule. The string aexp is an atom expression that indicates which atoms are to be allowed to move in minimization or dynamics: atoms that do not match aexp will have their positions in the gradient vector set to zero. A NULL atom expression will allow all atoms to move. The second string, aexp2 identifies atoms whose positions are to be restrained to the positions in the array xyz_ref. The strength of this restraint will be given by the wcons variable set in mm_options(). A NULL value for aexp2 will cause all atoms to be constrained. The last parameter to mme_init() is a file name without extension for the output trajectory file. This should be NULL if no output file is desired. NAB writes trajectories in the *netCDF* format, which can be read by *cpptraj*, and either analyzed, or converted to another format. The default netCDF extension of .nc is automatically added to the file name.

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.

keyword	*default*	*meaning*
ntpr	10	Frequency of printing of the energy and its components.
e_debug	0	If nonzero printout additional components of the energy.
gb_debug	0	If nonzero printout information about Born first derivatives.
gb2_debug	0	If nonzero printout information about Born second derivatives.
nchk	10000	Frequency of writing checkpoint file during first derivative calculation, i.e., in the mme() routine.
nchk2	10000	Frequency of writing checkpoint file during second derivative calculation, i.e., in the mme2() routine.
nsnb	25	Frequency at which the non-bonded list is updated.

keyword	*default*	*meaning*
nscm	0	If > 0, remove translational and rotational center-of-mass (COM) motion after every nscm steps. For Langevin dynamics (gamma_ln>0) without HCP (hcp=0), the position of the COM is reset to zero every nscm steps, but the velocities are not affected. With HCP (hcp>0) COM translation and rotation are also removed, with or without Langevin dynamics. It is strongly recommended that this option be used whenever HCP is used.
cut	8.0	Non-bonded cutoff, in angstroms. This parameter is ignored if hcp > 0.
wcons	0.0	Restraint weight for keeping atoms close to their positions in xyz_ref (see *mme_init*).
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 dim=4. If this is nonzero, a penalty function will be added to the bounds-violation energy, which is equal to 0.5 * *k4d* * *w* * *w*, where *w* is the value of the fourth dimensional coordinate.
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; if set to 2, bonds to hydrogens will be constrained; 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 mme_rattle() function to *conjgrad()*.
tautp	999999.	Temperature coupling parameter, in ps. The time constant determines the strength of the weak-coupling ("Berendsen") temperature bath.[253] Set *tautp* 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 *gamma_ln* remains at its default value of zero; if *gamma_ln* is not zero, Langevin dynamics is assumed, as discussed below.
gamma_ln	0.0	Collision frequency for Langevin dynamics, inps^{-1} . Values in the range 2-5ps^{-1} often give acceptable temperature control, while allowing transitions to take place.[12] 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.[254]
temp0	300.0	Target temperature, K.
vlimit	20.0	Maximum absolute value of any component of the velocity vector.
ntpr_md	10	Printing frequency for dynamics information to stdout.
ntwx	0	Frequency for dumping coordinates to traj_file.
zerov	0	If nonzero, then the initial velocities will be set to zero.

keyword	*default*	*meaning*
tempi	0.0	If *zerov*=0 and *tempi*>0, then the initial velocities will be randomly chosen for this temperature. If both *zerov* and *tempi* are zero, the velocities passed into the md() function will be used as the initial velocities; this combination is useful to continue an existing trajectory.
genmass	10.0	The general mass to use for MD if individual masses are not read from a prmtop file; value in amu.
diel	C	Code for the dielectric model. "C" gives a dielectric constant of 1; "R" makes the dielectric constant equal to distance in angstroms; "RL" uses the sigmoidal function of Ramstein & Lavery, PNAS **85**, 7231 (1988); "RL94" is the same thing, but speeded up assuming one is using the Cornell *et al* force field; "R94" is a distance-dependent dielectric, again with speedups that assume the Cornell *et al.* force field.
dielc	1.0	This is the dielectric constant used for *non-GB* simulations. It is implemented in routine mme_init() by scaling all of the charges by sqrt(dielc). This means that you need to set this (if desired) in mm_options() before calling mme_init().
gb	0	If set to 0 then GB is off. Setting gb=1 turns on the Hawkins, Cramer, Truhlar (HCT) form of pairwise generalized Born model for solvation. See ref [62] for details of the implementation; this is equivalent to the *igb=1* option in *sander* and *pmemd*. Set diel to "C" if you use this option. Setting gb=2 turns on the Onufriev, Bashford, Case (OBC) variant of GB,[40, 45] with α=0.8, β=0.0 and γ=2.909. This is equivalent to the *igb=2* option in *sander* and *pmemd*. Setting gb=5 just changes the values of α, β and γ to 1.0, 0.8, and 4.85, respectively, corresponding to the *igb=5* option in *sander*. Setting gb=7 turns on the GB Neck variant of GB,[64] corresponding to the *igb=7* option in *sander* and *pmemd*. Setting gb=8 turns on the updated GB Neck variant of GB, corresponding to the *igb=8* option in *sander* and *pmemd*.
rgbmax	999.0	A maximum value for considering pairs of atoms to contribute to the calculation of the effective Born radii. The default value means that there is effectively no cutoff. Calculations will be sped up by using smaller values, say around 15. Å or so. This parameter is ignored if hcp > 0.
gbsa	0	If set to 1, add a surface-area dependent energy equal to surfen*SASA, where surften is discussed below, and SASA is an approximate surface area term. NAB uses the "LCPO" approximation developed by Weiser, Shenkin, and Still.[33]
surften	0.005	Surface tension (see *gbsa*, above) in kcal/mol/Å^2.
epsext	78.5	Exterior dielectric for generalized Born; interior dielectric is always 1.

keyword	*default*	*meaning*
kappa	0.0	Inverse of the Debye-Hueckel length, if gb is turned on, in $Å^{-1}$. This parameter is related to the ionic strength as $\kappa = [8\pi\beta I/\varepsilon]^{1/2}$, where I is the ionic strength (same as the salt concentration for a 1-1 salt). For T=298.15 and ε=78.5, $\kappa = (0.10806\, I)^{1/2}$, where I is in [M].
static_arrays	1	If set to 1, do not allocate dynamic arrays for each call to the mme() and mme2() 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.
hcp	0	Use the GB-HCP model: **= 0** No GB-HCP. **= 1** 1-charge approximation. **= 2** 2-charge approximation. **= 4** 2-charge based on optimal point charge approximation (recommended for GB-HCP). See Section 14.6 for detailed instructions on using the GB-HCP. It is strongly recommended that the NSCM option above be used whenever GB-HCP is used.
dhcp	0.25	Adjusts the separation between the charges used to approximate uncharged components for hcp=4. dhcp is empirically determined so that the RMS error in force, compared to GB without further approximation, is minimized. Our testing on various structures suggests that the optimal value for dhcp can be found within the range of 0.1 and 0.4. See Section 14.6 for details.
hcp_h1	15	GB-HCP level 1 threshold distance. The recommended level 1 threshold distance for amino acids is 15A. For structures with nucleic acids the recommended level 1 threshold distance is 21A.
hcp_h2	50	GB-HCP level 2 threshold distance. The recommended level 2 threshold distance for proteins is 50A. For structures with nucleic acids the recommended level 2 threshold distance is 90A.
hcp_h3	150	GB-HCP level 3 threshold distance. The recommended level 3 threshold distance for amino acids is 150A. For structures with nucleic acids the recommended level 1 threshold distance is 169A.

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 *ntpr* variable) and non-bonded updates (see *nsnb*). The mme_rattle() function has the same interface, but constrains the bond lengths and returns a corrected gradient. If you want to minimize with constrained bond lengths, pass *mme_rattle* and not *mme* to the *conjgrad* routine.

The conjgrad() function will carry out conjugate gradient minimization of the function func that depends upon n parameters, whose initial values are in the x array. The function func must be of the form func(x[], g[], iter), where x contains the input values, and the function value is returned through the function call, and its gradient with respect to x through the g array. The iteration number is passed through iter, which func can use for whatever purpose it wants; a typical use would just be to determine when to print results. The input parameter dfpred is the expected drop in the function value on the first iteration; generally only a rough estimate is needed. The minimization will proceed until maxiter steps have been performed, or until the root-mean-square of the components of the gradient is less than rmsgrad. The value of the function at the end of the minimization is returned in the variable fret. conjgrad can return a variety of exit codes:

Return codes for conjgrad routine	
>0	minimization converged; gives number of final iteration
-1	bad line search; probably an error in the relation of the function to its gradient (perhaps from round-off if you push too hard on the minimization).
-2	search direction was uphill
-3	exceeded the maximum number of iterations
-4	could not further reduce function value

Finally, the md function will run maxstep steps of molecular dynamics, using func as the force field (this would typically be set to a function like mme.) The number of dynamical variables is given as input parameter n: this would be 3 times the number of atoms for ordinary cases, but might be different for other force fields or functions. The arrays x[], f[] and v[] hold the coordinates, gradient of the potential, and velocities, respectively, and are updated as the simulation progresses. The method of temperature regulation (if any) is specified by the variables *tautp* and *gamma_ln* that are set in *mm_options()*.

Note: In versions of NAB up to 4.5.2, there was an additional input variable to md() called *minv* that reserved space for the inverse of the masses of the particles; this has now been removed. This change is not backwards compatible: you must modify existing NAB scripts that call md() to remove this variable.

14.3 NetCDF read/write routines

NAB has several routines for reading/writing Amber NetCDF trajectory and restart files. All of the routines except netcdfGetNextFrame() return a 1 on error, 0 on success. The netcdfGetNextFrame() routine returns 0 on error, 1 on success to make it easier to use in loops. For an example of how to use NetCDF files in NAB see the NAB script in **'$AMBERHOME/AmberTools/test/nab/tnetcdf.nab'**.

14.3.1 struct AmberNetcdf

An AmberNetcdf struct must be used to interface with the netcdf commands in NAB (except netcdfWriteRestart()). It contains many fields, but the following are the ones commonly needed by users:

temp0 Temperature of current frame (if temperature is present).

restartTime Simulation time if NetCDF restart.

isNCrestart 0 if trajectory, 1 if restart.

ncframe Number of frames in the file.

currentFrame Current frame number.

ncatom Number of atoms.

ncatom3 Number of coordinates (ncatom * 3).

velocityVID If not -1, velocity information is present.

TempVID If not -1, temperature information is present.

In order to use it, you must include nab_netcdf.h and declare it as a struct, e.g.:

```
#include "nab_netcdf.h"
struct AmberNetcdf NC;
```

14.3.2 netcdfClose

```
int netcdfClose(struct AmberNetcdf NC)
```

Close NetCDF file associated with **NC**.

14.3.3 netcdfCreate

```
int netcdfCreate(struct AmberNetcdf NC, string filename, int natom, int isBox)
```

NC `AmberNetcdf struct to set up.`

filename `Name of file to create.`

natom `Number of atoms in file.`

isBox `0 = No box coordinates, 1 = Has box coordinates.`

Create NetCDF trajectory file and associate with struct **NC**. For writing NetCDF restarts, use netcdfWriteRestart().

14.3.4 netcdfDebug

```
int netcdfDebug(struct AmberNetcdf NC)
```

Print debug information for NetCDF file associated with **NC**.

14.3.5 netcdfGetFrame

```
int netcdfGetFrame(struct AmberNetcdf NC, int set, float X[], float box[])
```

NC `AmberNetcdf struct, previously set up and opened.`

set `Frame number to read.`

X `Array to store coordinates (dimension NC.ncatom3).`

box `Array of dimension 6 to store box coordinates if present (X Y Z ALPHA BETA GAMMA); can be NULL.`

Get coordinates at frame **set** (starting from 0).

14.3.6 netcdfGetNextFrame

```
int netcdfGetNextFrame(struct AmberNetcdf NC, float X[], float box[])
```

NC `AmberNetcdf struct, previously set up and opened.`

X `Array to store coordinates (dimension NC.ncatom3).`

box `Array of size 6 to store box coordinates if present (X Y Z ALPHA BETA GAMMA); can be NULL.`

Get the coordinates at frame **NC.currentFrame** and increment **NC.currentFrame** by one. Unlike the other netcdf routines, this returns 1 on success and 0 on error to make it easy to use in loops.

14.3.7 netcdfGetVelocity

```
int netcdfGetVelocity(struct AmberNetcdf NC, int set, float V[])
```

NC `AmberNetcdf struct, previously set up and opened.`

set `Frame number to read.`

V `Array to store velocities (dimension NC.ncatom3).`

Get velocities at frame **set** (starting from 0).

14.3.8 netcdfInfo

```
int netcdfInfo(struct AmberNetcdf NC)
```

Print information for **NC**, including file type, presence of velocity/box/temperature info, and number of atoms, coordinates, and frames present.

14.3.9 netcdfLoad

```
int netcdfLoad(struct AmberNetcdf NC, string filename)
```

NC `AmberNetcdf struct to set up.`

filename `Name of NetCDF file to load.`

Load NetCDF file filename and set up the AmberNetcdf structure **NC** for reading. The file type is automatically detected.

14.3.10 netcdfWriteFrame

```
int netcdfWriteFrame(struct AmberNetcdf NC, int set, float X[], float box[])
```

NC `AmberNetcdf struct, previously set up and opened.`

set `Frame number to write.`

X `Array of coordinates to write (dimension NC.ncatom3).`

box `Array of size 6 of box coordinates to write (X Y Z ALPHA BETA GAMMA); can be NULL.`

Write to NetCDF trajectory at frame **set** (starting from 0). NOTE: This routine is for writing NetCDF trajectories only; to write NetCDF restarts use netcdfWriteRestart().

14.3.11 netcdfWriteNextFrame

```
int netcdfWriteNextFrame(struct AmberNetcdf NC, float X[], float box[])
```

NC `AmberNetcdf struct, previously set up and opened.`

X `Array of coordinates to write (dimension NC.ncatom3).`

box `Array of size 6 of box coordinates to write (X Y Z ALPHA BETA GAMMA); can be NULL.`

Write coordinates to frame **NC.currentFrame** and increment **NC.currentFrame** by one. NOTE: This routine is for writing NetCDF trajectories only; to write NetCDF restarts use netcdfWriteRestart().

14.3.12 netcdfWriteRestart

```
int netcdfWriteRestart(string filename, int natom, float X[], float V[],
                       float box[], float time, float temperature)
```

filename `Name of NetCDF restart file to create.`

natom `Number of atoms in netcdf restart file.`

X `Array of coordinates to write (dimension natom*3).`

V `Array of velocities to write (dimension natom*3); can be NULL.`

box `Array of size 6 of box coordinates to write (X Y Z ALPHA BETA GAMMA); can be NULL.`

time `Restart time in ps.`

temperature `Restart temperature; if < 0 no temperature will be written.`

14.4 Second derivatives and normal modes

Russ Brown has contributed codes that compute analytically the second derivatives of the Amber functions, including the generalized Born terms.[252] This capability resides in the three functions described here.

```
int newton( float x[], int n, float fret, float func1(), float func2(), float rms,
        float nradd, int maxiter );
float nmode( float x[], int n, float func(), int eigp, int ntrun, float eta, float hrmax, int ioseen );
```

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*natom* vector of coordinates x[], and returns a gradient vector g[], a Hessian matrix, stored columnwise in a *3*natom x 3*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 cannot call mme2() directly, but will pass this as an argument to one of the next two routines.

The newton() routine takes a input coordinates x[] and a size parameter n (must be set to *3*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.[255]

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:

```
molecule m;
float x[4000], fret;

m = getpdb_prm( "mymolecule.pdb", "leaprc.protein.ff14SB", "", 0 );
mm_options( "cut=999., ntpr=50, nsnb=99999, diel=C, gb=1, dielc=1.0" );
mme_init( m, NULL, "::Z", x, NULL);
setxyz_from_mol( m, NULL, x );

//  conjugate gradient minimization
conjgrad(x, 3*m.natoms, fret, mme, 0.1, 0.001, 2000 );

//  Newton-Raphson minimization\fP
mm_options( "ntpr=1" );
newton( x, 3*m.natoms, fret, mme, mme2, 0.00000001, 0.0, 6 );

// get the normal modes:
nmode( x, 3*m.natoms, mme2, 0, 0, 0.0, 0.0, 0);
```

14.5 Low-MODe (LMOD) optimization methods

István Kolossváry has contributed functions, which implement the LMOD methods for minimization, conformational searching, and flexible docking.[136–139] 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().*

There are now **four "real-life" examples** of carrying out LMOD searches: look in *$AMBERHOME/AmberTools/examples/nab/lmod_**. Each directory has a README file that give more information.

14.5.1 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 systems, 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,

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

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 $\bigtriangledown x$ 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.

14.5.2 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 ith (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 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.

14.5.3 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:

Parameter list for xmin()		
keyword	*default*	*meaning*
func	N/A	The name of the function that computes the function value and gradient of the objective function to be minimized. *func()* must have the following argument list: `float func( float x[], float g[], int i)` where `x[]` is the vector of the iterate, `g[]` is the gradient and `i` is currently ignored except when `func` = `mme` where `i` is handled internally.
natm	N/A	Number of atoms. **NOTE:** if `func` is other than `mme`, `natm` is used to pass the total number of variables of the objective function to be minimized. However, `natm` retains its original meaning in case `func` is a user-defined energy function for 3-dimensional (molecular) structure optimization. Make sure that the meaning of `natm` is compatible with the setting of `mol_struct_opt` below.
x[]	N/A	Coordinate vector. User has to allocate memory in calling program and fill x[] with initial coordinates using, e.g., the setxyz_from_mol function (see sample program below). Array size = 3*natm.
g[]	N/A	Gradient vector. User has to allocate memory in calling program. Array size = 3*natm.
ene	N/A	On output, ene stores the minimized energy.
grms_out	N/A	On output, grms_out stores the gradient RMS achieved by XMIN.

Table 14.2: *Arguments for xmin().*

`method`

1: PRCG Polak-Ribiere conjugate gradient method, similar to the *conjgrad()* function [140].

2: L-BFGS Limited-memory Broyden-Fletcher-Goldfarb-Shanno quasi-Newton algorithm [141]. 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 [140, 142]. 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.

4: Debugging option; printing analytical and numerical derivatives for comparison. Almost all failures with *xmin* can be attributed to inaccurate analytical derivatives, e.g., when SCF hasn't converged with a quantum based Hamiltonian.

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.

14.5.4 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.

```
//   XMIN reverse communication external minimization package.
//   Written by Istvan Kolossvary.

```

Parameter list for xmin_opt		
keyword	*default*	*meaning*
mol_struct_opt	1	*1*= 3-dimensional molecular structure optimization. Any other value means general function optimization.
maxiter	1000	Maximum number of iteration steps allowed for XMIN. A value of zero means single point energy calculation, no minimization.
grms_tol	0.05	Gradient RMS threshold below which XMIN should minimize the input structure.
method	3	Minimization algorithm. See text for description.
numdiff	1	Finite difference method used in TNCG for approximating the product of the Hessian matrix and some vector in the conjugate gradient iteration (the same approximation is used in LMOD, see Eq. 14.1 in section 14.5.1). 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. If *print_level* > 2, print minimization output every *print_level* steps
iter	N/A	Output parameter. The total number of iteration steps completed by XMIN.
xmin_time	N/A	Output parameter. CPU time in seconds used by XMIN.
ls_method	2	*1*= modified Armijo [256](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). *Only change it if you know what you are doing.*
c_armijo	0.4	Armijo c parameter, range (0, 0.5). *Only change it if you know what you are doing.*
mu_armijo	1.0	Armijo mu parameter, range [0, 2). *Only change it if you know what you are doing.*
ftol_wolfe	0.0001	Wolfe ftol parameter, range (0, 0.5). *Only change it if you know what you are doing.*
gtol_wolfe	0.9	Wolfe gtol parameter, range (ftol_wolfe, 1). *Only change it if you know what you are doing.*
ls_iter	N/A	Output parameter. The total number of line search steps completed by XMIN.
error_flag	N/A	Output parameter. A nonzero value indicates an error. In case of an error XMIN will always print a descriptive error message.

Table 14.3: *Options for xmin_opt.*

```
#include "xmin_opt.h"

// M A I N   P R O G R A M  to carry out XMIN minimization on a molecule:

struct xmin_opt xo;

molecule mol;
int natm;
float xyz[ dynamic ], grad[ dynamic ];
float energy, grms;
point dummy;

    xmin_opt_init( xo );  //  set up defaults (shown here)

    // xo.mol_struct_opt = 1;
    // xo.maxiter        = 1000;
    // xo.grms_tol       = 0.05;
    // xo.method         = 3;
    // xo.numdiff        = 1;
    // xo.m_lbfgs        = 3;
    //    xo.ls_method   = 2;
    //    xo.ls_maxiter  = 20;
    //    xo.maxatmov    = 0.5;
    //    xo.beta_armijo = 0.5;
    //    xo.c_armijo    = 0.4;
    //    xo.mu_armijo   = 1.0;
    //    xo.ftol_wolfe  = 0.0001;
    //    xo.gtol_wolfe  = 0.9;
    // xo.print_level    = 0;

    xo.maxiter     = 10;  //  non-defaults are here
    xo.grms_tol    = 0.001;
    xo.method      = 3;
    xo.ls_maxatmov = 0.15;
    xo.print_level = 2;

    mol = getpdb( "gbrna.pdb" );
    readparm( mol, "gbrna.prmtop" );
    natm = mol.natoms;
    allocate xyz[ 3*natm ]; allocate grad[ 3*natm ];
    setxyz_from_mol( mol, NULL, xyz );

    mm_options( "ntpr=1, gb=1, kappa=0.10395, rgbmax=99., cut=99.0, diel=C ");
    mme_init( mol, NULL, "::ZZZ", dummy, NULL );

    energy = mme( xyz, grad, 0 );
    energy = xmin( mme, natm, xyz, grad, energy, grms, xo );

// 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:  ntpr=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.

14.5.5 LMOD

```
float lmod( int natm, float x[], float g[], float ene, float conflib[],
       float lmod_traj[], int lig_start[], int lig_end[], int lig_cent[],
       float tr_min[], float tr_max[], float rot_min[], float rot_max[],
       struct xmin_opt, struct xmin_opt, struct lmod_opt);
```

At a glance: The *lmod()* function is similar to *xmin()* in that it optimizes the energy of a molecular structure with initial coordinates given in the x[] array. However, the optimization goes beyond local minimization, it is a sophisticated conformational search procedure. On output, *lmod()* returns the global minimum energy of the LMOD conformational search as the function value and the coordinates in x[] will be updated to the global minimum-energy conformation. Moreover, a set of the best low-energy conformations is also returned in the array conflib[]. Coordinates, energy, and gradient are in NAB units. The parameters are given in the table below; items above the line are passed as parameters; the rest of the parameters are all preceded by "`lo.`", because they are members of an *lmod_opt* struct with that name; see the sample program below to see how this works.

Also note that *xmin()*'s *xmin_opt* struct is passed to *lmod()* as well. *lmod()* changes the default values of some of the "`xo.`" parameters via the call to *lmod_opt_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()*.

keyword	*default*	*meaning*
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 setxyz_from_mol 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 lig_start and lig_end 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.
lig_end[]	N/A	See above.

keyword	*default*	*meaning*
lig_cent[]	N/A	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 = *nlig*, but in case *nlig=0* there is no need to allocate memory.
tr_max[]		See tr_min[], above.
rot_min[]		See tr_min[], above.
rot_max[]		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 14.5.2.). 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 14.5.7, 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.
lmod_restart	10	The frequency, in LMOD iterations, of updating the conflib storage, that is, discarding structures outside the energy window, and restarting LMOD with a randomly chosen structure from the low-energy pool defined by n_best_struct below. A value >maxiter will prevent LMOD from doing any restarts.

keyword	*default*	*meaning*
n_best_struct	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. n_best_struct = 1 allows the user to explore the neighborhood of the then current global minimum.
mc_option	1	The Monte Carlo method. 1= Metropolis Monte Carlo (see rtemp 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 kmod 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.
rtemp	1.5	The value of RT in NAB energy units. This is utilized in the Metropolis criterion.
lmod_step_size_min	2.0	The minimum length of a single LMOD ZIG move in Å. See section 14.5.2.
lmod_step_size_max	5.0	The maximum length of a single LMOD ZIG move in Å. See section 14.5.2.
nof_lmod_steps	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 14.5.2. nof_lmod_steps $>$ 0 means that multiple barriers may be crossed and LMOD can carry the molecule to a large distance on the potential energy surface without severely distorting the geometry.
lmod_relax_grms	1.0	The gradient RMS convergence criterion of structure relaxation, see ZAG move in section 14.5.2.
nlig	0	Number of ligands considered for flexible docking. The default, zero, means no docking.
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.

keyword	*default*	*meaning*
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 nonzero 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 nmod + 1 <= ndim_arnoldi <= 3*natm. The default means that the "small" problem and the "large" problem are identical. This is the preferred, i.e., fastest, calculation for small to medium size systems, because ARPACK is guaranteed to converge in a single iteration. The ARPACK calculation scales with three times the number of atoms times the Arnoldi dimension squared and, therefore, for larger molecules there is an optimal ndim_arnoldi much less than three times the number of atoms that converges much faster in multiple iterations (possibly thousands or tens of thousands of iterations). The key to good performance is to select ndim_arnoldi such that all the ARPACK storage fits in memory. For proteins, ndim_arnoldi =1000 is generally a good value, but often a very small $\sim$50-100 Arnoldi dimension provides the fastest net computational cost with very many iterations.

14.5.6 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.

```
//   LMOD reverse communication external minimization package.
//   Written by Istvan Kolossvary.

#include "xmin_opt.h"
#include "lmod_opt.h"

// M A I N  P R O G R A M  to carry out LMOD simulation on a molecule/complex:

struct xmin_opt xo;
struct lmod_opt lo;

molecule mol;
int   natm;
float energy;
int lig_start[ dynamic ], lig_end[ dynamic ], lig_cent[ dynamic ];
float xyz[ dynamic ], grad[ dynamic ], conflib[ dynamic ], lmod_trajectory[ dynamic ];
float tr_min[ dynamic ], tr_max[ dynamic ], rot_min[ dynamic ], rot_max[ dynamic ];
float glob_min_energy;
point dummy;

     lmod_opt_init( lo, xo );   //  set up defaults

     lo.niter          = 3;     //  non-default options are here
     lo.mc_option      = 2;
     lo.nof_lmod_steps = 5;
     lo.random_seed    = 99;
     lo.print_level    = 2;

     xo.ls_maxatmov    = 0.15;
```

```

    mol = getpdb( "trpcage.pdb" );
    readparm( mol, "trpcage.top" );
    natm = mol.natoms;

    allocate xyz[ 3*natm ]; allocate grad[ 3*natm ];
    allocate conflib[ lo.nconf * 3*natm ];
    allocate lmod_trajectory[ (lo.niter+1) * 3*natm ];
    setxyz_from_mol( mol, NULL, xyz );

    mm_options( "ntpr=5000, gb=0, cut=999.0, nsnb=9999, diel=R ");
    mme_init( mol, NULL, "::ZZZ", dummy, NULL );

    mme( xyz, grad, 1 );
    glob_min_energy = lmod( natm, xyz, grad, energy,
          conflib, lmod_trajectory, lig_start, lig_end, lig_cent,
          tr_min, tr_max, rot_min, rot_max, xo, lo );

    printf( "\nGlob. min. E         = %12.3lf kcal/mol\n", glob_min_energy );


// E N D  M A I N
```

The corresponding screen output should look similar to this.

```
Reading parm file (trpcage.top)
title:

      mm_options:  ntpr=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
```

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 14.5.2). 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 14.5.2) 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 glob. min. energy is printed from the sample NAB program, not from LMOD. The sample program in *$AMBERHOME/AmberTools/examples/nab/lmod_dock* shows how one could write the top ten low-energy structures in separate, numbered PDB files.

As a final note, it is instructive to be aware of a simple safeguard that LMOD applies . A copy of the *conflib[]* array is saved periodically in a binary disk file called *conflib.dat*. Since LMOD searches might run for a long time, in case of a crash low-energy structures can be recovered from this file. The format of *conflib.dat* is as follows. Each conformation is represented by 3 numbers (double energy, double radius of gyration, and int number of times found), followed by the double (x, y, z) coordinates of the atoms.

14.5.7 Tricks of the trade of running LMOD searches

1. The AMBER atom types HO, HW, and ho all have zero van der Waals parameters in all of the AMBER (and some other) force fields. Corresponding Aij and Bij coefficients in the PRMTOP file

are set to zero. This means there is no repulsive wall to prevent two oppositely charged atoms, one being of type HO, HW or ho, to fuse as a result of the ever decreasing electrostatic energy as they come closer and closer to each other. This potential problem is rarely manifest in molecular dynamics simulations, but it presents a nuisance when running LMOD searches. The problem is local minimization, especially "aggressive" TNCG minimization (XMIN xo.method=3) that can easily result in atom fusion. Therefore, before running an LMOD simulation, the PRMTOP file (let's call it prmtop.in) must be processed by running the script "lmodprmtop prmtop.in prmtop.out". This script will replace all the repulsive Aij coefficients set to zero in prmtop.in with a high value of 1e03 in prmtop.out in order to re-create the van der Waals wall. It is understood that this procedure is parameter fudging; however, note that the primary goal of using LMOD is the quick generation of approximate, low-energy structures that can be further refined by high-accuracy MD.

2. LMOD requires that the potential energy surface is continuous everywhere to a great degree. Therefore, always use a distance dependent dielectric constant in mm_options when running searches in vacuo, or use GB solvation (note that GB calculations will be slow), and always apply a large cut-off. It does make sense to run quick and dirty LMOD searches in vacuo to generate low-energy starting structures for MD runs. Note that the most likely symptom of discontinuities causing a problem is when your NAB program utilizing LMOD is grabbing CPU time, but the LMOD search does not seem to progress. This is the result of NaN's that often can be seen when print_level is set to > 0.

3. LMOD is NOT INTENDED to be used with explicit water models and periodic boundary conditions. Although explicit-water solvation representation is not recommended, LMOD docking can be readily used with crystallographic water molecules as ligands.

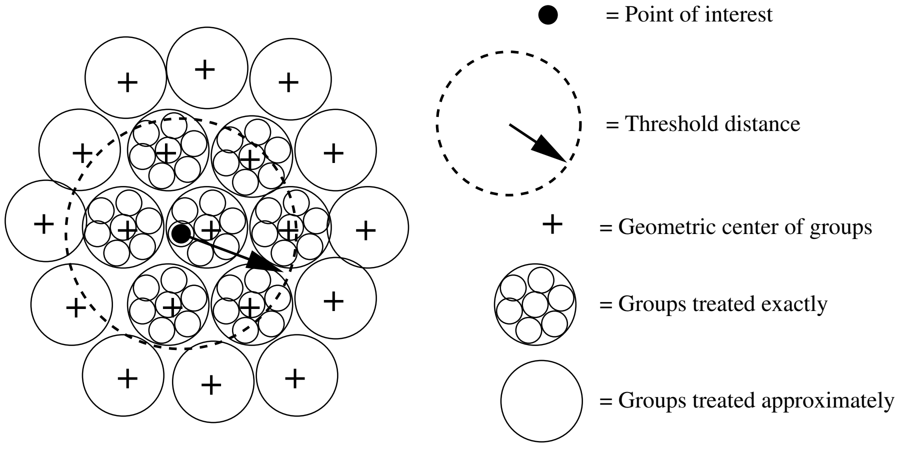
4. Conformations in the conflib and lmod_trajectory files can have very different orientations. One trick to keep them in a common orientation is to restrain the position of, e.g., a single benzene ring. This will ensure that the molecule cannot be translated or rotated as a whole. However, when applying 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. However, the eigenvector calculations are likely to converge faster with the tethered systems.

14.6 The Generalized Born with Hierarchical Charge Partitioning (GB-HCP)

GB-HCP (and its latest version, GB-HCPO[257]) is a multi-scale, yet fully atomistic, approach to perform MD simulations based on the generalized Born model, mainly intended for large and very large structures. For example, it was used to refine a 1.1M atom structure of 30nm chromatin fiber[257]. Compared to the reference GB model without further approximations, GB-HCP can deliver up to 3 orders of magnitude speedup, depending on structure size. In contrast to cutoff GB that completely ignores the effect of long range electrostatic interactions beyond a certain distance, which can lead to serious artifacts under many circumstances such as for highly charged systems, GB-HCP takes into account the

Figure 14.1: *The HCP threshold distance. For the level 1 approximation shown here, groups within the threshold distance are treated exactly using atomic charges, while groups beyond the threshold distance are approximated by a small number of charges, e.g. 1 charge for hcp=1 shown here.*



long range electrostatic interactions by using N log N Hierarchical Charge Partitioning (HCP) approximation [258, 259]. Based on this method, structures are partitioned into multiple hierarchical levels of components using the natural organization of the biomolecular structures - atoms, groups, chains, and complexes. The charge distribution for each of these components is approximated by 1 (hcp=1) or 2 (hcp=2 and hcp=4) charges. Setting hcp=4 (strongly recommended) uses GB-HCPO, which takes advantage of the Optimal Point Charge Approximation approach for placing the approximate point charges[260]: two point charges are placed so that the three lowest order multipole moments of the reference charge distribution are optimally reproduced. The approximate charges are then used for computing electrostatic interactions with distant components while the full set of atomic charges are used for nearby components (Figure 40.1). The HCP can be used for generalized Born (gb=1-8) simulations, for gas phase (dielec=C) and distant dependent dielectric (dielec=R/RL), with or without Langevin dynamics (gamma_ln>0).

The usage of the new feature (hcp=4) requires that the separation between the two charges used to approximate the uncharged components is specified by dhcp. The value of dhcp is empirically adjusted so that the RMS error in force, compared to the GB without further approximation, is minimized. Our testing on a various set of structures suggests that dhcp=0.25 is optimal for many systems. However, if further accuracy is desired for specific systems, the value for dhcp can be further optimized within the range of 0.1 and 0.4 following the steps below. To find the optimal value for hcp, one time step simulation for the starting configuration of the structure can be performed using the GB model without approximation (hcp=0), and with e_debug=1setting, that automatically prints out the forces on each atom into a text file called reference.frc. Rename reference.frc to exact.frc. Then, run one step of the starting configuration of the structure using the GB-HCP (hcp=4) by setting the dhcp parameter within the range of 0.1 and 0.4 in increments of 0.05. The reference.frc file produced for each value of dhcp can be compared to the exact.frc to compute the RMS error in force. The following command line computes the RMS error:

```
paste exact.frc reference.frc | awk '{x+=($9-$20)^2+($10-$21)^2+($11-$22)^2}END{print sqrt(x/NR)}'
```

The optimal value for dhcp is the one that results in minimum RMS error in the force.

14.6.1 Level 1 HCP approximation

The HCP option can now be used with one level of approximation (groups) using NAB molecular dynamics scripts. No additional manipulation of the input structure files is required for one level of approximation. For an example see AmberTools/examples/hcp/2trx.nab. The level 1 approximation is recommended for single domain and small (< 10,000 atoms) multi-domain structures. Speedups of 2x-10x can be realized using the level 1 approximation, depending on structure size.

14.6.2 Level 2 and 3 HCP approximation

For larger multi-domain structures higher levels of approximations (chains and complexes) can be used to achieve up to 3 orders of magnitude speedups, depending on structure size. The following additional steps are required to include information about these higher level components in the prmtop file. For an example see AmberTools/examples/hcp/1kx5.nab. A fully working example (including the MD run scripts) of a 3 level partitioning of a giant structure, one million atom chromatin fiber, can be found at http://people.cs.vt.edu/onufriev/software.php.

1. Ensure the pdb file identifies the higher level structures: Chains (level 2) separated by TER, and Complexes (level 3) separated by REMARK END-OF-COMPLEX:
   ...
   ATOM ...
   TER (end of chain)
   ATOM ...
   ...
   ATOM ...
   TER (end of chain)
   REMARK END-OF-COMPLEX
   ATOM ...
2. Execute hcp_getpdb to generate prmtop entries for HCP: hcp_getpdb pdb-filename hcp-prmtop
3. Concatenate the HCP prmtop entries to the end of the standard prmtop file generated by LEaP: cat prmtop-file hcp-prmtop > new-prmtop
4. Use this new prmtop file in the NAB molecular dynamics scripts instead of the prmtop file generated by LEaP

15 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 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 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.

This chapters introduces the language through a series of sample programs, and illustrate 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. Subsequent chapters give 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.

15.1 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.

15.1.1 Conformation build-up procedures

MC-SYM[261–263] 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.*[264] used a Monte-Carlo build-up procedure based on sets of low energy dinucleotide conformers to construct longer low energy single stranded sequences that would be suitable for incorporation into larger structures. Sets of low energy dinucleotide conformers were created by selecting one value from each of the sterically allowed ranges for the six backbone torsion angles and χ. Instead of an exhaustive build- up search over a small set of conformers, this method samples a much larger region of conformational space by randomly combining members of a larger set of initial conformers. Unlike strict build-up procedures, any member of the initial set is allowed to follow any other member, even if their corresponding torsion angles do not exactly match, a concession to the extreme flexibility of the nucleic acid backbone. A key feature determined the probabilities of the initial conformers so that the probability of each created structure accurately reflected its energy.

Tung and Carter[265, 266] 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.

15.1.2 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.[267] 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,[268] 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.*[269] 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[270] 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.

15.2 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.

15.2.1 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. Details are given in Chap. 17.

nab uses the matrix type to hold a 4×4 transformation matrix. Transformations are applied to residues and molecules to move them into new orientations or positions. nab does *not* require that transformations applied to parts of residues or molecules be chemically valid. It simply transforms the coordinates of the selected atoms leaving it to the user to correct (or ignore) any chemically incorrect geometry caused by the transformation.

Every nab molecule includes a frame, or "handle" that can be used to position two molecules in a generalization of superimposition. Traditionally, when a molecule is superimposed on a reference molecule, the user first forms a correspondence between a set of atoms in the first molecule and another set of atoms in the reference molecule. The superimposition algorithm then determines the transformation that will minimize the rmsd between corresponding atoms. Because superimposition is based on actual atom positions, it requires that the two molecules have a common substructure, and it can only place one molecule on top of another and not at an arbitrary point in space.

The nab frame is a way around these limitations. A frame is composed of three orthonormal vectors originally aligned along the axes of a right handed coordinate frame centered on the origin. nab provides two builtin functions setframe() and setframep() that are used to reposition this frame based on vectors defined by atom expressions or arbitrary 3-D points, respectively. To position two molecules via their frames, the user moves the frames so that when they are superimposed via the nab builtin alignframe(), the two molecules have the desired orientation. This is a generalization of the methods described above for OCL.

15.2.2 Distance geometry

nab's second initial structure-creation method is *metric matrix distance geometry,[271, 272]* 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. Details are given in Chap. 18.

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[273] 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.

15.2.3 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. Details are given in Chap. **??**.

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,[274] calcyclin dimers,[275] HMG-protein/DNA complexes,[276] active sites of Rieske iron-sulfur proteins,[277] and supercoiled DNA.[278] The Examples chapter below provides a number of other sample applications.

15.3 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 (deafult is "a.out" on most operating systems)
-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 nab code would be accomplished by

```
nab -o bar bar.nab foo.c
```

The result is an executable bar file. To run the program, type:

```
./bar <command line options needed go here>
```

15.4 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.

15.5 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.

15.5.1 B-form DNA duplex

One of the goals of nab was that simple models should require simple programs. Here is an nab program that creates a model of a B-form DNA duplex and saves it as a PDB file.

```
1 // Program 1 - Average B-form DNA duplex
2 molecule m;
3
4 m = bdna( "gcgttaacgc" );
5 putpdb( "gcg10.pdb", m );
```

Line 2 is a declaration used to tell the nab compiler that the name m is a molecule variable, something nab programs use to hold structures. Line 4 creates the actual model using the predefined function bdna(). This function's argument is a literal string which represents the sequence of the duplex that is to be created. Here's how bdna() converts this string into a molecule. Each letter stands for one of the four standard bases: a for adenine, c for cytosine, g for guanine and t for thymine. In a standard DNA duplex every adenine is paired with thymine and every cytosine with guanine in an antiparallel double helix. Thus only one strand of the double helix has to be specified. As bdna() reads the string from left to right, it creates one strand from 5′ to 3′ (5′-gcgttaacgc -3′), automatically creating the other antiparallel strand using Watson/Crick pairing. It uses a uniform helical step of 3.38 Å rise and 36.0o twist. Naturally, nab has other ways to create helical molecules with arbitrary helical parameters and even mismatched base pairs, but if you need some "average" DNA, you should be able to get it without having to specify every detail. The last line uses the nab builtin putpdb() to write the newly created duplex to the file gcg10.pdb.

Program 1 is about the smallest nab program that does any real work. Even so, it contains several elements common to almost all nab programs. The two consecutive forward slashes in line 1 introduce a comment which tells the nab compiler to ignore all characters between them and the end of the line. This particular comment begins in column 1, but that is not required as comments may begin in any column. Line 3 is blank. It serves no purpose other than to visually separate the declaration part from the action part. nab input is free format. Runs of white space characters—spaces, tabs, blank lines and page breaks—act like a single space which is required only to separate reserved words like molecule from identifiers like m. Thus white space can be used to increase readability.

15.5.2 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 );
```

```
rmsd( m, "::C1'", mr, "::C1'", r );
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.

15.5.3 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 putpdb( "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

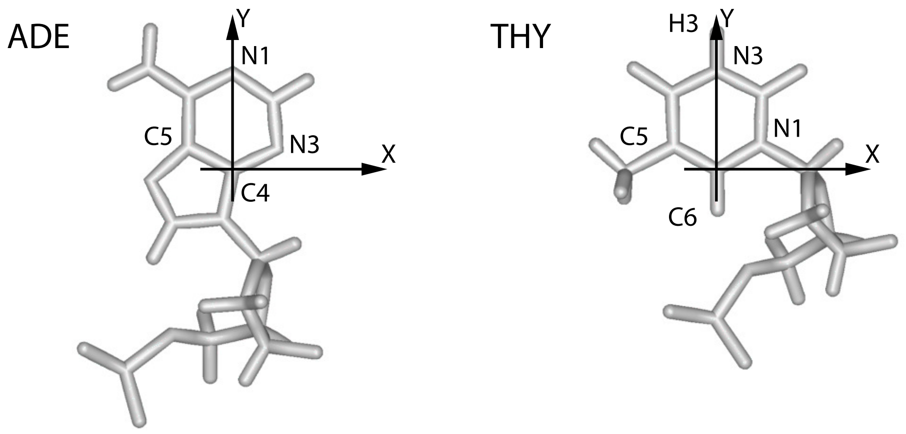


Figure 15.1: *ADE and THY after execution of Program 3.*

atoms as shown in Figure 15.1. 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 the cross product X×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×X; if the value is 2, then the specified Y axis becomes the formal Y axis and X is recreated from Y×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.

15.6 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.

15.7 Creating Molecules

The following functions are used to create molecules. Only an overview is given here; more details are in chapter 3.

```
molecule newmolecule();
int addstrand( molecule m, string str );
residue getresidue( string rname, string rlib );
residue transformres( matrix mat, residue res, string aex );
int addresidue( molecule m, string str, residue res );
int connectres( molecule m, string str,
       int rn1, string atm1, int rn2, string atm2 );
int mergestr( molecule m1, string str1, string 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  addresidue( m, \fIstr-name\fC, res );
10 ...
11 }
12 ...
```

In line 2, the function newmolecule() creates a molecule and stores it in m. The new molecule is empty—no strands, residues or atoms. Next addstrand() is used to add a strand named *str-1*. Strand names may be up to 255 characters in length and can include any characters except white space. Each strand in a molecule must have a unique name. There is no limit on the number of strands a molecule may have.

The actual structure would be created in the loop on lines 5-11. Each time around the loop, the function getresidue() is used to extract the next residue with the name *res-name* from some residue library *res-lib* and stores it in the residue variable res. Next the function transformres() applies a transformation matrix, held in the matrix variable mat to the residue in res, which places it in the orientation and position it will have in the new molecule. Finally, the function addresidue() appends the transformed residue to the end of the chain of residues in the strand *str-name* of the new molecule.

Residues in each strand are numbered from 1 to *N*, where *N* is the number of residues in that strand. The residue order is the order in which they were inserted with addresidue(). While nab does not require it, nucleic acid chains are usually numbered from 5′ to 3′ and proteins chains from the N-terminus to the C-terminus. The residues in nucleic acid strands and protein chains are usually bonded with the outgoing end of residue *i* bonded to the incoming end of residue *i*+1. However, as this is not always the case, nab requires the user to explicitly make all interresidue bonds with the builtin connectres().

connectres() makes bonds between two atoms in different residues of the same strand of a molecule. Only residues in the same strand can be bonded. connectres() takes six arguments. They are a molecule, the name of the strand containing the residues to be bonded, and two pairs each of a residue number and the name of an atom in that residue. As an example, this call to connectres(),

```
connectres( m, "sense", i, "O3'", i+1, "P" );
```

connects an atom named "O3'" in residue i to an atom named "P" in residue i+1, creating the phosphate bond that joins two nucleic acid monomers.

The function mergestr() is used to either move or copy the residues in one strand into another strand. Details are provided in chapter 3.

15.8 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://ambermd.org/).

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 it 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 $AMBERHOME/dat/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:

```
// getres() - map 1 letter names into 3 letter names
residue getres( string rname, string rlib )
{
    residue res;
    string map1to3[ hashed ];          // convert residue names

    map1to3["A"] = "ADE";     map1to3["C"] = "CYT";
    map1to3["G"] = "GUA";     map1to3["T"] = "THY";
    map1to3["U"] = "URA";

    map1to3["a"] = "ADE";     map1to3["c"] = "CYT";
    map1to3["g"] = "GUA";     map1to3["t"] = "THY";
    map1to3["u"] = "URA";

    if( r in map1to3 ) {
        res = getresidue( map1to3[ r ], rlib );
    }else{
        fprintf( stderr, "undefined residue %s\\n", r );
        exit( 1 );
    }
    return( res );
};
```

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.

15.9 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 with one colon consists of a strand part and a residue part; it selects *all* atoms in the selected residues in the selected strands. An empty part selects all strands, residues or atoms depending on which parts are empty.

nab patterns specify the *entire* string to be matched. For example, the atom pattern C matches only atoms named C , and not those named CA, HC, etc. To match any name that begins with C, use C*, to match any name ending with C, use *C and to match a C in any position use *C*. An atom expression is first parsed into its parts. The strand part is evaluated selecting one or more strands in a molecule. Next the residue part is evaluated. Only residues in selected strands can be selected. Finally the atom part is evaluated and only atoms in selected residues are selected. Here are some typical atom expressions and the atoms they match.

:ADE:	Select all atoms in any residue named ADE. All three parts are present but both the strand and atom parts are empty. The atom expression :ADE selects the same set of atoms.
::C,CA,N	select all atoms with names C, CA or N in all residues in all strands—typically the peptide backbone.
A:1-10,13,URA:C1'	Select atoms named C1' (the glycosyl-carbons) in residues 1 to 10 and 13 and in any residues named URA in the strand named A.
::C*[^']	Select all non-sugar carbons. The [^'] is an example of a negated character class. It matches any character in the last position except '.
::P,O?P,C[3-5]?,O[35]?	The nucleic acid backbone. This P selects phosphorous atoms. The O?P matches phosphate oxygens that have various second letters O1P, O2P or OAP or OBP. The C[3-5]? matches the backbone carbons, C3', C4', C5' or C3*, C4*, C5*. And the O[35]? matches the backbone oxygens O3', O5' or O3*, O5*.
:: or :	Select all atoms in the molecule.

An important property of nab atom expressions is that the order in which the strands, residues, and atoms are listed is unimportant. That is, the atom expression "2,1:5,2,3:N1,C1'" is the exact same atom expression as "1,2:3,2,5:C1',N1". All atom expressions are reordered, internal to nab, in increasing atom number. So, in the above example, the selected atoms will be selected in the following sequence:

```
1:2:N1, 1:2:C1', 1:3:N1, 1:3:C1', 1:5:N1, 1:5:C1', 2:2:N1, 2:2:C1',
2:3:N1, 2:3:C1', 2:5:N1, 2:5:C1'
```

The order in which atoms are selected internal to a specific residue are the order in which they appear in a nab PDB file. As seen in the above example, N1 appears before C1' in all nab nucleic acid residues and PDB files.

15.10 Looping over atoms in molecules

Another thing that many nab programs have to do is visit every atom of a molecule. nab provides a special form of its for-loop for accomplishing this task. These loops have this form:

```
    for( a in m ) stmt;
```

a and *m* represent an atom and a molecule variable. The action of the loop is to set *a* to each atom in *m* in this order. The first atom is the first atom of the first residue of the first strand. This is followed by the rest of the atoms of this residue, followed by the atoms of the second residue, etc until all the atoms in the first strand have been visited. The process is then repeated on the second and subsequent strands in *m* until *a* has been set to every atom in *m*. The order of the strands in a molecule is the order in which they were created with addstrand(), the order of the residues in a strand is the order in which they were added with addresidue() and the order of the atoms in a residue is the order in which they are listed in the residue library entry that the residue is based on.

The following program uses two nested for-in loops to compute all the proton-proton distances in a molecule. Distances less than cutoff are written to stdout. The program uses the second argument on the command to hold the cutoff value. The program also uses the =∼ operator to compare a character string , in this case an atom name to pattern, specified as a regular expression.

```
 1  // Program 4 - compute H-H distances <= cutoff
 2  molecule    m;
 3  atom        ai, aj;
 4  float       d, cutoff;
 5
 6  cutoff = atof( argv[ 2 ] );
 7  m = getpdb( "gcg10.pdb" );
 8
 9  for( ai in m ){
10      if( ai.atomname !~ "H" )continue;
11      for( aj in m ){
12          if( aj.tatomnum <= ai.tatomnum )continue;
13          if( aj.atomname !~ "H" )continue;
14          if(( d=distp(ai.pos,aj.pos))<=cutoff){
15              printf(
16                  "%3d %-4s %-4s %3d %-4s %-4s %8.3f\\n",
17                  ai.tresnum, ai.resname, ai.atomname,
18                  aj.tresnum, aj.resname, aj.atomname,
19                  d );
20          }
21      }
22  }
```

The molecule is read into m using getpdb(). Two atom variables ai and aj are used to hold the pairs of atoms. The outer loop in lines 9-22 sets ai to each atom in m in the order discussed above. Since this program is only interested in proton-proton distances, if ai is not a proton, all calculations involving that atom can be skipped. The if in line 10 tests to see if ai is a proton. It does so by testing to see if ai's name, available via the atomname attribute doesn't match the regular expression "H". If it doesn't match then the program executes the continue statement also on line 10, which has the effect of advancing the outer loop to its next atom.

>From the section on attributes, ai.atomname behaves like a character string. It can be compared against other character strings or tested to see if it matches a pattern or regular expression. The two operators, =∼ and !∼ stand for *match* and *doesn't-match* They also inform the nab compiler that the string on their right hand sides is to be treated like a regular expression. In this case, the regular expression "H" matches any name that contains the letter H, or any proton which is just what is required.

If ai is a proton, then the inner loop from 11-21 is executed. This sets aj to each atom in the same order as the loop in 9. Since distance is reflexive (*dist i, j* = *dist j, i*), and the distance between an atom and itself is 0, the inner loop uses the if on line 12 to skip the calculation on aj unless it follows ai in the molecule's atom order. Next the if on line 13 checks to see if aj is a proton, skipping to the next atom if it is not. Finally, the if on line 14 computes the distance between the two protons ai and aj and if it is <= cutoff writes the information out using the C-like I/O function printf().

15.11 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.

15.11.1 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.

15.11.2 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 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 that one atom, the average coordinate of all selected atoms is used as the point's value. (Note that a positive rotation angle here is defined to be clockwise, which is in accord with the IUPAC rules for defining torsional angles in molecules, but is opposite to the convention found in many other branches of mathematics.) Similarly, the functions trans4() and trans4p() create a transformation that effects a translation by a distance along the axis defined by two points. A positive translation is from tail to head.

transformres() applies a transformation to those atoms of res that match the atom expression aex. It returns a *copy* of the input residue with the changed coordinates. The input residue is unchanged. It returns NULL if the new residue could not be created. transformmol() applies a transformation to those atoms of mol that match aex . Unlike transformres(), transformmol() *changes* the coordinates of the input molecule. It returns the number of atoms selected by aex. In both functions, the special atom expression NULL selects all atoms in the input residue or molecule.

15.11.3 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 that one atom, the average of the selected atoms' coordinates is used. Z is created from X×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×X. If use is 2, then Y is chosen and X is recreated from Y×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 setframep( 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.

15.12 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 16.13. 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 **??** at this point.

15.12.1 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.

```
// bdna() - create average B-form duplex
molecule bdna( string seq )
{
        molecule m;
        string cseq;
        cseq = wc_complement( seq, "", "dna" );
        m = wc_helix( seq, "", "dna",
                 cseq, "", "dna",
                 2.25, -4.96, 36.0, 3.38, "s5a5s3a3" );
         return( m );
};
```

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 *s*ense and *a*nti 5′ terminal phosphates with hydrogens and adds hydrogens to both the *s*ense and *a*nti 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 16.13.

15.12.2 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.

```
// wc_complement() - create a string that is the W/C
// complement of the string seq
```

```
3  string wc_complement( string seq, string rlib, string rlt )
4  //  (note that rlib is unused: included only for backwards compatibility
5  {
6     string acbase, base, wcbase, wcseq;
7     int i, len;
8  
9     if( rlt == "dna" )      acbase = "t";
10    else if( rlt == "rna" ) acbase = "u";
11    else{
12       fprintf( stderr,
13         "wc_complement: rlt (%s) is not dna/rna, no W/C comp.", rlt );
14       return( NULL );
15    }
16    len = length( seq );
17    wcseq = NULL;
18    for( i = 1; i <= len; i = i + 1 ){
19        base = substr( seq, i, 1 );
20        if( base == "a" || base == "A" )       wcbase = acbase;
21        else if( base == "c" || base == "C" )  wcbase = "g";
22        else if( base == "g" || base == "G" )  wcbase = "c";
23        else if( base == "t" || base == "T" )  wcbase = "a";
24        else if( base == "u" || base == "U" )  wcbase = "a";
25        else{
26            fprintf( stderr, "wc_complement: unknown base %sn", base );
27            return( NULL );
28        }
29        wcseq = wcseq + wcbase;
30    }
31    return( wcseq );
32 }
```

wc_complement() begins its work in line 9, where the nucleic acid type, as indicated by rlt as DNA or RNA is used to determine the correct complement for an a. The complementary sequence is created in the for loop that begins in line 18 and extends to line 30. The nab builtin substr() is used to extract single characters from the input sequence beginning with with position 1 and working from left to right until entire input sequence has been converted. The if-tree from lines 20 to 28 is used to set the character complementary to the current character, using the previously determined acbase if the input character is an a or A. Any character other than the expected a, c, g, t, u (or A, C, G, T, U) is an error causing wc_complement() to print an error message and return NULL, indicating that it failed. Line 29 shows how nab uses the infix + to concatenate character strings. When the entire string has been complemented, the for loop terminates and the complementary sequence now in wcseq is returned as the function value. Note that if the input sequence is empty, wc_complement() returns NULL, indicating failure.

15.12.3 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

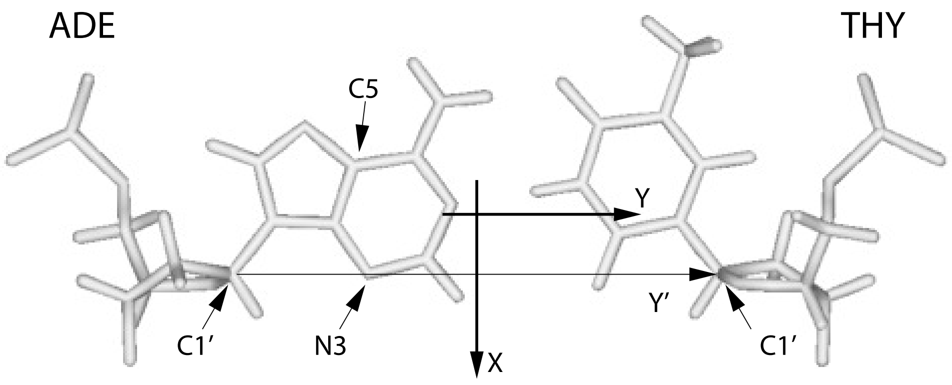


Figure 15.2: *ADE.THY from wc_basepair().*

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 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.[279]

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.

15.12.4 wc_basepair()

The function wc_basepair() takes two residues and assembles them into a two stranded nab molecule containing one base pair. Residue sres is placed in the "sense" strand and residue ares is placed in the "anti" strand. The work begins in line 14 where newmolecule() is used to create an empty molecule stored in m. Two strands, sense and anti are added using addstrand(). In addition, two more molecules are created, m_sense for the sense residue and m_anti for the anti residue. The if-trees in lines 26-61 and 63-83 are used to select residue dependent atoms that will be used to move the base pairs into a convenient orientation for helix generation. The *purine*:C4 and *pyrimidine*:C6 distance which is residue dependent is also set. In line 62, addresidue() adds sres to the strand sense of m_sense. In line 84, addresidue() adds ares to the strand anti of m_anti. Lines 86 and 87 align the molecules containing the sense residue and anti residue so that sres and ares are on top of each other. Line 88 creates a transformation matrix that rotates m_anti (containing ares) 180o about the X-axis. After applying this transformation, the two bases are still occupying the same space but ares is now antiparallel to sres. Line 90 creates a transformation matrix that displaces m_anti and ares along the Y-axis by sep. The properly positioned molecules containing sres and ares are merged into a single molecule, m, completing the base pair. Lines 97-98 move this base pair to a more convenient orientation for helix generation. Initially the base as shown in Figure 15.2 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'.

```
// wc_basepair() - create Watson/Crick base pair
#define AT_SEP 8.29
#define CG_SEP 8.27

molecule wc_basepair( residue sres, residue ares )
{
        molecule m, m_sense, m_anti;
        float sep;
        string srname, arname;
        string xtail, xhead;
        string ytail, yhead;
        matrix mat;

        m = newmolecule();
        m_sense = newmolecule();
        m_anti = newmolecule();
        addstrand( m, "sense" );
        addstrand( m, "anti" );
        addstrand( m_sense, "sense" );
        addstrand( m_anti, "anti" );

        srname = getresname( sres );
        arname = getresname( ares );
        ytail = "sense::C1'";
        yhead = "anti::C1'";
        if( ( srname == "ADE" ) || ( srname == "DA" ) ||
            ( srname == "RA" ) || ( srname =~ "[DR]A[35]" ) ){
                sep = AT_SEP;
                xtail = "sense::C5";
                xhead = "sense::N3";
                setframe( 2, m_sense,
                        "::C4", "::C5", "::N3", "::C4", "::N1" );
        }else if( ( srname == "CYT" ) || ( srname =~ "[DR]C[35]*" ) ){
                sep = CG_SEP;
                xtail = "sense::C6";
                xhead = "sense::N1";
                setframe( 2, m_sense,
                        "::C6", "::C5", "::N1", "::C6", "::N3" );
        }else if( ( srname == "GUA" ) || ( srname =~ "[DR]G[35]*" ) ){
                sep = CG_SEP;
                xtail = "sense::C5";
                xhead = "sense::N3";
                setframe( 2, m_sense,
                        "::C4", "::C5", "::N3", "::C4", "::N1" );
        }else if( ( srname == "THY" ) || ( srname =~ "DT[35]*" ) ){
                sep = AT_SEP;
                xtail = "sense::C6";
                xhead = "sense::N1";
                setframe( 2, m_sense,
                        "::C6", "::C5", "::N1", "::C6", "::N3" );
        }else if( ( srname == "URA" ) || ( srname =~ "RU[35]*" ) ){
                sep = AT_SEP;
                xtail = "sense::C6";
                xhead = "sense::N1";
                setframe( 2, m_sense,
                        "::C6", "::C5", "::N1", "::C6", "::N3" );
        }else{
                fprintf( stderr,
                        "wc_basepair : unknown sres %s\\n",srname );
                exit( 1 );
        }
        addresidue( m_sense, "sense", sres );
        if( ( arname == "ADE" ) || ( arname == "DA" ) ||
            ( arname == "RA" ) || ( arname =~ "[DR]A[35]" ) ){
                setframe( 2, m_anti,
                        "::C4", "::C5", "::N3", "::C4", "::N1" );
        }else if( ( arname == "CYT" ) || ( arname =~ "[DR]C[35]*" ) ){
                setframe( 2, m_anti,
                        "::C6", "::C5", "::N1", "::C6", "::N3" );
        }else if( ( arname == "GUA" ) || ( arname =~ "[DR]G[35]*" ) ){
                setframe( 2, m_anti,
                        "::C4", "::C5", "::N3", "::C4", "::N1" );
        }else if( ( arname == "THY" ) || ( arname =~ "DT[35]*" ) ){
                setframe( 2, m_anti,
                        "::C6", "::C5", "::N1", "::C6", "::N3" );
        }else if( ( arname == "URA" ) || ( arname =~ "RU[35]*" ) ){
                setframe( 2, m_anti,
                        "::C6", "::C5", "::N1", "::C6", "::N3" );
        }else{
                fprintf( stderr,
                        "wc_basepair : unknown ares %s\\n",arname );
                exit( 1 );
        }
        addresidue( m_anti, "anti", ares );

        alignframe( m_sense, NULL );
        alignframe( m_anti, NULL );
        mat = newtransform( 0., 0., 0., 180., 0., 0. );
        transformmol( mat, m_anti, NULL );
        mat = newtransform( 0., sep, 0., 0., 0., 0. );
        transformmol( mat, m_anti, NULL );
        mergestr( m, "sense", "last", m_sense, "sense", "first" );
        mergestr( m, "anti", "last", m_anti, "anti", "first" );

        freemolecule( m_sense ); freemolecule( m_anti );

        setframe( 2, m, "::C1'", xtail, xhead, ytail, yhead );
        alignframe( m, NULL );
        return( m );
};
```

15.12.5 wc_helix() Implementation

The function wc_helix() assembles base pairs from wc_basepair() into a helical duplex. It is a fairly complicated function that uses several transformations and shows how mergestr() is used to combine smaller molecules into a larger one. In addition to creating complete duplexes, wc_helix() can also create molecules that contain only one strand of a duplex. Using the special value NULL for either seq or aseq creates a duplex that omits the residues for the NULL sequence. The molecule still contains two strands, sense and anti, but the strand corresponding to the NULL sequence has zero residues. wc_helix() first determines which strands are required, then creates the first base pair, then creates the subsequent base pairs and assembles them into a helix and finally packages the requested strands into the returned molecule.

Lines 20-34 test the input sequences to see which strands are required. The variables has_s and has_a are flags where a value of 1 indicates that seq and/or aseq was requested. If an input sequence is NULL, wc_complement() is used to create it and the appropriate flag is set to 0. The nab builtin setreslibkind() is used to set the nucleic acid type so that the proper residue (DNA or RNA) is extracted from the residue library.

The first base pair is created in lines 42-63. The two letters corresponding the 5′ base of seq and the 3′ base of aseq are extracted using the nab builtin substr(), converted to residues using getresidue() and assembled into a base pair by wc_basepair(). This base pair is oriented as in Figure 2 with the origin at the intersection of the lines X and Y′. Two transformations are created, xomat for the x-offset and inmat for the inclination and applied to this pair.

Base pairs 2 to slen-1 are created in the for loop in lines 66-87. substr() is used to extract the appropriate letters from seq and aseq which are converted into another base pair by getresidue() and wc_basepair(). Four transformations are applied to these base pairs - two to set the x-offset and the inclination and two more to set the twist and the rise. Next m2, the molecule containing the newly created properly positioned base pair must be bonded to the previously created molecule in m1. Since nab only permits bonds between residues in the same strand, mergestr() must be used to combine the corresponding strands in the two molecules before connectres() can create the bonds.

Because the two strands in a Watson/Crick duplex are antiparallel, adding a base pair to one end requires that one residue be added *after* the *last* residue of one strand and that the other residue added *before* the *first* residue of the other strand. In wc_helix() the sense strand is extended after its last residue and the anti strand is extended before its first residue. The call to mergestr() in line 79 extends the sense strand of m1 with the the residue of the sense strand of m2. The residue of m2 is added after the "last" residue of of the sense strand of m1. The final argument "first" indicates that the residue of m2 are copied in their original order m1:sense:last is followed by m2:sense:first. After the strands have been merged, connectres() makes a bond between the O3' of the next to last residue (i-1) and the P of the last residue (i). The next call to mergestr() works similarly for the residues in the anti strands. The residue in the anti strand of m2 are copied into the the anti strand of m1 *before* the first residue of the anti strand of m1 m2:anti:*last* precedes m1:anti:*first* . After merging connectres() creates a bond between the O3' of the new first residue and the P of the second residue.

Lines 121-130 create the returned molecule m3. If the flag has_s is 1, mergestr() copies the entire sense strand of m1 into the empty sense strand of m3. If the flag has_a is 1, the anti strand is also copied.

```
1  // wc_helix() - create Watson/Crick duplex
2  string  wc_complement();
3  molecule wc_basepair();
4  molecule wc_helix(
5         string seq, string sreslib, string snatype,
6         string aseq, string areslib, string anatype,
7         float xoff, float incl, float twist, float rise,
8         string opts )
9  {
10 molecule m1, m2, m3;
11 matrix xomat, inmat, mat;
12 string arname, srname;
13 string sreslib_use, areslib_use;
14 string loup[ hashed ];
15 residue sres, ares;
16 int     has_s, has_a;
17 int i, slen;
18 float   ttwist, trise;
19 
20 has_s = 1; has_a = 1;
21 if( sreslib == "" ) sreslib_use = "all_nucleic94.lib";
```

```
        else sreslib_use = sreslib;
if( areslib == "" ) areslib_use = "all_nucleic94.lib";
        else areslib_use = areslib;

if( seq == NULL && aseq == NULL ){
        fprintf( stderr, "wc_helix: no sequence\\n" );
        return( NULL );
}else if( seq == NULL ){
        seq = wc_complement( aseq, areslib_use, snatype );
        has_s = 0;
}else if( aseq == NULL ){
        aseq = wc_complement( seq, sreslib_use, anatype );
        has_a = 0;
}

slen = length( seq );
loup["g"] = "G"; loup["a"] = "A";
loup["t"] = "T"; loup["c"] = "C";

//                      handle the first base pair:
setreslibkind( sreslib_use, snatype );
srname = "D" + loup[ substr( seq, 1, 1 ) ];
if( opts =~ "s5" )
    sres = getresidue( srname + "5", sreslib_use );
else if( opts =~ "s3" && slen == 1 )
    sres = getresidue( srname + "3", sreslib_use );
else sres = getresidue( srname, sreslib_use );

setreslibkind( areslib_use, anatype );
arname = "D" + loup[ substr( aseq, 1, 1 ) ];
if( opts =~ "a3" )
    ares = getresidue( arname + "3", areslib_use );
else if( opts =~ "a5" && slen == 1 )
    ares = getresidue( arname + "5", areslib_use );
else ares = getresidue( arname, areslib_use );
m1 = wc_basepair( sres, ares );
freeresidue( sres ); freeresidue( ares );
xomat = newtransform(xoff, 0., 0., 0., 0., 0. );
transformmol( xomat, m1, NULL );
inmat = newtransform( 0., 0., 0., incl, 0., 0.);
transformmol( inmat, m1, NULL );

//                      add in the main portion of the helix:
trise = rise; ttwist = twist;
for( i = 2; i <= slen-1; i = i + 1 ){
        srname = "D" + loup[ substr( seq, i, 1 ) ];
        setreslibkind( sreslib, snatype );
        sres = getresidue( srname, sreslib_use );
        arname = "D" + loup[ substr( aseq, i, 1 ) ];
        setreslibkind( areslib, anatype );
        ares = getresidue( arname, areslib_use );
        m2 = wc_basepair( sres, ares );
        freeresidue( sres ); freeresidue( ares );
        transformmol( xomat, m2, NULL );
        transformmol( inmat, m2, NULL );
        mat = newtransform( 0., 0., trise, 0., 0., ttwist );
        transformmol( mat, m2, NULL );
        mergestr( m1, "sense", "last", m2, "sense", "first" );
        connectres( m1, "sense", i-1, "O3'", i, "P" );
        mergestr( m1, "anti", "first", m2, "anti", "last" );
        connectres( m1, "anti", 1, "O3'", 2, "P" );
        trise = trise + rise;
        ttwist = ttwist + twist;
        freemolecule( m2 );
}


i = slen;          // add in final residue pair:

if( i > 1 ){
        srname = substr( seq, i, 1 );
        srname = "D" + loup[ substr( seq, i, 1 ) ];
        setreslibkind( sreslib, snatype );
        if( opts =~ "s3"  )
           sres = getres( srname + "3", sreslib_use );
        else
           sres = getres( srname, sreslib_use );
        arname = "D" + loup[ substr( aseq, i, 1 ) ];
        setreslibkind( areslib, anatype );
        if( opts =~ "a5" )
           ares = getres( arname + "5", areslib_use );
        else
           ares = getres( arname, areslib_use );

        m2 = wc_basepair( sres, ares );
        freeresidue( sres ); freeresidue( ares );
        transformmol( xomat, m2, NULL );
        transformmol( inmat, m2, NULL );
        mat = newtransform( 0., 0., trise, 0., 0., ttwist );
        transformmol( mat, m2, NULL );
        mergestr( m1, "sense", "last", m2, "sense", "first" );
        connectres( m1, "sense", i-1, "O3'", i, "P" );
        mergestr( m1, "anti", "first", m2, "anti", "last" );
        connectres( m1, "anti", 1, "O3'", 2, "P" );
        trise = trise + rise;
        ttwist = ttwist + twist;
        freemolecule( m2 );
}

m3 = newmolecule();
addstrand( m3, "sense" );
addstrand( m3, "anti" );
if( has_s )
        mergestr( m3, "sense", "last", m1, "sense", "first" );
if( has_a )
        mergestr( m3, "anti", "last",  m1, "anti", "first" );
freemolecule( m1 );

return( m3 );
};
```

16 NAB: Language Reference

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.

16.1 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.

16.1.1 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

16.1.2 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

16.1.3 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:

\a	Bell (a for audible alarm)
\b	Back space
\f	Form feed (new page)
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\″	Literal double quote
\\	Literal backspace
\ooo	Octal character
\xhh	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.

16.1.4 Operators

nab uses several additional 1 or 2 character symbols as operators. Operators combine literals and identifiers into expressions.

Operator	Meaning	Precedence	Associates
( )	expression grouping	9	
[ ]	array indexing	9	
.	select attribute	8	
unary −	negation	8	right to left
!	not	8	
^	cross product	6	left to right
@	dot product	6	
*	multiplication	6	left to right
/	division	6	left to right
%	modulus	6	left to right
+	addition, concatenation	5	left to right
binary −	subtraction	5	left to right
<	less than	4	
<=	less than or equal to	4	
==	equal	4	
!=	not equal	4	
>=	greater than or equal to	4	
>	greater than	4	
=~	match	4	
!~	doesn't match	4	
in	hashed array member *or* atom in molecule	4	
&&	and	3	
||	or	2	
=	assignment	1	right to left

16.1.5 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.

16.2 Higher-level constructs

16.2.1 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.

16.2.2 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 *each* strand in the molecule.
tatomnum	int	no	The *total* 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 *strandnum:resnum:atomname*.
resid	string	yes	The *resid* of the residue containing this atom; see the **Residue attributes** table.
resname	string	yes	The name of the residue containing this atom.
resnum	int	no	The number of the residue containing the atom. resnum starts at 1 for *each* strand.
tresnum	int	no	The *total* 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 *putpdb*.
resname	string	yes	Three-character identifier
resnum	int	no	The number of the residue. resnum starts at 1 for *each* strand.
tresnum	int	no	The *total* number of the residue, starting at 1. Unlike resnum, tresnum 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.

16.2.3 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 *Nd* , where *Nd* 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.

16.2.4 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* nonzero 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.

16.2.5 Regular expressions

The =~ and *!~* 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.

16.2.6 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 of 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.

16.2.7 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.

%...c	convert a character
%...d	convert and integer
%...lf	convert a float
%...s	convert a string
%%	convert a literal %

Input and output format descriptors and format expressions resemble each other and in many cases the same format expression can be used for both input and output. However, the two types of format descriptors have different options and their actions are sufficiently distinct to consider in some detail. Generally, C based formatted output is more useful than C based formatted input.

When an input format expression is executed, it is scanned at most once from left to right. If the current format expression character is an ordinary character (anything but space or %), it must match the current character in the input stream. If they match then both the current character of the format expression and current character of the stream are advanced one character to the right. If they don't match, the scan ends. If the current format expression character is a space or a run of spaces and if the current input stream is one or more "white space" characters (space, tab, *newline*), then both the format and input stream are advanced to the next non-white space character. If the input format is one or more spaces but the current character of the input stream is non-blank, then only the format expression is advanced to the next non-blank character. If the current format character is a percent sign, the format descriptor is used to convert the next "field" in the input stream. A field is a sequence of non-blank characters surrounded by white space or the beginning or end of the stream. This means that a format descriptor will *skip* white space including newlines to find non blank characters to convert, even if it is the first element of the format expression. This implicit scanning is what limits the ability of C based formatted input to read fixed format data that contains any spaces.

Note that `lf` is used to input a NAB *float* variable, rather than the `f` argument that would be used in C. This is because *float* in NAB is converted to *double* in the output C code (see *defreal.h* if you want to change this behavior.) Ideally, the NAB compiler should parse the format string, and make the appropriate substitutions, but this is not (yet) done: NAB translates the format string directly into the C code, so that the NAB code must also generally use `lf` as a format descriptor for floating point values.

`nab` input format descriptors have two options, a field width, and an assignment suppression indicator. The field width is an integer which specifies how much of current *field* and not the input stream is to be converted. Conversion begins with the first character of the field and stops when the correct number of characters have been converted or white space is encountered. A star (*) option indicates that the field is to be converted, but the result of the conversion is not stored. This can be used to skip unwanted items in a data stream. The order of the two options does not matter.

The execution of an output format expression is somewhat different. It is scanned once from left to right. If the current character is not a percent sign, it placed on the output stream. Thus spaces have no special significance in formatted output. When the scan encounters a percent sign it replaces the entire format descriptor with the properly formatted value of the corresponding output expression.

Each output format descriptor has four optional attributes—width, alignment, padding and precision. The width is the *minimum* number of characters the data is to occupy for output. Padding controls how the field will be filled if the number of characters required for the data is less than the field width. Alignment specifies whether the data is to start in the first character of the field (left aligned) or end in the last (right aligned). Finally precision, which applies only to string and float conversions controls how much of the string is be converted or how many digits should follow the decimal point.

Output field attributes are specified by optional characters between the initial percent sign and the final data type character. Alignment is first, with left alignment specified by a minus sign (-). Any other character after the percent sign indicates right alignment. Padding is specified next. Padding depends on both the alignment and the type of the data being converted. Character conversions (`%c`) are always filled with spaces, regardless of their alignment. Left aligned conversions are also always filled with spaces. However, right aligned string and numeric conversions can use a `0` to indicate that left fill should be zeroes instead of spaces. In addition numeric conversions can also specify an optional + to indicate that non-negative numbers should be preceded by a plus sign. The default action for numeric conversions is that negative numbers are preceded by a minus, and other numbers have no sign. If both `0` and + are specified, their order does not matter.

Output field width and precision are last and are specified by one or two integers or stars (*) separated by a period (.). The first number (or star) is the field width, the second is its precision. If the precision is not specified, a default precision is chosen based on the conversion type. For floats (`%f`), it is six decimal places and for strings it is the entire string. Precision is not applicable to character or integer conversions and is ignored if specified. Precision may be specified without the field width by use of single integer (or star) preceded by a period. Again, the action is conversion type dependent. For strings (`%s`), the action is to print the first N characters of the string or the entire string, whichever is shorter. For floats (`%f`), it will print N decimal places but will extend the field to whatever size if required to print the whole number part of the float. The use of the star (*) as an output width or precision indicates that the width or precision is specified as the next argument in the conversion list which allows for runtime widths and precisions.

Ouput format options	
Alignment	
-	left justified
default	right justified
Padding	
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.
Width & precision	
W	*minimum* field width of *W*. *W* is either an integer or a * where the star indicates that the width is the next argument in the parameter list.
W.P	*minimum* field width of *W*, with a precision of *P*. *W*,*P* 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 *P* characters of the string or the entire string whichever is shorter. %f, print *P* 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 *P* 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.

16.3 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.

16.3.1 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.
```

16.3.2 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.

16.3.3 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 nonzero 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;
```

16.3.4 While Statement

The while statement is used to execute the statement under its control as long as the the while expression is true (nonzero). 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;
}
```

16.3.5 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 (nonzero). 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.

16.3.6 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
      ...
}
```

16.3.7 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
    ...
}
```

16.3.8 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.

16.3.9 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.

16.4 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 { ... };
```

16.5 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. It 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.

16.5.1 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
};
```

16.5.2 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;
```

16.6 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
Unary -	-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
Binary -	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.

16.7 String Functions

nab provides the following awk-like string functions. Unlike awk, the nab functions do not have optional parameters or builtin variables that control the actions or receive results from these functions. nab strings are indexed from 1 to N where N is the number of characters in the string.

```
int length( string s );
int index( string s, string t );
int match( string s, string r, int rlength );
string substr( string s, int pos, int len );
```

int split(string s, string fields[], string fsep);
int sub(string r, string s, string t);
int gsub(string r, string s, string t);

length() returns the length of the string s. Both "" and NULL have length 0. index() returns the position of the left most occurrence of t in s. If t is not in s, index() returns 0. match returns the position of the longest leftmost substring of s that matches the regular expression r. The length of this substring is returned in rlength. If no substring of s matches r, match() returns 0 and rlength is set to 0. substr() extracts the substring of length len from s beginning at position pos. If len is greater than the rest of the string beginning at pos, return the substring from pos to N where N is the length of the string. If pos is < 1 or $>$ N, return "".

split() partitions s into fields separated by fsep. These field strings are returned in the array fields. The number of fields is returned as the function value. The array fields must be allocated before split() is called and must be large enough to hold all the field strings. The action of split() depends on the value of fsep. If fsep is a string containing one or more blanks, the fields of s are considered to be separated by *runs* of white space. Also, leading and trailing white space in s do not indicate an empty initial or final field. However, if fsep contains any value but blank, then fields are considered to be delimited by *single* characters from fsep and initial and/or trailing fsep characters do represent initial and/or trailing fields with values of "". NULL and the empty string "" have 0 fields. If both s and fsep are composed of only white space then s also has 0 fields. If fsep is not white space and s consists of nothing but characters from fsep, s will have $N + 1$ fields of "" where N is the number of characters of s.

sub() replaces the leftmost, longest substring of the *target string* t that matches the *regular expression* r with the *substitution string* s. gsub() replaces all non-overlapping substrings of t that match the regular expression r with the string s. Each function returns the number of substitutions made. Unlike awk, the regular expression r is a string variable, with no surrounding '/' characters. For example:

```
int nmatch;
string regexp, substitute, target;
target = "water, water, everywhere";
regexp = "at";
substitute = "ith";
nmatch = gsub( regexp, substitute, target);
```

After this, `target` will contain "`wither, wither, everywhere`", and `nmatch` will be 2.

The special substitute character '&' stands for the precise substring matched by the regular expression. Hence

```
target = "daabaaa";
sub( "a+", "c&c", target);
```

will yield "`dcaacbaaa`". Note what "leftmost, longest substring" means here: "lefmost" takes precedence over "longest".

16.8 Math Functions

nab provides the builtin mathematical functions shown in Table 16.1. Since nab is intended for chemical structure calculations which always measure angles in degrees, the argument to the trig functions—cos(), sin() and tan()— and the return value of the inverse trig functions—acos(), asin(), atan() and atan2()—are in degrees instead of radians as they are in other languages. Note that the pseudo-random number functions have a different calling sequence than in earlier versions of NAB; you may have to edit and re-compile earlier programs that used those routines.

16.9 System Functions

int exit(int i);

Inverse Trig Functions.	
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].
Trig Functions	
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.
Conversion Functions.	
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.
Other Functions.	
float rand2();	Return a pseudo-random number in (0,1).
float gauss(float mean, float sd);	Return a pseudo-random number taken from a Gaussian distribution with the given mean and standard deviation. The rand2() and gauss() routines share a common seed.
int setseed(int seed);	Reset the pseudo-random number sequence with the new seed, which must be a negative integer.
int rseed( );	Use the system time() command to set the random number sequence with a reasonably random seed. Returns the seed it used; this could be used in a later call to setseed() to regenerate the same sequence of pseudo-random values.
float ceil(float x);	Return $\lceil x \rceil$.
float exp(float x);	Return e^x.
float cosh(float x);	Return the hyperbolic cosine of x.
float fabs(float x);	Return $\|x\|$.
float floor(float x);	Return $\lfloor x \rfloor$.
float fmod(float x, float y);	Return r, the remainder of x with respect to y; the signs of r and y are the same.
float log(float x);	Return the natural logarithm of x.
float log10(float x);	Return the base 10 logarithm of x.
float pow(float x, float y);	Return x^y, $x > 0$.
float sinh(float x);	Return the hyperbolic sine of x.
float tanh(float x);	Return the hyperbolic tangent of x.
float sqrt(float x);	Return positive square root of x, x >= 0.

Table 16.1: *NAB built-in mathematical functions*

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.

16.10 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.

16.10.1 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.

fopen() mode values	
"r"	Open for reading. The file fname 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 covert 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 $AMBERHOME/AmberTools/src/nab/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.

16.10.2 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 on non-sensical results.

```
int putmatrix(string filename, matrix mat);
```

Write matrix mat to 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 writen as 4 lines of 4 numbers. Return 0 on success and 1 on failure.

16.11 Molecule Creation Functions

The nab molecule type has a complex and dynamic internal structure organized in a three level hierarchy. A molecule contains zero or more named strands. Strand names are strings of any characters except white space and can not exceed 255 characters in length. Each strand in a molecule must have a unique name. Strands in different molecules may have the same name. A strand contains zero or more residues. Residues in each strand are numbered from 1. There is no upper limit on the number of residues a strand may contain. Residues have names, which need not be unique. However, the combination of *strand-name:res-num* is unique for every residue in a molecule. Finally residues contain one or more atoms. Each atom name in a residue should be distinct, although this is neither required nor checked by nab. nab uses the following functions to create and modify molecules.

```
molecule newmolecule();
molecule copymolecule( molecule mol );
int freemolecule( molecule mol );
int freeresidue( residue r );
int addstrand( molecule mol, string sname );
int addresidue( molecule mol, string sname, residue res );
int connectres( molecule mol, string sname, int res1, string aname1,
        int res2, string aname2 );
int mergestr( molecule mol1, string str1, string end1, molecule mol2, string str2, string end2 );
```

newmolecule() creates an “empty” molecule—one with no strands, residues or atoms. It returns NULL if it can not create it. copymolecule() makes a copy of an existing molecule and returns a NULL on failure. freemolecule() and *freeresidue()* are used to deallocate memory set aside for a molecule or residue. In most programs, these functions are usually not necessary, but should be used when a large number of molecules are being copied. Once a molecule has been created, addstrand() is used to add one or more named strands. Strands can be added at any to a molecule. There is no limit on the number of strands in a molecule. Strands can be added to molecules created by getpdb() or other functions as long as the strand names are unique. addstrand() returns 0 on success and 1 on failure. Finally addresidue() is used to add residues to a strand. The first residue is numbered 1 and subsequent residues are numbered 2, 3, etc. addresidue() also returns 0 on success and 1 on failure.

nab requires that users explicitly make all inter-residue bonds. connectres() makes a bond between two atoms of *different* residues of the strand with name sname. It returns 0 on success and 1 on failure. Atoms in different strands can not be bonded. The bonding between atoms in a residue is set by the residue library entry and can not be changed at runtime at the nab level.

The last function mergestr() is used to merge two strands of the same molecule or copy a strand of the second molecule into a strand of the first. The residues of a strand are ordered from 1 to N, where N is the number of residues in that strand. nab imposes no chemical ordering on the residues in a strand. However, since the strands are generally ordered, there are four ways to combine the two strands. mergestr() uses the two values "first" and "last" to stand for residues 1 and N. The four combinations and their meanings are shown in the next table. In the table, str1 has N residues and str2 has M residues.

end1	end2	Action
first	first	The residues of str2 are reversed and then inserted before those of str1: M , ..., 2, 1 : 1 , 2 , ..., N
first	last	The residues of str2 are inserted before those of str1: 1 , 2, ..., M : 1 , 2 , ..., N
last	first	The residues of str2 are inserted after those of str1: 1 , 2 , ..., N : 1 , 2 , ..., M
last	last	The residues of str2 are reversed and then inserted after those of str1: 1 , 2 , ..., N : M , ..., 2 , 1

16.12 Creating Biopoloymers

molecule linkprot(string strandname, string seq, string reslib);
molecule link_na(string strandname, string seq, string reslib, string natype, string opts);

Although many nab functions don't care what kind of molecule they operate on, many operations require molecules that are compatible with the Amber force field libraries (see Chapter 6). The best and most general way to do this is to use tleap commands, described in Chapter 8). The *linkprot()* and *link_na()* routines given here are limited commands that may sometimes be useful, and are included for backwards compatibility with earlier versions of NAB.

linkprot() takes a strand identifier and a sequence, and returns a molecule with this sequence. The molecule has an extended structure, so that the ϕ, $\psi$and ω angles are all 180o . The *reslib* input determines which residue library is used; if it is an empty string, the AMBER 94 all-atom library is used, with charged end groups at the N and C termini. All nab residue libraries are denoted by the suffix .rlb and LEaP residue libraries are denoted by the suffix .lib. If *reslib* is set to "nneut", "cneut" or "neut", then neutral groups will be used at the N-terminus, the C-terminus, or both, respectively.

The seq string should give the amino acids using the one-letter code with upper-case letters. Some non-standard names are: "H" for histidine with the proton on the δ position; "h" for histidine with the proton at the ϵ position; "3" for protonated histidine; "n" for an acetyl blocking group; "c" for an HNMe blocking group, "a" for an NH 2 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 **??**.

16.13 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()	
arna	Right Handed A-RNA (Arnott)
aprna	Right Handed A'-RNA (Arnott)
lbdna	Right Handed B-DNA (Langridge)
abdna	Right Handed B-DNA (Arnott)
sbdna	Left Handed B-DNA (Sasisekharan)
adna	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.; Chandraseharan, 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.

16.14 Reduced Representation DNA Modeling Functions

nab provides several functions for creating the reduced representation models of DNA described by R. Tan and S. Harvey.[280] 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.

16.15 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. Options for putpdb are given in Table 16.2.

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.

rnum1 rname1 aname1 rnum2 rname2 aname2 distance

16.16 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);

keyword	*meaning*
-pqr	Put (or get) 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 *-pqr* 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 *-brook* flag is not present, no conversion of atom and residue names is made, and no id is in column 78.
-wwpdb	Same as the -brook option, except use the “wwPDB” (aka version 3) residue and atom naming scheme.
-nocid	For *getpdb*, ignore the input chain id’s (column 22 of PDB-format files), and generate strand names as consecutive integers. For *putpdb*, do not put the chain-id in the output (i.e., if this flag is present, the chain-id column will be blank). This can be useful when many water molecules are present.
-allcid	If set, create a chain ID for every strand in the molecule being written. Use the strand’s name if it is an upper case letter, else use the next free upper case letter. Use a blank if no more upper case letters are available. Default is false.
-tr	Do not start numbering residues over again when a new chain is encountered, i.e., the residue numbers are consecutive across chains, as required by some force-field programs like Amber.

Table 16.2: *Options for getpdb and putpdb.*

int sugarpuckeranal(molecule mol, int strandnum, int startres, int endres);
int helixanal(molecule mol);
int plane(molecule mol, string aex, float A, float B, float C);
float molsurf(molecule mol, string aex, float probe_rad);

superimpose() transforms molecule mol so that the root mean square deviation between corresponding atoms in mol and r_mol is minimized. The corresponding atoms are those selected by the atom expressions aex1 applied to mol and aex2 applied to r_mol. The atom expressions must select the same number of atoms in each molecule. No checking is done to insure that the atoms selected by the two atom expressions actually correspond. superimpose() returns the transformation matrix it found. rmsd() computes the root mean square deviation between the pairs of corresponding atoms selected by applying aex1 to mol and aex2 to r_mol and returns the value in r. The two atom expressions must select the same number of atoms. Again, it is the user's responsibility to insure the two atom expressions select corresponding atoms. rmsd() returns 0 on success and 1 on failure.

angle() and anglep() compute the angle in degrees between three points. angle() uses atoms expressions to determine the average coordinates of the sets. anglep() takes as an argument three explicit points. Similarly, torsion() and torsionp() compute a torsion angle in degrees defined by four points. torsion() uses atom expressions to specify the points. These atom expression match sets of atoms in mol. The points are defined by the average coordinates of the sets. torsionp() uses four explicit points. Both functions return 0 if the torsion angle is not defined.

dist() and distp() compute the distance in angstroms between two explicit atoms. dist() uses atom expressions to determine which atoms to include in the calculation. An atom expression which selects more than one atom results in the distance being calculated from the average coordinate of the selected atoms. distp() returns the distance between two explicit points. The function countmolatoms() returns the number of atoms selected by aex in mol.

sugarpuckeranal() is a function that reports the various torsion angles in a nucleic acid structure. helixanal() is an interactive helix analysis function based on the methods described by Babcock *et al.*.[281]

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.

16.17 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 nonzero 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.

16.18 Time and date routines

NAB incorporates a few interfaces to time and date routines:

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

The date() routine returns a string in the format "03/08/1999", and the timeofday() routine returns the current time as "13:45:00". If you need access to more sophisticated time and date functions, the ftime() routine is just a wrapper for the standard C routine strftime, where the format string is used to determine what is output; see standard C documentation for how this works.

16.19 Computational resource consumption functions

NAB has a small number of functions to provide information about computational resources used during the run:

```
int mme_timer();
int mme_rism_max_memory();
```

mme_timer() provides tables of execution times for mme functions executed. It does not provide a complete summary nor does it include functions not in the mme family. It is, however, useful for identifying the most expensive routines. mme_rism_max_memory() reports the maximum amount of memory allocated during a 3D-RISM calculation.

17 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.

17.1 Transformation Matrix Functions

nab uses 4×4 matrices to hold coordinate transformations. nab provides these functions to create transformation matrices.

matrix newtransform(float dx, float dy, float dz, float rx, float ry, float rz);
matrix rot4(molecule mol, string aex1, string aex2, float ang);
matrix rot4p(point p1, point p2, float angle);

newtransform() creates a 4×4 matrix that will rotate an object by rz degrees about the Z axis, ry degrees about the Y axis, rx degrees about the X axis and then translate the rotated object by dx, dy, dz along the X, Y and Z axes. All rotations and transformations are with respect 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 that 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.

17.2 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×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×X. If use == 2, then the specified Y-axis is the real Y-axis and X is recreated from Y×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).

17.3 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.

17.4 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.

17.4.1 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]$\rightarrow$pts[2]. The other two points (if required) define two additional directions: pts[1]$\rightarrow$pts[3] and pts[1]$\rightarrow$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]$\rightarrow$pts[2]. Axis-3 is along the vector pts[1]$\rightarrow$pts[2] $\times$ pts[1]$\rightarrow$pts[3] and axis-2 is recreated along the vector axis-3 $\times$ 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]$\rightarrow$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]$\rightarrow$pts[2]. The rotations are in multiples of the angle ang beginning with o, 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]$\rightarrow$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 nonzero 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]$\rightarrow$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]$\rightarrow$pts[2], pts[1]$\rightarrow$pts[3] and pts[1]$\rightarrow$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.

17.4.2 Matrix I/O Functions

int MAT_fprint(file f, int nmats, matrix mats[1])
int MAT_sprint(string str, int nmats, matrix mats[1])
int MAT_fscan(file f, int smats, matrix mats[1])
int MAT_sscan(string str, int smats, matrix mats[1])
string MAT_getsyminfo()

This group of functions is used to read and write nab matrix variables. The two functions MAT_fprint() and MAT_sprint() write the the matrix to the file f or the string str. The number of matrices is specified by the parameter nmats and the matrices are passed in the array mats[].

The two functions MAT_fscan() and MAT_sscan() read matrices from the file f or the string str into the array mats[]. The parameter smats is the size of the matrix array and if the source file or string contains more than smats only the first smats will be returned. These two functions return the number of matrices read unless there the number of matrices is greater than smat or the last matrix was incomplete in which case they return -1.

In order to understand the last function in this group, MAT_getsyminfo(), it is necessary to discuss both the internal structure the nab matrix type and one of its most important uses. The nab matrix type is used to hold transformation matrices. Although these are atomic objects at the nab level, they are actually 4×4 matrices where the first three elements of the fourth row are the X Y and Z components of the translation part of the transformation. The matrix print functions write each matrix as four lines of four numbers separated by a single space. Similarly the matrix read functions expect each matrix to be represented as four lines of four white space (any number of tabs and spaces) separated numbers. The print functions use %13.6e for each number in order to produce output with aligned columns, but the scan functions only require that each matrix be contained in four lines of four numbers each.

Most nab programs use matrix variables as intermediates in creating structures. The structures are then saved and the matrices disappear when the program exits. Recently nab was used to create a set of routines called a "symmetry server". This is a set of nab programs that work together to create matrix streams that are used to assemble composite objects. In order to make it most general, the symmetry server produces only matrices leaving it to the user to apply them. Since these programs will be used to create hierarchies of symmetries or transformations we decided that the external representation (files or strings) of matrices would consist of two kinds of information — required lines of row values and optional lines beginning with the character # some of which are used to contain information that describes how these matrices were created.

MAT_getsyminfo() is used to extract this symmetry information from either a matrix file or a string that holds the contents of a matrix file. Each time the user calls MAT_fscan() or MAT_sscan(), any symmetry information present in the source file or string is saved in private buffer. The previous contents of this buffer are overwritten and lost. MAT_getsyminfo() returns the contents of this buffer. If the buffer is empty, indicating no symmetry information was present in either the source file or string, MAT_getsyminfo() returns NULL.

17.5 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.

17.5.1 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:	$IM_1 \times R_1, IM_2 \times R_1, IM_3 \times R_1, IM_1 \times R_2, IM_2 \times R_2, IM_3 \times R_2$

17.5.2 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 the transformation hierarchy and their meaning and use is covered in the section on "Searching Transformation Spaces". A complete list of keywords, their acceptable values and defaults is shown below.

Keyword	Default Value	Possible Values
symmetry	None	cube, cyclic, dihedral, dodeca, helix, ico, octa, tetra.
transform	None	orient, rotate, translate.
name	mPid	Any string of nonblank characters.
noid	false	true, false.
axestype	relative	absolute, relative.
center	None	Any three numbers separated by tabs or spaces.
axis, axis1	None	
axis2	None	
axis3	None	
angle,angle1	0	Any number.
angle2	0	
angle3	0	
dist	0	
count	1	Any integer.

axis and axis1 are synonyms as are angle and angle1.

The symmetry and transform keywords specify the operation. One or the other but not both must be specified.

The name keyword names a particular symmetry operation. The default name is m immediately followed by the process ID, eg m2286. name is used by the transformation space search routines tss_init and tss_next and is described later in the section "Searching Transformation Spaces".

The noid keyword with value true suppresses generation of the identity matrix in symmetry operations. For example, the keywords below

```
symmetry cyclic
noid false
center 0 0 0
axis 0 0 1
count 3
```

produce three matrices which perform rotations of 0o, 120o and 240o about the Z-axis. If noid is true, only the two non-identity matrices are created. This option is useful in building objects with two or three orthogonal 2-fold axes and is discussed further in the example "Icosahedron from Rotations". The default value of noid is false.

The axestype, center and axis* keywords defined the symmetry axes. The center and axis* keywords each require a point value which is three numbers separated by tabs or spaces. Numbers may integer or real and in fixed or exponential format. Internally all numbers are converted to nab type float which is actually double precision. No space is permitted between the minus sign of a negative number and the digits.

The interpretation of these points depends on the value of the keyword axestype. If it is absolute then the axes are defined as the vectors center→axis1, center→axis2 and center→axis3. If it relative, then the axes are vectors whose directions are **O→axis1**, **O→axis2** and **O→axis3** with their origins at center. If the value of center is 0,0,0, then absolute and relative are equivalent. The default value axestype is relative; center and the axis* do not have defaults.

The angle keywords specify the rotation about the axes. angle1 is associated with axis1 etc. Note that angle and angle1 are synonyms. The angle is in degrees, with positive being in the counterclockwise direction as you sight from the axis point to the center point. Either an integer or real value is acceptable. No space is permitted between the minus sign of a negative number and its digits. All angle* keywords have a default value of 0.

The dist keyword specifies the translation along an axis. The positive direction is from center to axis. Either integer or real value is acceptable. No space is permitted between the minus sign of a negative number and its digits. The default value of dist is 0.

The count keyword is used in three related ways. For the cyclic value of the symmetry it specifies ount matrices, each representing a rotation of 360/count. It also specifies the same rotations about the non 2-fold axis of dihedral symmetry. For helix symmetry, it indicates that count matrices should be created, each with a rotation of angle. In all cases the default value is 1.

This table shows which keywords are used and/or required for each type of operation.

symmetry	**name**	**noid**	**axestype**	**center**	*axes*	*angles*	**dist**	**count**
cube	m*Pid*	false	relative	Required	1,2	-	-	-
cyclic	m*Pid*	false	relative	Required	1	-	-	D=1
dihedral	m*Pid*	false	relative	Required	1,2	-	-	D=1
dodeca	m*Pid*	false	relative	Required	1,2	-	-	-
helix	m*Pid*	false	relative	Required	1	1,D=0	D=0	D=1
ico	m*Pid*	false	relative	Required	1,2	-	-	-
octa	m*Pid*	false	relative	Required	1,2	-	-	-
tetra	m*Pid*	false	relative	Required	1,2	-	-	-
transform	**name**	**noid**	**axestype**	**center**	*axes*	*angles*	**dist**	**count**
orient	m*Pid*	-	relative	Required	All	All,D=0	-	-
rotate	m*Pid*	-	relative	Required	1	1,D=0	-	-
translate	m*Pid*	-	relative	Required	1	-	D=0	-

17.5.3 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.

17.5.4 matmul

The matmul program takes two files of matrices, and creates a new stream of matrices formed by the pair wise product of the matrices in the input streams. The new matrices are written to stdout. If the number of matrices in the two input files differ, the last matrix of the shorter file is replicated and applied to all remaining matrices of the longer file. For example, if the file 3.mat has three matrices and the file 5.mat has five, then the command "matmul 3.mat 5.mat" would result in the third matrix of 3.mat multiplying the third, forth and fifth matrices of 5.mat.

17.5.5 matextract

The matextract is used to extract matrices from the matrix stream presented on stdin and writes them to stdout. Matrices are numbered from 1 to N, where N is the number of matrices in the input stream. The matrices are selected by giving their numbers as the arguments to the matextract command. Each argument is comma or space separated list of one or more ranges, where a range is either a number or two numbers separated by a dash (-). A range beginning with - starts with the first matrix and a range ending with - ends with the last matrix. The range - selects all matrices. Here are some examples.

Command	Action
matextract 2	Extract matrix number 2.
matextract 2,5	Extract matrices number 2 and 5.
matextract 2 5	Extract matrices number 2 and 5.
matextract 2-5	Extract matrices number 2 up to and including 5.
matextract -5	Extract matrices 1 to 5.
matextract 2-	Extract all matrices beginning with number 2.
matextract -	Extract all matrices.
matextract 2-4,7 13 15,19-	Extract matrices 2 to 4, 7, 13, 15 and all matrices numbered 19 or higher.

17.5.6 transform

The transform program applies matrices to an object creating a composite object. The matrices are read from stdin and the new object is written to stdout. transform takes one argument, the name of the file holding the object to be transformed. transform is limited to two types of objects, a molecule in PDB format, or a set of points in a text file, three space/tab separated numbers/line. The name of object file is preceded by a flag specifying its type.

Command	Action
transform -pdb X.pdb	Transform a PDB format file.
transform -point X.pts	Transform a set of points.

18 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.

18.1 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.[271, 282] 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:

$$g_{ij} \equiv \mathbf{x}_i \cdot \mathbf{x}_j \tag{18.1}$$

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

$$g_{ij} = 2(d_{i0}^2 + d_{j0}^2 - d_{ij}^2) \tag{18.2}$$

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 **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 **w** and the eigenvalues λ, the metric matrix can be written in two ways:

$$g_{ij} = \sum_{k=1}^{3} x_{ik}x_{jk} = \sum_{k=1}^{3} w_{ik}w_{jk}\lambda_k \tag{18.3}$$

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 nonzero eigenvalues. Eq. (3) then provides an expression for the coordinates $\mathbf{x}_i$ in terms of the eigenvalues and eigenvectors of the metric matrix:

$$x_{ik} = \lambda_k^{1/2} w_{ik} \tag{18.4}$$

If the input distances are not exact, then in general the metric matrix will have more than three nonzero 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.[283, 284] 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.[273] 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.[285] 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,[286] 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.

18.2 Creating and manipulating bounds, embedding structures

A variety of metric-matrix distance geometry routines are included as builtins in nab.

```
bounds newbounds( molecule mol, string opts );
int andbounds( bounds b, molecule mol, string aex1, string aex2,
        float lb, float ub );
int orbounds( bounds b, molecule mol, string aex1, string aex2,
        float lb, float ub );
int setbounds( bounds b, molecule mol, string aex1, string aex2,
        float lb, float ub );
int showbounds( bounds b, molecule mol, string aex1, string aex2 );
int useboundsfrom( bounds b, molecule mol1, string aex1, molecule mol2,
            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 *option*s surrounded by white space. The formats of an option are

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 *-rbm* 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 *nchi* allocates *n* 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 18.1: *Options to newbounds.*

```
-name=value
-name to select the default value if it exists.
```

The options to newbounds() are listed in Table 18.1.

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 (*a*1 in aex1 and *a*2 in aex2) andbounds() sets the lower bound to max (*current_lb, lb*) and the upper bound to the min (*current_ub, ub*). If $ub < current\_lb$ or if $lb > current\_ub$, the bounds for that pair are unchanged. The routine orbounds() works in a similar fashion, except that it uses the less restrictive of the two sets of bounds, rather than the more restrictive one. 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(ˆ3). 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.

keyword	*default*	*meaning*
ddm	none	Dump distance matrix to this file.
rdm	none	Instead of creating a distance matrix, read it from this file.
dmm	none	Dump the metric matrix to this file.
rmm	none	Instead of creating a metric matrix, read it from this file.
gdist	0	If set to nonzero 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 gdist=0, select distances from a uniform distribution in the allowed range.
randpair	0	Use random pair-wise metrization for this percentage of the distances, i.e., randpair=10. would metrize 10% of the distance pairs.

keyword	*default*	*meaning*
eamax	10	Maximum number of embed attempts before bailing out.
seed	-1	Initial seed for the random number generator.
pembed	0	If set to a nonzero value, use the "proximity embedding" scheme of de Groot *et al.*, [26] and Agrafiotis [27], rather than metric matrix embedding.
shuffle	1	Set to 1 to randomize coordinates inside a box of dimension *rbox* at the beginning of the *pembed* scheme; if 0, use whatever coordinates are fed to the routine.
rbox	20.0	Size, in angstroms, of each side of the cubic into which the coordinates are randomly created in the proximity-embed procedure, if *shuffle* is set.
riter	1000	Maximum number of cycles for random-embed procedure. Each cycle selects 1000 pairs for adjustment.
slearn	1.0	Starting value for the learning parameter in proximity embedding; see [27] for details.
kchi	1.0	Force constant for enforcement of chirality constraints.
k4d	1.0	Force constant for squeezing out the fourth dimensional coordinate. If this is nonzero, a penalty function will be added to the bounds-violation energy, which is equal to 0.5 * *k4d* * *w* * *w*, where *w* is the value of the fourth dimensional coordinate.
sqviol	0	If set to nonzero value, use parabolas for the violation energy when upper or lower bounds are violated; otherwise use functions based on those in the dgeom program. See the code in embed.c for details.
lbpen	3.5	Weighting factor for lower-bounds violations, relative to upper-bounds violations. The default penalizes lower bounds 3.5 times as much as the equivalent upper-bounds violations, which is frequently appropriate distance geometry calculations on molecules.
ntpr	10	Frequency at which the bounds matrix violations will be printed in subsequent refinements.
pencut	-1.0	If pencut >= 0.0, individual distance and chirality violations greater than pencut will be printed out (along with the total energy) every ntpr steps.

Typical calling sequences. The following segment shows some ways in which these routines can be put together to do some simple embeds:

```
molecule m;
bounds b;
float fret, xyz[ 10000 ];
int ier;

m = getpdb( argv[2] );
b = newbounds( m, "" );
tsmooth( b, 0.0005 );

dg_options( b, "gdist=1, ntpr=50, k4d=2.0, randpair=10." );
embed( b, xyz );
```

```
ier = conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 200 );
printf( "conjgrad returns %d\\n", ier );

setmol_from_xyzw( m, NULL, xyz );
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.

18.3 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.[287] The templates are in PDB format and are located in $AMBERHOME/dat/dgdb/basepairs/ and $AMBERHOME/dat/dgdb/stacking/.

A typical use of these templates would be to set the bounds between two residues to some percentage of the idealized distance described by the template. In this case, the template would be the reference molecule (the second molecule passed to the function). A typical call might be:

useboundsfrom(b, m, "1:2,3:??,H?",
    getpdb(PATH + "gc.bdna.pdb"), "::??,H?", 0.1);

where PATH is $AMBERHOME/dat/dgdb/stacking/. This call sets the bounds of all the base atoms in residues 2 (GUA) and 3 (CYT) of strand 1 to be within 10% of the distances found in the template.

The basepair templates are named so that the first field of the template name is the one-character initials of the two individual residues and the next field is the Roman numeral corresponding to same bonding scheme described by Sanger, p. 120. *Note: since no specific sugar or backbone conformation is assumed in the templates, the non-base atoms should not be referenced.* The base atoms of the templates are show in figures 18.1 and 18.2.

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 shemes can be found in the *$AMBERHOME/dat/dgdb/stacking* directory.

18.4 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

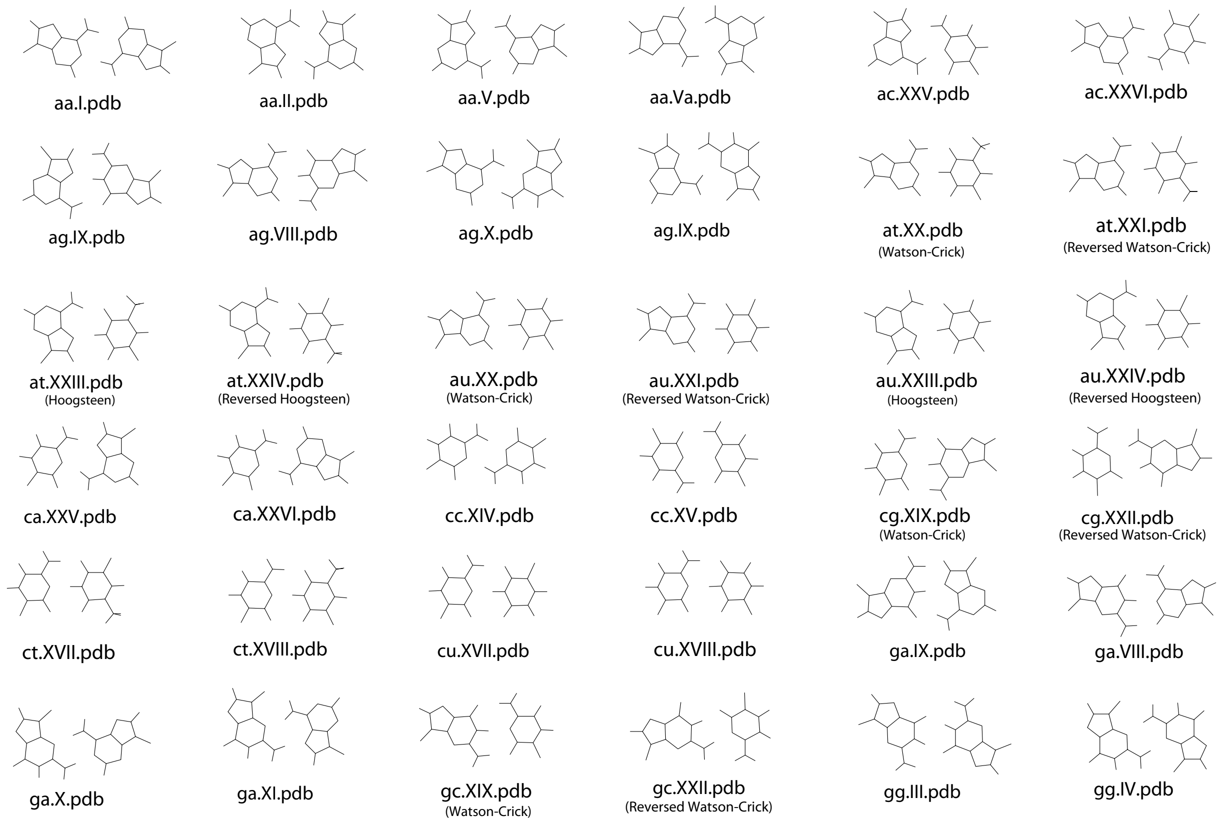


Figure 18.1: *Basepair templates for use with* **usebounds from()**, *(aa-gg)*

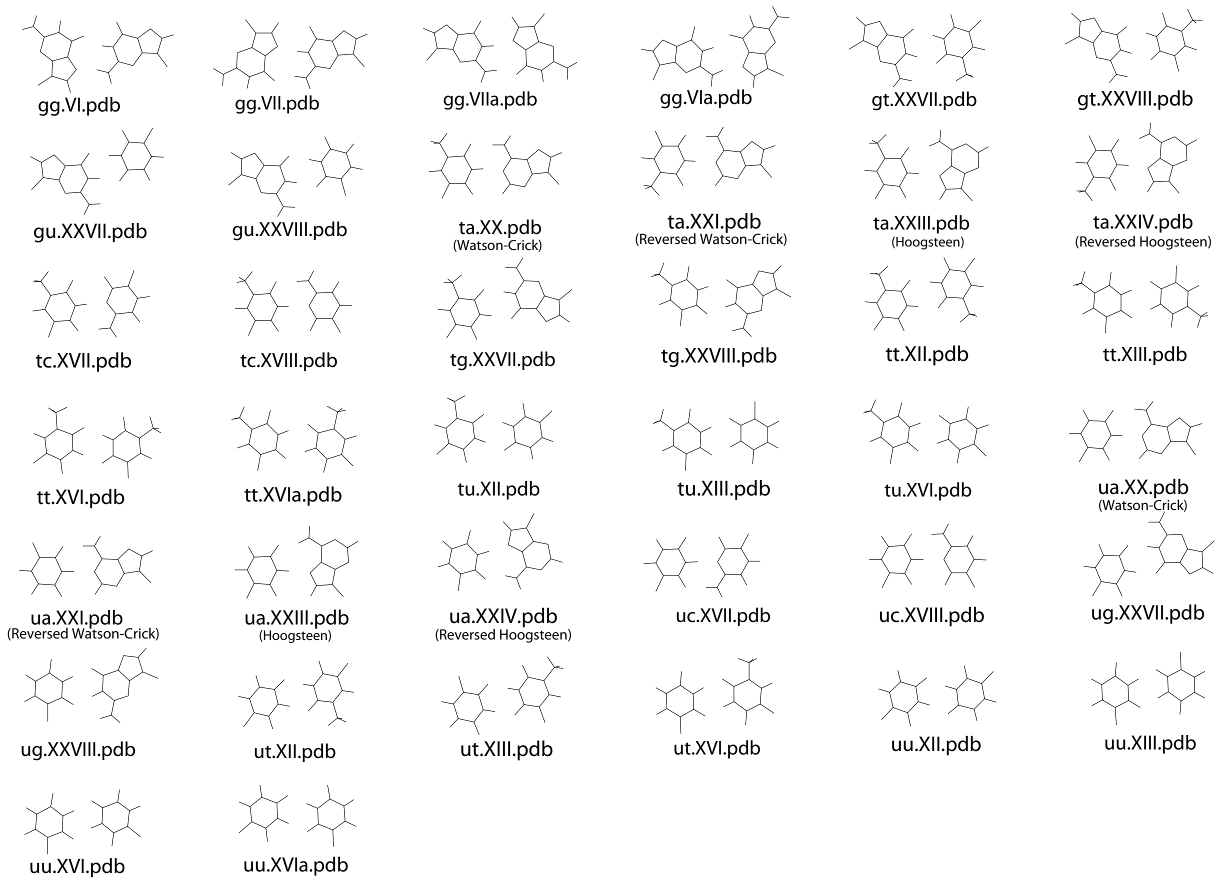


Figure 18.2: *Basepair templates for use with* usebounds from(), *(gg-uu)*

are three types of database files - Those containing bounds information between Watson-Crick basepairs, those containing bounds information between helically stacked residues, and those containing intra-residue bounds information for residues in any conformation. The standard deviation is multiplied by the parameter *mul* and subtracted from the average distance to determine the lower bound and similarly added to the average distance to determine the upper bound of all base-base atom distances. Base-backbone bounds, that is, bounds between pairs of atoms in which one atom is a base atom and the other atom is a backbone atom, are set to be looser than base-base atoms. Specifically, the lower bound between a base-backbone atom pair is set to the smallest measured distance of all the structures considered in creating the database. Similarly, the upper bound between a base-backbone atom pair is set to the largest measured distance of all the structures considered. Base-base, and base-sugar bounds are set in a similar manner. This was done to avoid imposing false constraints on the atomic bounds, since Watson-Crick basepairing and stacking does not preclude any specific backbone and sugar conformation. setboundsfromdb() first searches the current directory for *dbase* before checking the default database location, $AMBERHOME/dat/dgdb.

Each entry in the database file has six fields: The atoms whose bounds are to be set, the number of separate structures sampled in constructing these statistics, the average distance between the two atoms, the standard deviation, the minimum measured distance, and the maximum measured distance. For example, the database bdna.basepair.db has the following sample entries:

```
A:C2-T:C1' 424 6.167 0.198 5.687 6.673
A:C2-T:C2 424 3.986 0.175 3.554 4.505
A:C2-T:C2' 424 7.255 0.304 5.967 7.944
A:C2-T:C3' 424 8.349 0.216 7.456 8.897
A:C2-T:C4 424 4.680 0.182 4.122 5.138
A:C2-T:C4' 424 8.222 0.248 7.493 8.800
A:C2-T:C5 424 5.924 0.168 5.414 6.413
A:C2-T:C5' 424 9.385 0.306 8.273 10.104
A:C2-T:C6 424 6.161 0.163 5.689 6.679
A:C2-T:C7 424 7.205 0.184 6.547 7.658
```

The first column identifies the atoms from the adenosine C2 atom to various thymidine atoms in a Watson-Crick basepair. The second column indicates that 424 structures were sampled in determining the next four columns: the average distance, the standard deviation, and the minimum and maximum distances.

The databases were constructing 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 $AMBERHOME/dat/coords directory. The databases are made by issuing the command $AMBERHOME/-dat/dgdb/make_databases *dblist* where *dblist* is a list of nucleic acid types (i.e., bdna, arna, *etc.*). If one wants to add new structures to the structure repository at $AMBERHOME/dat/coords, it is necessary to make sure that the first two letters of the pdb file identify the nucleic acid type. That is, all bdna pdb files must begin with *bd*.

The nab functions used to create the databases are located in $AMBERHOME/dat/dgdb/functions. The stacking databases were constructed as follows: If two residues stacked 5′ to 3′ in a helix have fewer than ten inter-residue atom distances closer than 2.0 Å or larger than 9.0 Å, and if the normals between the base planes are less than 20.0o, 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 Waton-Crick basepair and is used in the database. A C-G basepair is considered

Watson-Crick paired if the N3-N1 distance is between 2.3 and 3.3 Å, the N4-O6 distance is between 2.3 and 3.2 Å, and the O2-N2 distance is between 2.3 and 3.2 Å.

The nucleotide databases contain all the distance information between atoms in the same residue. No residues in the coordinates directory are excluded from this database. The intent was to allow the residues of this database to assume all possible conformations and ensure that a nucleotide residue would not be biased to a particular conformation.

For the basepair and stacking databases, setting the parameter *mul* to 1.0 results in lower bounds being set from the average database distance minus one standard deviation, and upper bounds as the average database distance plus one standard deviation, between base-base atoms. Base-backbone and base-sugar upper and lower bounds are set to the maximum and minimum measured database values, respectively. *Note, however, that a stacking multiple of 0.0 may not correspond to consistent bounds. A stacking multiple of 0.0 will probably have conflicting bounds information as the bounds information is derived from many different structures.*

The database types are named *nucleic_acid_type.database_type*.db, and can be found in the *$AMBERHOME/dat/dgdb* directory.

18.5 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.nucleic.OL15", "", 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, ntpr=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, ntpr_md=10, tempi=50." );
26  md( 3*m.natoms, 1000, m_xyz, f_xyz, v, mme );
27  setmol_from_xyz( m, NULL, m_xyz );
28  putpdb( "gcgc.md.pdb", m );
```

Line 7 creates an nab molecule; any nab creation method could be used here. Then a temporary pdb file is created, and this is used to generate a NAB molecule that can be used for force-field calculations (line 9). Lines 11-13 allocate some memory, and fill the coordinate array with the molecular position. Lines 15-17 initialize the force field routine, and call it once to get the initial energy. The atom expression "::ZZZ" will match no atoms, so that there will be no restraints on the atoms; hence the fourth argument to mme_init can just be a place-holder, since there are no reference positions for this example. Minimization takes place at line 21, which will call mme repeatedly, and which also arranges for its own printout of results. Finally, in lines 25-28, a short (1000-step) molecular dynamics run is made. Note the the initialization routine mme_init *must* be called before calling the evaluation routines mme or md.

Elaboration of the the above scheme is generally straightforward. For example, a simulated annealing run in which the target temperature is slowly reduced to zero could be written as successive calls to mm_options (setting the temp0 parameter) and md (to run a certain number of steps with the new target temperature.) Note also that routines other than mme could be sent to conjgrad and md: any routine that takes the same three arguments and returns a float function value could be used. In particular, the routines db_viol (to get violations of distance bounds from a bounds matrix) or mme4 (to compute molecular mechanics energies in four spatial dimensions) could be used here. Or, you can write your own nab routine to do this as well. For some examples, see the *gbrna*, *gbrna_long* and *rattle_md* programs in the $AMBERHOME/AmberTools/test/nab directory.

19 NAB: Sample programs

This chapter provides a variety of examples that use the basic NAB functionality described in earlier chapters to solve interesting molecular manipulation problems. Our hope is that the ideas and approaches illustrated here will facilitate construction of similar programs to solve other problems.

19.1 Duplex Creation Functions

nab provides a variety of functions for creating Watson/Crick duplexes. A short description of four of them is given in this section. All four of these functions are written in nab and the details of their implementation is covered in the section **Creating Watson/Crick Duplexes** of the **User Manual**. You should also look at the function fd_helix() to see how to create duplex helices that correspond to fibre-diffraction models. As with the PERL language, "there is more than one way to do it."

```
molecule bdna( string seq );
string wc_complement( string seq, string rlib, string rlt );
molecule wc_helix( string seq, string rlib, string natype, string cseq, string crlib,
    string cnatype, float xoffset, float incl, float twist, float rise, string options );
molecule dg_helix( string seq, string rlib, string natype,
    string cseq, string crlib, string cnatype, float xoffset, float incl, float twist, float rise,
    string options );
molecule wc_basepair( residue res, residue cres );
```

bdna() converts the character string seq containing one or more A, C, G or Ts (or their lower case equivalents) into a uniform ideal Watson/Crick B-form DNA duplex. Each basepair has an X-offset of 2.25 Å, an inclination of -4.96 Å and a helical step of 3.38 Å rise and 36.0o 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 and hydrogens have been added to the two O3' atoms of the three prime bases. bdna() returns NULL if it can not create the molecule.

wc_complement() returns a string that is the Watson/Crick complement of its argument seq. Each C, G, T (U) in seq is replaced by G, C and A. The replacements for A depends if rlt is DNA or RNA. If it is DNA, A is replaced by T. If it is RNA A is replaced by U. wc_complement() considers lower case and upper case letters to be the same and always returns upper case letters. wc_complement() returns NULL on error. Note that 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 the that must be specified as *lower-case* strings, such as "ggact". The nucleic acid type (DNA or RNA) of the sense strand is specified by natype and of the complementary strand cseq by cnatype. Two residue libraries—rlib and crlib— permit creation of DNA:RNA heteroduplexes. If either seq or cseq (but not both) is NULL only the specified strand of what would have been a uniform duplex is created. The options string contains some combination of the strings "s5", "s3", "a5" and "a3"; these indicate which (if any) of the ends of the helices should be "capped" with hydrogens attached to the O5′ atom (in place of a phosphate) if "s5" or "a5" is specified, and a proton added to the O3′ position if "s3" or "a3" is specified. A blank string indicates no capping, which would be appropriate if this section of helix were to be inserted into a larger molecule. The string "s5a5s3a3" would cap the 5′ and 3′ ends of both the "sense" and "anti" strands, leading to a chemically complete molecule. wc_helix() returns NULL on error.

dg_helix() is the functional equivalent of wc_helix() but with the backbone geometry minimized via a distance constraint error function. dg_helix() takes the same arguments as wc_helix().

wc_basepair() assembles two nucleic acid residues (assumed to be in a standard orientation) into a two stranded molecule containing one Watson/Crick base pair. The two strands of the new molecule are "sense" and "anti". It returns NULL on error.

19.2 nab and Distance Geometry

Distance geometry is a method which converts a molecule represented as a set of interatomic distances and related information into a 3-D structure. nab has several builtin functions that are used together to provide metric matrix distance geometry. nab also provides the bounds type for holding a molecule's distance geometry information. A bounds object contains the molecule's interatomic distance bounds matrix and a list of its chiral centers and their volumes. nab uses chiral centers with a volume of 0 to enforce planarity.

Distance geometry has several advantages. It is unique in its power to create structures from very incomplete descriptions. It easily incorporates "low resolution structural data" such as that derived from chemical probing since these kinds of experiments generally return only distance bounds. And it also provides an elegant method by which structures may be described functionally.

The nab distance geometry package is described more fully in the section **NAB Language Reference**. Generally, the function newbounds() creates and returns a bounds object corresponding to the molecule mol. This object contains two things—a distance bounds matrix containing initial upper and lower bounds for every pair of atoms in mol and a initial list of the molecules chiral centers and their volumes. Once a bounds object has been initialized, the modeller uses functions from the middle of the distance geometry function list to tighten, loosen or set other distance bounds and chiralities that correspond to experimental measurements or parts of the model's hypothesis. The four functions andbounds(), orbounds(), setbounds and 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.

19.2.1 Refine DNA Backbone Geometry

As mentioned previously, wc_helix() performs rigid body transformations on residues and does not correct for poor backbone geometry. Using distance geometry, several techniques are available to correct the backbone geometry. In program 7, an 8-basepair dna sequence is created using wc_helix(). A new bounds object is created on line 14, which automatically sets all the 1-2, 1-3, and 1-4 distance bounds information according 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 ˚, 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, "" );
```

```

 b = newbounds(m, "");
 allocate xyz[ 4*m.natoms ];

 useboundsfrom(b, m, "::??,H?[^T']", m, "::??,H?[^T']", 0.0 );
 for ( i = 1; i < m.nresidues/2 ; i = i + 1 ){
        setbounds(b,m, sprintf("1:%d:O3'",i),
                       sprintf("1:%d:P",i+1),    1.595,1.595);
        setbounds(b,m, sprintf("1:%d:O3'",i),
                       sprintf("1:%d:O5'",i+1), 2.469,2.469);
        setbounds(b,m, sprintf("1:%d:C3'",i),
                       sprintf("1:%d:P",i+1),    2.609,2.609);
        setbounds(b,m, sprintf("1:%d:O3'",i),
                       sprintf("1:%d:O1P",i+1), 2.513,2.513);
        setbounds(b,m, sprintf("1:%d:O3'",i),
                       sprintf("1:%d:O2P",i+1), 2.515,2.515);
        setbounds(b,m, sprintf("1:%d:C4'",i),
                       sprintf("1:%d:P",i+1),    3.550,4.107);
        setbounds(b,m, sprintf("1:%d:C2'",i),
                       sprintf("1:%d:P",i+1),    3.550,4.071);
        setbounds(b,m, sprintf("1:%d:C3'",i),
                       sprintf("1:%d:O1P",i+1), 3.050,3.935);
        setbounds(b,m, sprintf("1:%d:C3'",i),
                       sprintf("1:%d:O2P",i+1), 3.050,4.004);
        setbounds(b,m, sprintf("1:%d:C3'",i),
                       sprintf("1:%d:O5'",i+1), 3.050,3.859);
        setbounds(b,m, sprintf("1:%d:O3'",i),
                       sprintf("1:%d:C5'",i+1), 3.050,3.943);

        setbounds(b,m, sprintf("2:%d:P",i+1),
                       sprintf("2:%d:O3'",i),    1.595,1.595);
        setbounds(b,m, sprintf("2:%d:O5'",i+1),
                       sprintf("2:%d:O3'",i),    2.469,2.469);
        setbounds(b,m, sprintf("2:%d:P",i+1),
                       sprintf("2:%d:C3'",i),    2.609,2.609);
        setbounds(b,m, sprintf("2:%d:O1P",i+1),
                       sprintf("2:%d:O3'",i),    2.513,2.513);
        setbounds(b,m, sprintf("2:%d:O2P",i+1),
                       sprintf("2:%d:O3'",i),    2.515,2.515);
        setbounds(b,m, sprintf("2:%d:P",i+1),
                       sprintf("2:%d:C4'",i),    3.550,4.107);
        setbounds(b,m, sprintf("2:%d:P",i+1),
                       sprintf("2:%d:C2'",i),    3.550,4.071);
        setbounds(b,m, sprintf("2:%d:O1P",i+1),
                       sprintf("2:%d:C3'",i),    3.050,3.935);
        setbounds(b,m, sprintf("2:%d:O2P",i+1),
                       sprintf("2:%d:C3'",i),    3.050,4.004);
        setbounds(b,m, sprintf("2:%d:O5'",i+1),
                       sprintf("2:%d:C3'",i),    3.050,3.859);
        setbounds(b,m, sprintf("2:%d:C5'",i+1),
                       sprintf("2:%d:O3'",i),    3.050,3.943);
}
tsmooth( b, 0.0005 );
dg_options(b, "seed=33333, gdist=0, ntpr=100, k4d=4.0" );
setxyzw_from_mol( m, NULL, xyz );
conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 500 );
setmol_from_xyzw( m, NULL, xyz );
```

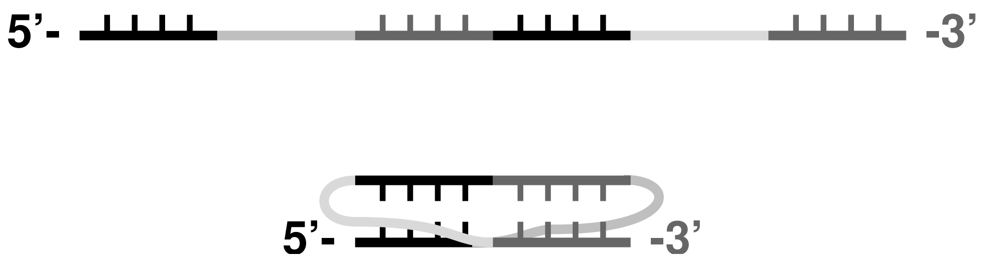


Figure 19.1: *Single-stranded RNA* (top) *folded into a pseudoknot* (bottom). *The black and dark grey base pairs can be stacked.*

```
putpdb( "acgtacgt.pdb", m );
```

The approach of Program 7 is effective but has a disadvantage in that it does not scale linearly with the number of atoms in the molecule. In particular, tsmooth() and conjgrad() require extensive CPU cycles for large numbers of residues. For this reason, the function dg_helix() was created. dg_helix() takes uses the same method of Program 7, but employs a 3-basepair helix template which traverses the new helix as it is being constructed. In this way, the helix is built in a piecewise manner and the maximum number of residues considered in each refinement is less than or equal to six. This is the preferred method of helix construction for large, idealized canonical duplexes.

19.2.2 RNA Pseudoknots

In addition to the standard helix generating functions, nab provides extensive support for generating initial structures from low structural information. As an example, we will describe the construction of a model of an RNA pseudoknot based on a small number of secondary and tertiary structure descriptions. Shen and Tinoco (*J. Mol. Biol.* **247,** 963-978, 1995) used the molecular mechanics program X-PLOR to determine the three dimensional structure of a 34 nucleotide RNA sequence that folds into a pseudoknot. This pseudoknot promotes frame shifting in Mouse Mammary Tumor Virus. A pseudoknot is a single stranded nucleic acid molecule that contains two improperly nested hairpin loops as shown in Figure 19.1. NMR distance and angle constraints were converted into a three dimensional structure using a two stage restrained molecular dynamics protocol. Here we show how a three-dimensional model can be constructed using just a few key features derived from the NMR investigation.

```
// Program 8 - create a pseudoknot using distance geometry
molecule m;
float   xyz[ dynamic ],f[ dynamic ],v[ dynamic ];
bounds  b;
int     i, seqlen;
float   fret;
string  seq, opt;

seq = "GCGGAAACGCCGCGUAAGCG";

seqlen = length(seq);

m = link_na("1", seq, "", "RNA", "35");

allocate xyz[ 4*m.natoms ];
allocate f[ 4*m.natoms ];
allocate v[ 4*m.natoms ];

b = newbounds(m, "");

```

```
21 for ( i = 1; i <= seqlen; i = i + 1) {
22         useboundsfrom(b, m, sprintf("1:%d:??,H?[^'T]", i), m,
23         sprintf("1:%d:??,H?[^'T]", i), 0.0 );
24 }
25 
26 setboundsfromdb(b, m, "1:1:",  "1:2:",  "arna.stack.db",    1.0);
27 setboundsfromdb(b, m, "1:2:",  "1:3:",  "arna.stack.db",    1.0);
28 setboundsfromdb(b, m, "1:3:",  "1:18:", "arna.stack.db",    1.0);
29 setboundsfromdb(b, m, "1:18:", "1:19:", "arna.stack.db",    1.0);
30 setboundsfromdb(b, m, "1:19:", "1:20:", "arna.stack.db",    1.0);
31 
32 setboundsfromdb(b, m, "1:8:",  "1:9:",  "arna.stack.db",    1.0);
33 setboundsfromdb(b, m, "1:9:",  "1:10:", "arna.stack.db",    1.0);
34 setboundsfromdb(b, m, "1:10:", "1:11:", "arna.stack.db",    1.0);
35 setboundsfromdb(b, m, "1:11:", "1:12:", "arna.stack.db",    1.0);
36 setboundsfromdb(b, m, "1:12:", "1:13:", "arna.stack.db",    1.0);
37 
38 setboundsfromdb(b, m, "1:1:",  "1:13:", "arna.basepair.db", 1.0);
39 setboundsfromdb(b, m, "1:2:",  "1:12:", "arna.basepair.db", 1.0);
40 setboundsfromdb(b, m, "1:3:",  "1:11:", "arna.basepair.db", 1.0);
41 
42 setboundsfromdb(b, m, "1:8:",  "1:20:", "arna.basepair.db", 1.0);
43 setboundsfromdb(b, m, "1:9:",  "1:19:", "arna.basepair.db", 1.0);
44 setboundsfromdb(b, m, "1:10:", "1:18:", "arna.basepair.db", 1.0);
45 
46 tsmooth(b, 0.0005);
47 
48 opt = "seed=571, gdist=0, ntpr=50, k4d=2.0, randpair=5., sqviol=1";
49 dg_options( b, opt );
50 embed(b, xyz );
51 
52 conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 500 );
53 
54 setmol_from_xyzw( m, NULL, xyz );
55 putpdb( "rna_pseudoknot.pdb", m );
```

Program 8 uses distance geometry followed by minimization and simulated annealing to create a model of a pseudoknot. Distance geometry code begins in line 20 with the call to newbounds() and ends on line 53 with the call to embed(). The structure created with distance geometry is further refined with molecular dynamics in lines 58-74. Note that very little structural information is given - only connectivity and general base-base interactions. The stacking and base-pair interactions here are derived from NMR evidence, but in other cases might arise from other sorts of experiments, or as a model hypothesis to be tested.

The 20-base RNA sequence is defined on line 9. The molecule itself is created with the link_na() function call which creates an extended conformation of the RNA sequence and caps the 5′ and 3′ ends. Lines 15-17 define arrays that will be used in the simulated annealing of the structure. The bounds object is created in line 19 which automatically sets the 1-2, 1-3, and 1-4 distance bounds in the molecule. The loop in lines 21-24 sets the bounds of each atom in each residue base to the actual distance to every other atom in the same base. This has the effect of enforcing the planarity of the base by treating the base somewhat like a rigid body. In lines 26-44, bounds are set according to information stored in a database. The setboundsfromdb() call sets the bounds from all the atoms in the two specified residues to a 1.0 multiple of the standard deviation of the bounds distances in the specified database. Specifically, line 26 sets the bounds between the base atoms of the first and second residues of strand 1 to be within one standard deviation of a *typical* aRNA stacked pair. Similarly, line 38 sets the bounds between residues 1 and 13 to be that of *typical* Watson-Crick basepairs. For a description of the setboundsfromdb() function, see Chapter 1.

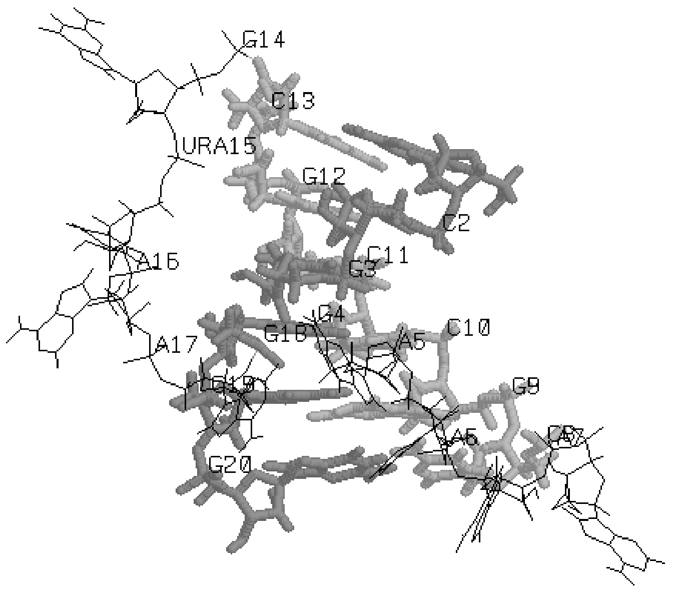


Figure 19.2: *Folded RNA pseudoknot.*

Line 46 smooths the bounds matrix, by attempting to adjust any sets of bounds that violate the triangle equality. Lines 48-49 initialize some distance geometry variables by setting the random number generator seed, declaring the type of distance distribution, how often to print the energy refinement process, declaring the penalty for using a 4th dimension in refinement, and which atoms to use to form the initial metric matrix. The coordinates are calculated and embedded into a 3D coordinate array, *xyz* by the embed() function call on line 50.

The coordinates *xyz* are subject to conjugate gradient refinements in line 52. Line 54 replaces the old molecular coordinates with the new refined ones, and lastly, on line 55, the molecule is saved as "rna_pseudoknot.pdb".

The resulting structure of Program 8 is shown in Figure 19.2. This structure had an final total energy of 46 units. The helical region, shown as polytubes, shows stacking and wc-pairing interactions and a well-defined right-handed helical twist. Of course, good modeling of a "real" pseudoknot would require putting in more constraints, but this example should illustrate how to get started on problems like this.

19.2.3 NMR refinement for a protein

Distance geometry techniques are often used to create starting structures in NMR refinement. Here, in addition to the covalent connections, one makes use of a set of distance and torsional restraints derived from NMR data. While NAB is not (yet?) a fully-functional NMR refinement package, it has enough capabilities to illustrate the basic ideas, and could be the starting point for a flexible procedure. Here we give an illustration of how the rough structure of a protein can be determined using distance geometry and NMR distance constraints; the structures obtained here would then be candidates for further refinement in programs like X-plor or Amber.

The program below illustrates a general procedure for a primarily helical DNA binding domain. Lines 15-22 just construct the sequence in an extended conformation, such that bond lengths and angles are correct, but none of the torsions are correct. The bond lengths and angles are used by newbounds() to construct the "covalent" part of the bounds matrix.

```
//  Program 8a. General driver routine to do distance geometry \fC
//    on proteins, with DYANA-like distance restraints.\fC

#define MAXCOORDS 12000

molecule m;
atom    a;
bounds    b;
int    ier,i, numstrand, ires,jres;
float    fret, rms, ub;
float    xyz[ MAXCOORDS ], f[ MAXCOORDS ], v[ MAXCOORDS ];
file    boundsf;
string  iresname,jresname,iat,jat,aex1,aex2,aex3,aex4,line,dgopts,seq;

//  sequence of the mrf2 protein:
seq = "RADEQAFLVALYKYMKERKTPIERIPYLGFKQINLWTMFQAAQKLGGYETITARRQWKHIY"
    + "DELGGNPGSTSAATCTRRHYERLILPYERFIKGEEDKPLPPIKPRK";

//  build this sequence in an extended conformation, and construct a bounds
//     matrix just based on the covalent structure:
m = linkprot( "A", seq, "" );
b = newbounds( m, "" );

//  read in constraints, updating the bounds matrix using "andbounds":

//  distance constraints are basically those from Y.-C. Chen, R.H.  Whitson
//    Q. Liu, K. Itakura and Y. Chen, "A novel DNA-binding motif shares
//    structural homology to DNA replication and repair nucleases and
//    polymerases,"  Nature Sturct. Biol. 5:959-964 (1998).

boundsf = fopen( "mrf2.7col", "r" );
while( line = getline( boundsf ) ){
    sscanf( line, "%d %s %s %d %s %s %lf", ires, iresname, iat,
        jres, jresname, jat, ub );

//   translations for DYANA-style pseudoatoms:
    if( iat == "HN" ){ iat = "H"; }
    if( jat == "HN" ){ jat = "H"; }

    if( iat == "QA" ){ iat = "CA"; ub += 1.0; }
    if( jat == "QA" ){ jat = "CA"; ub += 1.0; }
    if( iat == "QB" ){ iat = "CB"; ub += 1.0; }
    if( jat == "QB" ){ jat = "CB"; ub += 1.0; }
    if( iat == "QG" ){ iat = "CG"; ub += 1.0; }
    if( jat == "QG" ){ jat = "CG"; ub += 1.0; }
    if( iat == "QD" ){ iat = "CD"; ub += 1.0; }
    if( jat == "QD" ){ jat = "CD"; ub += 1.0; }
    if( iat == "QE" ){ iat = "CE"; ub += 1.0; }
    if( jat == "QE" ){ jat = "CE"; ub += 1.0; }
    if( iat == "QQG" ){ iat = "CB"; ub += 1.8; }
    if( jat == "QQG" ){ jat = "CB"; ub += 1.8; }
    if( iat == "QQD" ){ iat = "CG"; ub += 1.8; }
    if( jat == "QQD" ){ jat = "CG"; ub += 1.8; }
    if( iat == "QG1" ){ iat = "CG1"; ub += 1.0; }
    if( jat == "QG1" ){ jat = "CG1"; ub += 1.0; }
    if( iat == "QG2" ){ iat = "CG2"; ub += 1.0; }
    if( jat == "QG2" ){ jat = "CG2"; ub += 1.0; }
    if( iat == "QD1" ){ iat = "CD1"; ub += 1.0; }
    if( jat == "QD1" ){ jat = "CD1"; ub += 1.0; }
    if( iat == "QD2" ){ iat = "ND2"; ub += 1.0; }
    if( jat == "QD2" ){ jat = "ND2"; ub += 1.0; }
    if( iat == "QE2" ){ iat = "NE2"; ub += 1.0; }
    if( jat == "QE2" ){ jat = "NE2"; ub += 1.0; }

    aex1 = ":" + sprintf( "%d", ires) + ":" + iat;
    aex2 = ":" + sprintf( "%d", jres) + ":" + jat;
    andbounds( b, m, aex1, aex2, 0.0, ub );
}
fclose( boundsf );

//  add in helical chirality constraints to force right-handed helices:
//     (hardwire in locations 1-16, 36-43, 88-92)
for( i=1; i<=12; i++){
    aex1 = ":" + sprintf( "%d", i ) + ":CA";
    aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
    aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
    aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
    setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
}
for( i=36; i<=39; i++){
    aex1 = ":" + sprintf( "%d", i ) + ":CA";
    aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
    aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
    aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
    setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
}
for( i=88; i<=89; i++){
    aex1 = ":" + sprintf( "%d", i ) + ":CA";
    aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
    aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
    aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
    setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
}

//  set up some options for the distance geometry calculation
//    here use the random embed method:
dgopts = "ntpr=10000,rembed=1,rbox=300.,riter=250000,seed=8511135";
dg_options( b, dgopts );

//  do triangle-smoothing on the bounds matrix, then embed:
geodesics( b ); embed( b, xyz );

//  now do conjugate-gradient minimization on the resulting structures:

//  first, weight the chirality constraints heavily:
dg_options( b, "ntpr=20, k4d=5.0, sqviol=0, kchi=50." );
conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.02, 1000., 300 );

//  next, squeeze out the fourth dimension, and increase penalties for
//      distance violations:
dg_options( b, "k4d=10.0, sqviol=1, kchi=50." );
conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.02, 100., 400 );

//  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.protein.ff14SB", "", 0 );
122 setxyz_from_mol( m, NULL, xyz );
123 
124 mm_options( "cut=10.0" );
125 mme_init( m, NULL, "::ZZZ", xyz, NULL );
126 conjgrad( xyz, 3*m.natoms, fret, mme, 0.02, 100., 200 );
127 setmol_from_xyz( m, NULL, xyz );
128 putpdb( argv[3] + ".mm.pdb", m );
```

Once the covalent bounds are created, the the bounds matrix is modified by constraints constructed from an NMR analysis program. This particular example uses the format of the DYANA program, but NAB could be easily modified to read in other formats as well. Here are a few lines from the *mrf2.7col* file:

```
1 ARG+ QB   2 ALA  QB  7.0
4 GLU- HA  93 LYS+ QB  7.0
5 GLN  QB   8 LEU  QQD 9.9
5 GLN  HA   9 VAL  QQG 6.4
85 ILE HA  92 ILE  QD1 6.0
5 GLN  HN   1 ARG+ O   2.0
5 GLN  N    1 ARG+ O   3.0
6 ALA  HN   2 ALA  O   2.0
6 ALA  N    2 ALA  O   3.0
```

The format should be self-explanatory, with the final number giving the upper bound. Code in lines 31-69 reads these in, and translates pseudo-atom codes like "QQD" into atom names. Lines 71-93 add in chirality constraints to ensure right-handed alpha-helices: distance constraints alone do not distinguish chirality, so additions like this are often necessary. The "actual" distance geometry steps take place in line 101, first by triangle-smoothing the bounds, then by embedding them into a three-dimensional object. The structures at this point are actually generally quite bad, so "real-space" refinement is carried out in lines 103-112, and a final short molecular mechanics minimization in lines 119-126.

It is important to realize that many of the structures for the above scheme will get "stuck", and not lead to good structures for the complex. Helical proteins are especially difficult for this sort of distance geometry, since helices (or even parts of helices) start out left-handed, and it is not always possible to easily convert these to right-handed structures. For this particular example, (using different values for the *seed* in line 97), we find that about 30-40% of the structures are "acceptable", in the sense that further refinement in Amber yields good structures.

19.3 Building Larger Structures

While the DNA duplex is locally rather stiff, many DNA molecules are sufficiently long that they can be bent into a wide variety of both open and closed curves. Some examples would be simple closed circles, supercoiled closed circles that have relaxed into circles with twists and the nucleosome core fragment where the duplex itself is wound into a short helix. This section shows how nab can be used to “wrap” DNA around a curve. Three examples are provided: the first produces closed circles with or without supercoiling, the second creates a simple model of the nucleosome core fragment and the third shows how to wind a duplex around a more arbitrary open curve specified as a set of points. The examples are fairly general but do require that the curves be relatively smooth so that the deformation from a linear duplex at each step is small.

Before discussing the examples and the general approach they use, it will be helpful to define some terminology. The helical axis of a base pair is the helical axis defined by an ideal B-DNA duplex that contains that base pair. The base pair plane is the mean plane of both bases. The origin of a base pair is at the intersection the base pair's helical axis and its mean plane. Finally the rise is the distance between the origins of adjacent base pairs.

The overall strategy for wrapping DNA around a curve is to create the curve, find the points on the curve that contain the base pair origins, place the base pairs at these points, oriented so that their helical axes are tangent to the curve and finally rotate the base pairs so that they have the correct helical twist. In all the examples below, the points are chosen so that the rise is constant. This is by no means an absolute requirement, but it does simplify the calculations needed to locate base pairs, and is generally true for the gently bending curves these examples are designed for. In examples 1 and 2, the curve is simple, either a circle or a helix, so the points that locate the base pairs are computed directly. In addition, the bases are rotated about their original helical axes so that they have the correct helical orientation before being placed on the curve.

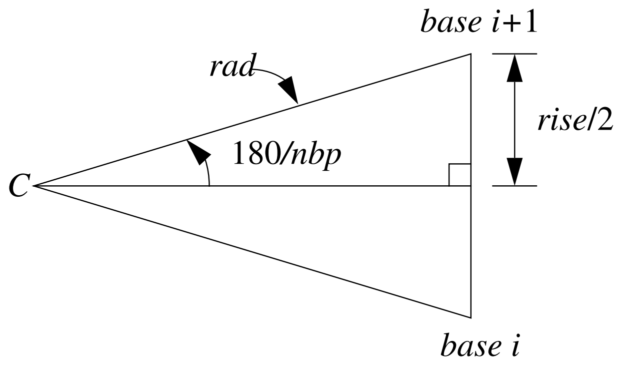
However, this method is inadequate for the more complicated curves that can be handled by example 3. Here each base is placed on the curve so that its helical axis is aligned correctly, but its helical orientation with respect to the previous base is arbitrary. It is then rotated about its helical axis so that it has the correct twist with respect to the previous base.

19.3.1 Closed Circular DNA

This section describes how to use nab to make closed circular duplex DNA with a uniform rise of 3.38 Å. Since the distance between adjacent base pairs is fixed, the radius of the circle that forms the axis of the duplex depends only on the number of base pairs and is given by this rule:

$$rad{=}rise/(2\sin(180/nbp))$$

where *nbp* is the number of base pairs. To see why this is so, consider the triangle below formed by the center of the circle and the centers of two adjacent base pairs. The two long sides are radii of the circle and the third side is the rise. Since the the base pairs are uniformly distributed about the circle the angle between the two radii is $360/nbp$. Now consider the right triangle in the top half of the original triangle. The angle at the center is $180/nbp$, the opposite side is *rise*/2 and *rad* follows from the definition of sin.



In addition to the radius, the helical twist which is a function of the amount of supercoiling must also be computed. In a closed circular DNA molecule, the last base of the duplex must be oriented in such a way that a single helical step will superimpose it on the first base. In circles based on ideal B-DNA, with 10 bases/turn, this requires that the number of base pairs in the duplex be a multiple of 10. Supercoiling adds or subtracts one or more whole turns. The amount of supercoiling is specified by the $\Delta linkingnumber$ which is the number of extra turns to add or subtract. If the original circle had $nbp/10$ turns, the supercoiled circle will have $nbp/10 + \Delta lk$ turns. As each turn represents 360o 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 0o, those steps are skipped for base one. A detailed description follows the code.

```
// Program 9 - Create closed circular DNA.
#define RISE    3.38

int     b, nbp, dlk;
float       rad, twist, ttw;
molecule    m, m1;
matrix      matdx, mattw, matry;
string      sbase, abase;
int     getbase();

if( argc != 3 ){
    fprintf( stderr, "usage: %s nbp dlk\\n", argv[ 1 ] );
    exit( 1 );
}

nbp = atoi( argv[ 2 ] );
if( !nbp || nbp % 10 ){
    fprintf( stderr,
    "%s: Num. of base pairs must be multiple of 10\\n",
        argv[ 1 ] );
    exit( 1 );
}

dlk = atoi( argv[ 3 ] );

twist = ( nbp / 10 + dlk ) * 360.0 / nbp;
rad = 0.5 * RISE / sin( 180.0 / nbp );

matdx = newtransform( rad, 0.0, 0.0, 0.0, 0.0, 0.0 );

m = newmolecule();
addstrand( m, "A" );
addstrand( m, "B" );
ttw = 0.0;
for( b = 1; b <= nbp; b = b + 1 ){

    getbase( b, sbase, abase );

    m1 = wc_helix(
        sbase, "", "dna", abase, "",
        "dna", 2.25, -4.96, 0.0, 0.0 );

    if( b > 1 ){
        mattw = newtransform( 0.,0.,0.,0.,0.,ttw );
        transformmol( mattw, m1, NULL );
    }

    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$\Delta linkingnumber$or the amount of supercoiling. Lines 11-24 process the arguments making sure that they conform to the model's assumptions. In lines 11-14, the code checks that there are exactly three arguments (the nab program's name is argument one), and exits with 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 $\Delta linkingnumber$(dlk) is set in line 24. The helical twist and circle radius are computed in lines 26 and 27 in accordance with the formulas developed above. Line 29 creates a transformation matrix, matdx, that is used to move each base from the global origin along the X-axis to the point where its center intersects the circle.

The circular DNA is built in the molecule variable m, which is initialized and given two strands, "A" and "B" in lines 30-32. The variable ttw in line 34 holds the total twist applied to each base pair The molecule is created in the loop from lines 35-66. The base pair number (b) is converted to the appropriate strings specifying the two nucleotides in this pair. This is done by the function getbase(). This source of this function must be provided by the user who is creating the circles as only he or she will know the actual DNA sequence of the circle. Once the two bases are specified they are passed to the nab builtin wc_helix() which returns a single base pair in the XY plane with its center at the origin. The helical axis of this base pair is on the Z-axis with the 5′-3′ direction oriented in the positive Z-direction.

One or three transformations is required to position this base in its correct place in the circle. It must be rotated about the Z-axis (its helical axis) so that it is one additional unit of twist beyond the previous base. This twist is done in lines 43-46. Since the first base needs 0o 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 0o for base 1, this step is also skipped for the first base.

In lines 56-57, the newly positioned base pair in m1 is added to the growing molecule in m. Note that since the two strands of DNA are antiparallel, the "sense" strand of m1 is added after the last base of the "A" strand of m and the "anti" strand of m1 is added before the first base of the "B" strand of m. For all but the first base, the newly added residues are bonded to the residues they follow (or precede). This is done by the two calls to connectres() in lines 59-60. Again, due to the antiparallel nature of DNA, the new residue in the "A" strand is residue b, but is residue 1 in the "B" strand. In line 63-65, the total twist (ttw) is updated and adjusted to keep in in the range [0,360). After all base pairs have been added the loop exits.

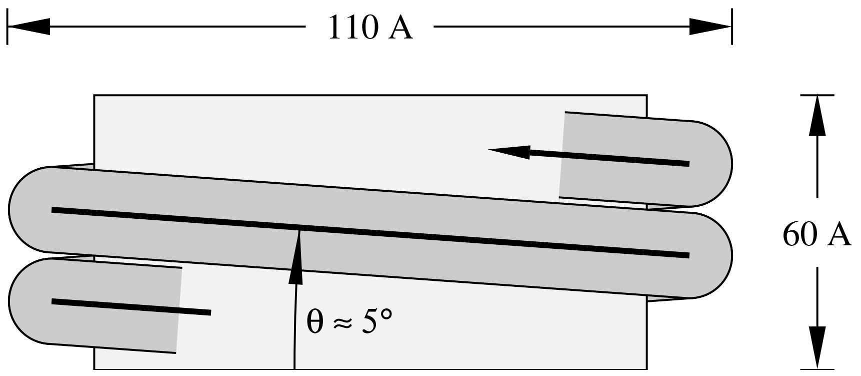
After the loop exit, since this is a *closed* circular molecule the first and last bases of each strand must be bonded and this is done with the two calls to connectres() in lines 67-68. The last step is to save the molecule's coordinates and connectivity in lines 71-72. The nab builtin putpdb() writes the coordinate information in PDB format to the file "circ.pdb" and the nab builtin putbnd() saves the bonding as pairs of integers, one pair/line in the file "circ.bnd", where each integer in a pair refers to an ATOM record in the previously written PDB file.

19.3.2 Nucleosome Model

While the DNA duplex is locally rather stiff, many DNA molecules are sufficiently long that they can be bent into a wide variety of both open and closed curves. Some examples would be simple closed circles, supercoiled closed circles that have relaxed into circles with twists, and the nucleosome core fragment, where the duplex itself is wound into a short helix.

The overall strategy for wrapping DNA around a curve is to create the curve, find the points on the curve that contain the base pair origins, place the base pairs at these points, oriented so that their helical axes are tangent to the curve, and finally rotate the base pairs so that they have the correct helical twist. In the example below, the simplifying assumption is made that the rise is constant at 3.38 Å.

The nucleosome core fragment is composed of duplex DNA wound in a left handed helix around a central protein core. A typical core fragment has about 145 base pairs of duplex DNA forming about 1.75 superhelical turns. Measurements of the overall dimensions of the core fragment indicate that there is very little space between adjacent wraps of the duplex. A side view of a schematic of core particle is shown below.



Computing the points at which to place the base pairs on a helix requires us to spiral an inelastic wire (representing the helical axis of the bent duplex) around a cylinder (representing the protein core). The system is described by four numbers of which only three are independent. They are the number of base pairs n, the number of turns its makes around the protein core t, the "winding" angle θ (which controls how quickly the the helix advances along the axis of the core) and the helix radius r. Both the the number of base pairs and the number of turns around the core can be measured. The leaves two choices for the third parameter. Since the relationship of the winding angle to the overall particle geometry seems more clear than that of the radius, this code lets the user specify the number of turns, the number of base pairs and the winding angle, then computes the helical radius and the displacement along the helix axis for each base pair:

$$d = 3.38\sin(\theta);\ \ \phi = 360t/(n-1) \tag{19.1}$$

$$r = \frac{3.38(n-1)\cos(\theta)}{2\pi t} \tag{19.2}$$

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 $360t$o. The duplex contains $(n-1)$ steps, resulting in $360t/(n-1)$o of rotation between successive base pairs.

```
// Program 10.  Create simple nucleosome model.
#define PI  3.141593
#define RISE    3.38
#define TWIST   36.0
int         b, nbp; int getbase();
float       nt, theta, phi, rad, dy, ttw, len, plen, side;
molecule    m, m1;
matrix      matdx, matrx, maty, matry, mattw;
string      sbase, abase;

nt = atof( argv[ 2 ] );     // number of turns
nbp = atoi( argv[ 3 ] );    // number of base pairs
theta = atof( argv[ 4 ] ); // winding angle

dy = RISE * sin( theta );
phi = 360.0 * nt / ( nbp-1 );
rad = (( nbp-1 )*RISE*cos( theta ))/( 2*PI*nt );

matdx = newtransform( rad, 0.0, 0.0, 0.0, 0.0, 0.0 );
matrx = newtransform( 0.0, 0.0, 0.0, -theta, 0.0, 0.0 );

m = newmolecule();
addstrand( m, "A" ); addstrand( m, "B" );
ttw = 0.0;
for( b = 1; b <= nbp; b = b + 1 ){
    getbase( b, sbase, abase );
    m1 = wc_helix( sbase, "", "dna", abase, "", "dna",
        2.25, -4.96, 0.0, 0.0 );
    mattw = newtransform( 0., 0., 0., 0., 0., ttw );
    transformmol( mattw, m1, NULL );
    transformmol( matrx, m1, NULL );
    transformmol( matdx, m1, NULL );
    maty = newtransform( 0.,dy*(b-1),0., 0.,-phi*(b-1),0.);
    transformmol( maty, m1, NULL );

    mergestr( m, "A", "last", m1, "sense", "first" );
    mergestr( m, "B", "first", m1, "anti", "last" );
    if( b > 1 ){
        connectres( m, "A", b - 1, "O3'", b, "P" );
        connectres( m, "B", 1, "O3'", 2, "P" );
    }
    ttw += TWIST; if( ttw >= 360.0 ) ttw -= 360.0;
}
putpdb( "nuc.pdb", m );
```

Finding the radius of the superhelix is a little tricky. In general a single turn of the helix will not contain an integral number of base pairs. For example, using typical numbers of 1.75 turns and 145 base pairs requires 82.9 base pairs to make one turn. An approximate solution can be found by considering the ideal superhelix that the DNA duplex is wrapped around. Let L be the arc length of this helix. Then $L\cos(\theta)$ is the arc length of its projection into the XZ plane. Since this projection is an overwound circle, L is also equal to $2\pi rt$, where t is the number of turns and r is the unknown radius. Now L is not known but is approximately $3.38(n-1)$. Substituting and solving for $r$gives Eq. 19.2.

The resulting nab code is shown in Program 2. This code requires three arguments—the number of turns, the number of base pairs and the winding angle. In lines 15-17, the helical rise (dy), twist (phi) and radius (rad) are computed according to the formulas developed above.

Two constant transformation matrices, matdx and matrx 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. matrx 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().

19.4 Wrapping DNA Around a Path

This last code develops two nab programs that are used together to wrap B-DNA around a more general open curve specified as a cubic spline through a set of points. The first program takes the initial set of points defining the curve and interpolates them to produce a new set of points with one point at the location of each base pair. The new set of points always includes the first point of the original set but may or may not include the last point. These new points are read by the second program which actually bends the DNA.

The overall strategy used in this example is slightly different from the one used in both the circular DNA and nucleosome codes. In those codes it was possible to directly compute both the orientation and position of each base pair. This is not possible in this case. Here only the location of the base pair's origin can be computed directly. When the base pair is placed at that point its helical axis will be tangent to the curve and point in the right direction, but its rotation about this axis will be arbitrary. It will have to be rotated about its new helical axis to give the proper amount of helical twist to stack it properly on the previous base. Now if the helical twist of a base pair is determined with respect to the previous base pair, either the first base pair is left in an arbitrary orientation, or some other way must be devised to define the helical of it. Since this orientation will depend both on the curve and its ultimate use, this code leaves this task to the user with the result that the helical orientation of the first base pair is undefined.

19.4.1 Interpolating the Curve

This section describes the code that finds the base pair origins along the curve. This program takes an ordered set of points

$$p\,1\,,p\,2\,,...,pn$$

and interpolates it to produce a new set of points

$$np\,1\,,np\,2\,,...,np\;m$$

such that the distance between each np_i and np_{i+1} is constant, in this case equal to 3.38 which is the rise of an ideal B-DNA duplex. The interpolation begins by setting np_1 to p_1 and continues through the p_i until a new point np_m has been found that is within the constant distance to p_n without having gone beyond it.

The interpolation is done via spline() [45] and splint(), two routines that perform a cubic spline interpolation on a tabulated function

$$y_i = f(x_i)$$

In order for spline()/splint() to work on this problem, two things must be done. These functions work on a table of (x_i, y_i) pairs, of which we have only the y_i. However, since the only requirement imposed on the $x_i$is that they be monotonically increasing we can simply use the sequence 1 , 2 , ... , n for the x_i, producing the producing the table (i, y_i). The second difficulty is that spline()/splint() interpolate along a one dimensional curve but we need an interpolation along a three dimensional curve. This is solved by creating three different splines, one for each of the three dimensions.

spline()/splint() perform the interpolation in two steps. The function spline() is called first with the original table and computes the value of the second derivative at each point. In order to do this, the values of the second derivative at two points must be specified. In this code these points are the first and last points of the table, and the values chosen are 0 (signified by the unlikely value of 1e30 in the calls to spline()). After the second derivatives have been computed, the interpolated values are computed using one or more calls to splint().

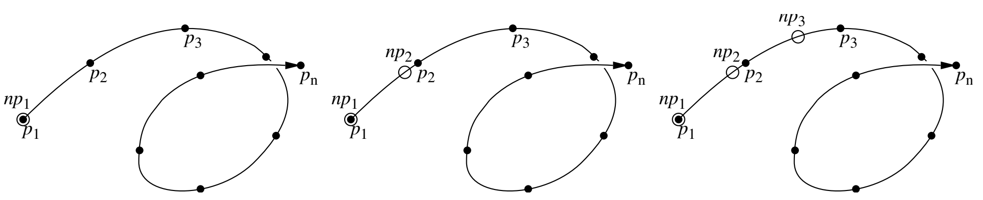
What is unusual about this interpolation is that the points at which the interpolation is to be performed are unknown. Instead, these points are chosen so that the distance between each point and its successor is the constant value RISE, set here to 3.38 which is the rise of an ideal B-DNA duplex. Thus, we have to search for the points and most of the code is devoted to doing this search. The details follow the listing.

```
// Program 11 - Build DNA along a curve
#define RISE    3.38

#define EPS 1e-3
#define APPROX(a,b) (fabs((a)-(b))<=EPS)
#define MAXI    20

#define MAXPTS  150
int npts;
float   a[ MAXPTS ];
float   x[ MAXPTS ], y[ MAXPTS ], z[ MAXPTS ];
float   x2[ MAXPTS ], y2[ MAXPTS ], z2[ MAXPTS ];
float   tmp[ MAXPTS ];

string  line;

int i, li, ni;
float   dx, dy, dz;
float   la, lx, ly, lz, na, nx, ny, nz;
float   d, tfrac, frac;
```

```
21
22 int spline();
23 int splint();
24
25 for( npts = 0; line = getline( stdin ); ){
26     npts = npts + 1;
27     a[ npts ] = npts;
28     sscanf( line, "%lf %lf %lf",
29         x[ npts ], y[ npts ], z[ npts ] );
30 }
31
32 spline( a, x, npts, 1e30, 1e30, x2, tmp );
33 spline( a, y, npts, 1e30, 1e30, y2, tmp );
34 spline( a, z, npts, 1e30, 1e30, z2, tmp );
35
36 li = 1; la = 1.0; lx = x[1]; ly = y[1]; lz = z[1];
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 second derivative values should be 0 at the first and last points of the table. The first point of the interpolated set is set to the first point of the original set and written to stdout in lines 36-37.

The search that finds the new points is lines 39-72. To see how it works consider the figure below. The dots marked $p_1, p_2, \ldots, p_n$ correspond to the original points that define the spline. The circles marked np_1, np_2, np_3 represent the new points at which base pairs will be placed. The curve is a function of the parameter a, which as it ranges from 1 to $npts$ sweeps out the curve from (x_1, y_1, z_1) to $(x_{npts}, y_{npts}, z_{npts})$. Since the original points will in general not be the correct distance apart we have to find new points by interpolating between the original points.



The search works by first finding a point of the original table that is at least RISE distance from the last point found. If the last point of the original table is not far enough from the last point found, the search loop exits and the program ends. However, if the search does find a point in the original table that is at least RISE distance from the last point found, it starts an interpolation loop in lines 47-61 to zero on the best value of a that will produce a new point that is the correct distance from the previous point. After this point is found, the new point becomes the last point and the loop is repeated until the original table is exhausted.

The main search loop uses li to hold the index of the point in the original table that is closest to, but does not pass, the last point found. The loop begins its search for the next point by assuming it will be before the next point in the original table (lines 40-42). It computes the distance between this point (nx,ny,nz) and the last point (lx,ly,lz) in lines 43-44 and then takes one of three actions depending it 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.

19.4.2 Driver Code

This section describes the main routine or driver of the second program which is the actual DNA bender. This routine reads in the points, then calls putdna() (described in the next section) to place base pairs at each point. The points are either read from stdin or from the file whose name is the second command line argument. The source of the points is determined in lines 8-18, being stdin if the command line contained a single arguments or in the second argument if it was present. If the argument count was greater than two, the program prints an error message and exits. The points are read in the loop in lines 20-26. Any line with a # in column 1 is a comment and is ignored. All other lines are assumed to contain three numbers which are extracted from the string, line and stored in the point array pts by the nab builtin sscanf() (lines 23-24). The number of points is kept in npts. Once all points have been read, the loop exits and the point file is closed if it is not stdin. Finally, the points are passed to the function putdna() which will place a base pair at each point and save the coordinates and connectivity of the resulting molecule in the pair of files dna.path.pdb and dna.path.bnd.

```
 1  // Program 12 - DNA bender main program
 2  string      line;
 3  file        pf;
 4  int     npts;
 5  point       pts[ 5000 ];
 6  int     putdna();
 7
 8  if( argc == 1 )
 9      pf = stdin;
10  else if( argc > 2 ){
11      fprintf( stderr, "usage: %s [ path-file ]\\n",
12          argv[ 1 ], argv[ 2 ] );
13      exit( 1 );
14  }else if( !( pf = fopen( argv[ 2 ], "r" ) ) ){
15      fprintf( stderr, "%s: can't open %s\\n",
16          argv[ 1 ], argv[ 2 ] );
17      exit( 1 );
18  }
19
20  for( npts = 0; line = getline( pf ); ){
21      if( substr( line, 1, 1 ) != "#" ){
22          npts = npts + 1;
23          sscanf( line, "%lf %lf %lf",
24              pts[ npts ].x, pts[ npts ].y, pts[ npts ].z );
25      }
26  }
27
28  if( pf != stdin )
29      fclose( pf );
30
31  putdna( "dna.path", pts, npts );
```

19.4.3 Wrap DNA

Every nab molecule contains a frame, a movable handle that can be used to position the molecule. A frame consists of three orthogonal unit vectors and an origin that can be placed in an arbitrary position and orientation with respect to its associated molecule. When the molecule is created its frame is initialized to the unit vectors along the global X, Y and Z axes with the origin at (0,0,0).

nab provides three operations on frames. They can be defined by atom expressions or absolute points (setframe() and setframep()), one frame can be aligned or superimposed on another (alignframe()) and a frame can be placed at a point on an axis (axis2frame()). A frame is defined by specifying its origin, two points that define its X direction and two points that define its Y direction. The Z direction is X×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×X; if Y is chosen then X is recreated from Y×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.

```
// Program 13 - place base pairs on a curve.
point       s_ax[ 4 ];
int     getbase();

int putdna( string mname, point pts[ 1 ], int npts )
{
    int p;
    float   tw;
    residue r;
    molecule    m, m_path, m_ax, m_bp;
    point   p1, p2, p3, p4;
    string  sbase, abase;
    string  aex;
    matrix  mat;

    m_ax = newmolecule();
    addstrand( m_ax, "A" );
    r = getresidue( "AXS", "axes.rlb" );
    addresidue( m_ax, "A", r );
    setxyz_from_mol( m_ax, NULL, s_ax );

    m_path = newmolecule();
    addstrand( m_path, "A" );

    m = newmolecule();
    addstrand( m, "A" );
    addstrand( m, "B" );

    for( p = 1; p < npts; p = p + 1 ){
        setmol_from_xyz( m_ax, NULL, s_ax );
        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 is 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 fours 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 36.0o. The amount of rotation required to give it the correct twist is computed in line 41. A transformation matrix that will rotate the new AXS residue about the ORG—ORG axis by this amount is created in line 42, the atom expression that names the AXS residue is created in line 43 and the residue rotated in line 44. Once the new residue is given the correct twist the frame m_path is moved to the new residue in lines 45-48.

The base pair is added in lines 51-60. The user defined function getbase() converts the point number (p) into the names of the nucleotides needed for this base pair which is created by the nab builtin wc_helix(). It is then placed on the curve in the correct orientation by aligning its frame on the frame of m_path that we have just created (line 55). The new pair is merged into m and bonded with the previous base pair if it exists. After the loop exits, the bend DNA duplex coordinates are saved as a PDB file and the connectivity as a bnd file in the calls to putpdb() and putbnd() in lines 64-65, whereupon putdna() returns to the caller.

19.5 Other examples

There are several additional pedagogical (and useful!) examples in *$AMBERHOME/examples*. These can be consulted to gain ideas of how some useful molecular manipulation programs can be constructed.

- The *peptides* example was created by Paul Beroza to construct peptides with given backbone torsion angles. The idea is to call linkprot to create a peptide in an extended conformation, then to set frames and do rotations to construct the proper torsions. This can be used as just a stand-alone program to perform this task, or as a source for ideas for constructing similar functionality in other nab programs.

- The *suppose* example was created by Jarrod Smith to provide a driver to carry out rms-superpositions. It has a man page that shows how to use it.

Bibliography

[1] Z. Zhang; X. Liu; Z. Chen; H. Zheng; K. Yan; J. Liu. A unified thermostat scheme for efficient configurational sampling for classical/quantum canonical ensembles via molecular dynamics. *J. Chem. Phys.*, **2017**, *147*, 034109.

[2] J. Liu; D. Li; X. Liu. A simple and accurate algorithm for path integral molecular dynamics with the Langevin thermostat. *J. Chem. Phys.*, **2016**, *145*, 024103.

[3] D. Li; X. Han; Y. Chai; C. Wang; Z. Zhang; Z. Chen; J. Liu; J. Shao. Stationary state distribution and efficiency analysis of the Langevin equation via real or virtual dynamics. *J. Chem. Phys.*, **2017**, *147*, 184104.

[4] D. Li; Z. Chen; Z. Zhang; J. Liu. Understanding molecular dynamics with stochastic processes via real or virtual dynamics. *Chinese Journal of Chemical Physics*, **2017**, *30*, 735–760.

[5] Z. Zhang; K. Yan; X. Liu; J. Liu. A leap-frog algorithm-based efficient unified thermostat scheme for molecular dynamics. *Chinese Science Bulletin*, **2018**, *63*, 3467–3483.

[6] C. W. Hopkins; S. Le Grand; R. C. Walker; A. E. Roitberg. Long-Time-Step Molecular Dynamics through Hydrogen Mass Repartitioning. *J. Chem. Theory Comput.*, **2015**, *11*, 1864–1874.

[7] T. Morishita. Fluctuation formulas in molecular-dynamics simulations with the weak coupling heat bath. *J. Chem. Phys.*, **2000**, *113*, 2976.

[8] A. Mudi; C. Chakravarty. Effect of the Berendsen thermostat on the dynamical properties of water. *Mol. Phys.*, **2004**, *102*, 681–685.

[9] B. P. Uberuaga; M. Anghel; A. F. Voter. Synchronization of trajectories in canonical molecular-dynamics simulations: Observation, explanation, and exploitation. *J. Chem. Phys.*, **2004**, *120*, 6363–6374.

[10] D. J. Sindhikara; S. Kim; A. F. Voter; A. E. Roitberg. Bad seeds sprout perilous dynamics: Stochastic thermostat induced trajectory synchronization in biomolecules. *J. Chem. Theory Comput.*, **2009**, *5*, 1624–1631.

[11] B. Leimkuhler; D. T. Margul; M. E. Tuckerman. Stochastic, Resonance-Free Multiple Time-Step Algorithm for Molecular Dynamics with Very Large Time Steps. *Mol. Phys.*, **2013**, *111*, 3579–3594.

[12] R. J. Loncharich; B. R. Brooks; R. W. Pastor. Langevin dynamics of peptides: The frictional dependence of isomerization rates of N-actylananyl-N′-methylamide. *Biopolymers*, **1992**, *32*, 523–535.

[13] Y. M. Rhee; V. S. Pande. Solvent viscosity dependence of the protein folding dynamics. *J. Phys. Chem. B*, **2008**, *112*, 6221–6227. PMID: 18229911.

[14] J. A. Izaguirre; D. P. Catarello; J. M. Wozniak; R. D. Skeel. Langevin stabilization of molecular dynamics. *J. Chem. Phys.*, **2001**, *114*, 2090–2098.

[15] R. Anandakrishnan; A. Drozdetski; R. C. Walker; A. V. Onufriev. Speed of Conformational Change: Comparing Explicit and Implicit Solvent Molecular Dynamics Simulations. *Biophysical Journal*, **2015**, *108*, 1153–1164.

[16] H. C. Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *J. Chem. Phys.*, **1980**, *72*, 2384–2393.

[17] Y. Zhang; S. E. Feller; B. R. Brooks; R. W. Pastor. Computer simulation of liquid/liquid interfaces. I. Theory and application to octane/water. *J. Chem. Phys.*, **1995**, *103*, 10252–10266.

[18] J.-P. Ryckaert; G. Ciccotti; H. J. C. Berendsen. Numerical integration of the cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes. *J. Comput. Phys.*, **1977**, *23*, 327–341.

[19] S. Miyamoto; P. A. Kollman. SETTLE: An analytical version of the SHAKE and RATTLE algorithm for rigid water models. *J. Comput. Chem.*, **1992**, *13*, 952–962.

[20] X. Wu; S. Subramaniam; D. A. Case; K. Wu; B. R. Brooks. Targeted conformational search with map-restrained self-guided langevin dynamics: application to flexible fitting into electron microscopic density maps. *J. Struct. Biology*, **2013**, *183*, 429–440.

[21] P. Li; K. M. Merz, Jr. Taking into Account the Ion-Induced Dipole Interaction in the Nonbonded Model of Ions. *J. Chem. Theory Comput.*, **2014**, *10*, 289–297.

[22] T. Darden; D. York; L. Pedersen. Particle mesh Ewald–an Nlog(N) method for Ewald sums in large systems. *J. Chem. Phys.*, **1993**, *98*, 10089–10092.

[23] U. Essmann; L. Perera; M. L. Berkowitz; T. Darden; H. Lee; L. G. Pedersen. A smooth particle mesh Ewald method. *J. Chem. Phys.*, **1995**, *103*, 8577–8593.

[24] M. F. Crowley; T. A. Darden; T. E. Cheatham, III; D. W. Deerfield, II. Adventures in improving the scaling and accuracy of a parallel molecular dynamics program. *J. Supercomput.*, **1997**, *11*, 255–278.

[25] C. Sagui; T. A. Darden. in *Simulation and Theory of Electrostatic Interactions in Solution*, L. R. Pratt; G. Hummer, Eds., pp 104–113. American Institute of Physics, Melville, NY, 1999.

[26] A. Toukmaji; C. Sagui; J. Board; T. Darden. Efficient particle-mesh Ewald based approach to fixed and induced dipolar interactions. *J. Chem. Phys.*, **2000**, *113*, 10913–10927.

[27] C. Sagui; L. G. Pedersen; T. A. Darden. Towards an accurate representation of electrostatics in classical force fields: Efficient implementation of multipolar interactions in biomolecular simulations. *J. Chem. Phys.*, **2004**, *120*, 73–87.

[28] L. David; R. Luo; M. K. Gilson. Comparison of generalized born and poisson models: Energetics and dynamics of hiv protease. *J. Comput. Chem.*, **2000**, *21*, 295–309.

[29] M. Feig. Kinetics from Implicit Solvent Simulations of Biomolecules as a Function of Viscosity. *J. Chem. Theory Comput.*, **2007**, *3*, 1734–1748.

[30] R. E. Amaro; X. Cheng; I. Ivanov; D. Xu; A. J. Mccammon. Characterizing Loop Dynamics and Ligand Recognition in Human- and Avian-Type Influenza Neuraminidases via Generalized Born Molecular Dynamics and End-Point Free Energy Calculations. *J. Am. Chem. Soc.*, **2009**, *131*, 4702–4709.

[31] B. Zagrovic; V. Pande. Solvent viscosity dependence of the folding rate of a small protein: Distributed computing study. *J. Comput. Chem.*, **2003**, *24*, 1432–1436.

[32] A. V. Onufriev; D. A. Case. Generalized Born implicit solvent models for biomolecules. *Annu. Rev. Biophys.*, **2019**, *48*, 275–296.

[33] J. Weiser; P. S. Shenkin; W. C. Still. Approximate Atomic Surfaces from Linear Combinations of Pairwise Overlaps (LCPO). *J. Comput. Chem.*, **1999**, *20*, 217–230.

[34] W. C. Still; A. Tempczyk; R. C. Hawley; T. Hendrickson. Semianalytical treatment of solvation for molecular mechanics and dynamics. *J. Am. Chem. Soc.*, **1990**, *112*, 6127–6129.

[35] M. Schaefer; M. Karplus. A comprehensive analytical treatment of continuum electrostatics. *J. Phys. Chem.*, **1996**, *100*, 1578–1599.

[36] S. R. Edinger; C. Cortis; P. S. Shenkin; R. A. Friesner. Solvation free energies of peptides: Comparison of approximate continuum solvation models with accurate solution of the Poisson-Boltzmann equation. *J. Phys. Chem. B*, **1997**, *101*, 1190–1197.

[37] B. Jayaram; D. Sprous; D. L. Beveridge. Solvation free energy of biomacromolecules: Parameters for a modified generalized Born model consistent with the AMBER force field. *J. Phys. Chem. B*, **1998**, *102*, 9571–9576.

[38] C. J. Cramer; D. G. Truhlar. Implicit solvation models: Equilibria, structure, spectra, and dynamics. *Chem. Rev.*, **1999**, *99*, 2161–2200.

[39] D. Bashford; D. A. Case. Generalized Born models of macromolecular solvation effects. *Annu. Rev. Phys. Chem.*, **2000**, *51*, 129–152.

[40] A. Onufriev; D. Bashford; D. A. Case. Modification of the generalized Born model suitable for macromolecules. *J. Phys. Chem. B*, **2000**, *104*, 3712–3720.

[41] M. S. Lee; F. R. Salsbury, Jr.; C. L. Brooks, III. Novel generalized Born methods. *J. Chem. Phys.*, **2002**, *116*, 10606–10614.

[42] B. N. Dominy; C. L. Brooks, III. Development of a generalized Born model parameterization for proteins and nucleic acids. *J. Phys. Chem. B*, **1999**, *103*, 3765–3773.

[43] V. Tsui; D. A. Case. Molecular dynamics simulations of nucleic acids using a generalized Born solvation model. *J. Am. Chem. Soc.*, **2000**, *122*, 2489–2498.

[44] N. Calimet; M. Schaefer; T. Simonson. Protein molecular dynamics with the generalized Born/ACE solvent model. *Proteins*, **2001**, *45*, 144–158.

[45] A. Onufriev; D. Bashford; D. A. Case. Exploring protein native states and large-scale conformational changes with a modified generalized Born model. *Proteins*, **2004**, *55*, 383–394.

[46] J. Srinivasan; M. W. Trevathan; P. Beroza; D. A. Case. Application of a pairwise generalized Born model to proteins and nucleic acids: inclusion of salt effects. *Theor. Chem. Acc.*, **1999**, *101*, 426–434.

[47] A. Onufriev; D. A. Case; D. Bashford. Effective Born radii in the generalized Born approximation: The importance of being perfect. *J. Comput. Chem.*, **2002**, *23*, 1297–1304.

[48] G. D. Hawkins; C. J. Cramer; D. G. Truhlar. Parametrized models of aqueous free energies of solvation based on pairwise descreening of solute atomic charges from a dielectric medium. *J. Phys. Chem.*, **1996**, *100*, 19824–19839.

[49] F. M. Richards. Areas, volumes, packing, and protein structure. *Ann. Rev. Biophys. Bioeng.*, **1977**, *6*, 151–176.

[50] M. Schaefer; C. Froemmel. A precise analytical method for calculating the electrostatic energy of macromolecules in aqueous solution. *J. Mol. Biol.*, **1990**, *216*, 1045–1066.

[51] M. Feig; A. Onufriev; M. Lee; W. Im; D. A. Case; C. L. Brooks, III. Performance comparison of the generalized Born and Poisson methods in the calculation of the electrostatic solvation energies for protein structures. *J. Comput. Chem.*, **2004**, *25*, 265–284.

[52] R. Geney; M. Layten; R. Gomperts; C. Simmerling. Investigation of salt bridge stability in a generalized Born solvent model. *J. Chem. Theory Comput.*, **2006**, *2*, 115–127.

[53] A. Okur; L. Wickstrom; C. Simmerling. Evaluation of salt bridge structure and energetics in peptides using explicit, implicit and hybrid solvation models. *J. Chem. Theory Comput.*, **2008**, *4*, 488–498.

[54] A. Okur; L. Wickstrom; M. Layten; R. Geney; K. Song; V. Hornak; C. Simmerling. Improved efficiency of replica exchange simulations through use of a hybrid explicit/implicit solvation model. *J. Chem. Theory Comput.*, **2006**, *2*, 420–433.

[55] D. R. Roe; A. Okur; L. Wickstrom; V. Hornak; C. Simmerling. 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.

[56] H. Nguyen; D. R. Roe; C. Simmerling. Improved Generalized Born Solvent Model Parameters for Protein Simulations. *J. Chem. Theory Comput.*, **2013**, *9*, 2020–2034.

[57] H. Nguyen; J. Maier; H. Huang; V. Perrone; C. Simmerling. Folding simulations for proteins with diverse topologies are accessible in days with a physics-based force field and implicit solvent. *J. Am. Chem. Soc.*, **2014**, *136*, 13959–13962. PMID: 25255057.

[58] H. Nguyen; A. Pérez; S. Bermeo; C. Simmerling. Refinement of Generalized Born Implicit Solvation Parameters for Nucleic Acids and Their Complexes with Proteins. *J. Chem. Theory Comput.*, **2015**, *11*, 3714–3728.

[59] A. Onufriev. in *Modeling Solvent Environments*, M. Feig, Ed., (Wiley, USA). pp 127–165. 2010.

[60] G. D. Hawkins; C. J. Cramer; D. G. Truhlar. Pairwise solute descreening of solute charges from a dielectric medium. *Chem. Phys. Lett.*, **1995**, *246*, 122–129.

[61] M. Schaefer; H. W. T. Van Vlijmen; M. Karplus. Electrostatic contributions to molecular free energies in solution. *Adv. Protein Chem.*, **1998**, *51*, 1–57.

[62] V. Tsui; D. A. Case. Theory and applications of the generalized Born solvation model in macromolecular simulations. *Biopolymers (Nucl. Acid. Sci.)*, **2001**, *56*, 275–291.

[63] C. P. Sosa; T. Hewitt; M. S. Lee; D. A. Case. Vectorization of the generalized Born model for molecular dynamics on shared-memory computers. *J. Mol. Struct. (Theochem)*, **2001**, *549*, 193–201.

[64] J. Mongan; C. Simmerling; J. A. McCammon; D. A. Case; A. Onufriev. Generalized Born with a simple, robust molecular volume correction. *J. Chem. Theory Comput.*, **2007**, *3*, 156–169.

[65] H. Huang; C. Simmerling. Fast Pairwise Approximation of Solvent Accessible Surface Area for Implicit Solvent Simulations of Proteins on CPUs and GPUs. *J. Chem. Theory Comput.*, **2018**, *14*, 5797–5814.

[66] D. Sitkoff; K. A. Sharp; B. Honig. Accurate calculation of hydration free energies using macroscopic solvent models. *J. Phys. Chem.*, **1994**, *98*, 1978–1988.

[67] T. Luchko; S. Gusarov; D. R. Roe; C. Simmerling; D. A. Case; J. Tuszynski; A. Kovalenko. Three-dimensional molecular theory of solvation coupled with molecular dynamics in Amber. *J. Chem. Theory Comput.*, **2010**, *6*, 607–624.

[68] D. Chandler; H. C. Andersen. Optimized cluster expansions for classical fluids. ii. theory of molecular liquids. *J. Chem. Phys.*, **1972**, *57*, 1930–1937.

[69] F. Hirata; P. J. Rossky. An extended RISM equation for molecular polar fluids. *Chem. Phys. Lett.*, **1981**, pp 329–334.

[70] F. Hirata; B. M. Pettitt; P. J. Rossky. Application of an extended rism equation to dipolar and quadrupolar fluids. *J. Chem. Phys.*, **1982**, *77*, 509–520.

[71] F. Hirata; P. J. Rossky; B. M. Pettitt. The interionic potential of mean force in a molecular polar solvent from an extended rism equation. *J. Chem. Phys.*, **1983**, *78*, 4133–4144.

[72] D. Chandler; J. McCoy; S. Singer. Density functional theory of nonuniform polyatomic systems. i. general formulation. *J. Chem. Phys.*, **1986**, *85*, 5971–5976.

[73] D. Chandler; J. McCoy; S. Singer. Density functional theory of nonuniform polyatomic systems. ii. rational closures for integral equations. *J. Chem. Phys.*, **1986**, *85*, 5977–5982.

[74] D. Beglov; B. Roux. Numerical solution of the hypernetted chain equation for a solute of arbitrary geometry in three dimensions. *J. Chem. Phys.*, **1995**, *103*, 360–364.

[75] D. Beglov; B. Roux. An integral equation to describe the solvation of polar molecules in liquid water. *J. Phys. Chem. B*, **1997**, *101*, 7821–7826.

[76] A. Kovalenko; F. Hirata. Three-dimensional density profiles of water in contact with a solute of arbitrary shape: a RISM approach. *Chem. Phys. Lett.*, **1998**, *290*, 237–244.

[77] A. Kovalenko; F. Hirata. 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*, 10095–10112.

[78] A. Kovalenko. In Hirata [288], chapter 4.

[79] A. Kovalenko; F. Hirata. 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.

[80] A. Kovalenko; F. Hirata. 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.

[81] J.-P. Hansen; I. R. McDonald. *Theory of simple liquids*. Academic Press, London, 1990.

[82] F. Hirata. In *Molecular Theory of Solvation* [288], chapter 1.

[83] J. W. Kaminski; S. Gusarov; T. A. Wesolowski; A. Kovalenko. Modeling solvatochromic shifts using the orbital-free embedding potential at statistically mechanically averaged solvent density. *J. Phys. Chem. A*, **2010**, *114*, 6082–6096.

[84] M. Frigo; S. G. Johnson. 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.

[85] M. Frigo. A fast Fourier transform compiler. in *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 34, pp 169–180. ACM, 1999.

[86] S. J. Singer; D. Chandler. Free energy functions in the extended RISM approximation. *Mol. Phys.*, **1985**, *55*, 621–625.

[87] B. M. Pettitt; P. J. Rossky. Alkali halides in water: Ion-solvent correlations and ion-ion potentials of mean force at infinite dilution. *J. Chem. Phys.*, **1986**, *15*, 5836–5844.

[88] S. M. Kast. Free energies from integral equation theories: Enforcing path independence. *Phys. Rev. E*, **2003**, *67*, 041203.

[89] G. Schmeer; A. Maurer. Development of thermodynamic properties of electrolyte solutions with the help of RISM-calculations at the Born-Oppenheimer level. *Phys. Chem. Chem. Phys.*, **2010**, *12*, 2407–2417.

[90] S. Gusarov; T. Ziegler; A. Kovalenko. 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*, 6083–6090.

[91] T. Miyata; F. Hirata. Combination of molecular dynamics method and 3D-RISM theory for conformational sampling of large flexible molecules in solution. *J. Comput. Chem.*, **2007**, *29*, 871–882.

[92] S. M. Kast; T. Kloss. Closed-form expressions of the chemical potential for integral equation closures with certain bridge functions. *J. Chem. Phys.*, **2008**, *129*, 236101.

[93] D. Chandler; Y. Singh; D. M. Richardson. Excess electrons in simple fluids. I. General equilibrium theory for classical hard sphere solvents. *J. Chem. Phys.*, **1984**, *81*, 1975–1982.

[94] T. Ichiye; D. Chandler. Hypernetted chain closure reference interaction site method theory of structure and thermodynamics for alkanes in water. *J. Phys. Chem.*, **1988**, *92*, 5257–5261.

[95] P. H. Lee; G. M. Maggiora. Solvation thermodynamics of polar molecules in aqueous solution by the XRISM method. *J. Phys. Chem.*, **1993**, *97*, 10175–10185.

[96] S. Genheden; T. Luchko; S. Gusarov; A. Kovalenko; U. Ryde. An MM/3D-RISM approach for ligand-binding affinities. *J. Phys. Chem.*, **2010**. Accepted.

[97] J. Johnson; D. A. Case; T. Yamazaki; S. Gusarov; A. Kovalenko; T. Luchko. Small molecule solvation energy and entropy from 3D-RISM. *J. Phys. Condens. Mat.*, **2016**.

[98] G. M. Giambasu; D. A. Case; D. M. York. Predicting Site-Binding Modes of Ions and Water to Nucleic Acids Using Molecular Solvation Theory. *J. Am. Chem. Soc.*, **2019**, *141*, 2435–2445.

[99] G. M. Giambasu; T. Luchko; D. Herschlag; D. M. York; D. A. Case. Ion counting from explicit-solvent simulations and 3d-RISM. *Biophys J*, **2014**, *106*, 883–894.

[100] G. M. Giambasu; M. K. Gebala; M. T. Panteva; T. Luchko; D. A. Case; D. M. York. Competitive Interaction of Monovalent Cations with DNA from 3D-RISM. *Nucleic Acids Res.*, **2015**, *43*, 8405–8415.

[101] R. C. Walker; M. F. Crowley; D. A. Case. The implementation of a fast and accurate QM/MM potential method in Amber. *J. Comput. Chem.*, **2008**, *29*, 1019–1031.

[102] T. J. Giese; D. M. York. Charge-dependent model for many-body polarization, exchange, and dispersion interactions in hybrid quantum mechanical/molecular mechanical calculations. *J. Chem. Phys.*, **2007**, *127*, 194101–194111.

[103] J. J. P. Stewart. Optimization of parameters for semiempirical methods I. Method. *J. Comput. Chem.*, **1989**, *10*, 209–220.

[104] M. J. S. Dewar; E. G. Zoebisch; E. F. Healy; J. J. P. Stewart. AM1: A new general purpose quantum mechanical molecular model. *J. Am. Chem. Soc.*, **1985**, *107*, 3902–3909.

[105] G. B. Rocha; R. O. Freire; A. M. Simas; J. J. P. Stewart. RM1: A Reparameterization of AM1 for H, C, N, O, P, S, F, Cl, Br and I. *J. Comp. Chem.*, **2006**, *27*, 1101–1111.

[106] M. J. S. Dewar; W. Thiel. Ground states of molecules. 38. The MNDO method, approximations and parameters. *J. Am. Chem. Soc.*, **1977**, *99*, 4899–4907.

[107] M. P. Repasky; J. Chandrasekhar; W. L. Jorgensen. PDDG/PM3 and PDDG/MNDO: Improved semiempirical methods. *J. Comput. Chem.*, **2002**, *23*, 1601–1622.

[108] J. P. McNamara; A. M. Muslim; H. Abdel-Aal; H. Wang; M. Mohr; I. H. Hillier; R. A. Bryce. Towards a quantum mechanical force field for carbohydrates: A reparameterized semiempirical MO approach. *Chem. Phys. Lett.*, **2004**, *394*, 429–436.

[109] M. I. Bernal-Uruchurtu; M. F. Ruiz-López. Basic ideas for the correction of semiempirical methods describing H-bonded systems. *Chem. Phys. Lett.*, **2000**, *330*, 118–124.

[110] O. I. Arillo-Flores; M. F. Ruiz-López; M. I. Bernal-Uruchurtu. Can semi-empirical models describe HCl dissociation in water? *Theoret. Chem. Acc.*, **2007**, *118*, 425–435.

[111] W. Thiel; A. A. Voityuk. Extension of the MNDO formalism to d orbitals: Integral approximations and preliminary numerical results. *Theoret. Chim. Acta*, **1992**, *81*, 391–404.

[112] W. Thiel; A. A. Voityuk. Extension of the MNDO formalism to d orbitals: Integral approximations and preliminary numerical results. *Theoret. Chim. Acta*, **1996**, *93*, 315.

[113] W. Thiel; A. A. Voityuk. Erratum: Extension of MNDO to d orbitals: Parameters and results for the second-row elements and for the zinc group. *J. Phys. Chem.*, **1996**, *100*, 616–626.

[114] P. Imhof; F. Noé; S. Fischer; J. C. Smith. AM1/d Parameters for Magnesium in Metalloenzymes. *J. Chem. Theory Comput.*, **2006**, *2*, 1050–1056.

[115] K. Nam; Q. Cui; J. Gao; D. M. York. Specific Reaction Parametrization of the AM1/d Hamiltonian for Phosphoryl Transfer Reactions: H, O, and P Atoms. *J. Chem. Theory Comput.*, **2007**, *3*, 486–504.

[116] J. J. P. Stewart. Optimization of parameters for semiempirical methods V: Modification of NDDO approximations and application to 70 elements. *J. Mol. Mod.*, **2007**, *13*, 1173–1213.

[117] G. M. Seabra; R. C. Walker; M. Elstner; D. A. Case; A. E. Roitberg. 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.

[118] D. Porezag; T. Frauenheim; T. Kohler; G. Seifert; R. Kaschner. Construction of tight-binding-like potentials on the basis of density-functional-theory: Applications to carbon. *Phys. Rev. B*, **1995**, *51*, 12947.

[119] G. Seifert; D. Porezag; T. Frauenheim. Calculations of molecules, clusters and solids with a simplified LCAO-DFT-LDA scheme. *Int. J. Quantum Chem.*, **1996**, *58*, 185.

[120] M. Elstner; D. Porezag; G. Jungnickel; J. Elsner; M. Haugk; T. Frauenheim; S. Suhai; G. Seifert. Self-consistent charge density functional tight-binding method for simulation of complex material properties. *Phys. Rev. B*, **1998**, *58*, 7260.

[121] M. Gaus; Q. Cui; M. Elstner. DFTB3: Extension of the self-consistent-charge density-functional tight-binding method (SCC-DFTB). *J. Chem. Theory Comput.*, **2011**, *7*, 931–948.

[122] M. Elstner; P. Hobza; T. Frauenheim; S. Suhai; E. Kaxiras. Hydrogen bonding and stacking interactions of nucleic acid base pairs: a density-functional-theory based treatment. *J. Chem. Phys.*, **2001**, *114*, 5149.

[123] J. A. Kalinowski; B. Lesyng; J. D. Thompson; C. J. Cramer; D. G. Truhlar. Class IV charge model for the self-consistent charge density-functional tight-binding method. *J. Phys. Chem. A*, **2004**, *108*, 2545–2549.

[124] Y. Yang; H. Yu; D. M. York; Q. Cui; M. Elstner. 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.

[125] M. Korth. Third-generation hydrogen-bonding corrections for semiempirical qm methods and force fields. *J. Chem. Theory Comput.*, **2010**, *6*, 3808.

[126] P. Jurecka; J. Cerný; P. Hobza; D. R. Salahub. Density functional theory augmented with an empirical dispersion term. Interaction energies and geometries of 80 noncovalent complexes compared with ab initio quantum mechanics calculations. *J. Comp. Chem.*, **2007**, *28*, 555–569.

[127] A. Bondi. van der Waals volumes and radii. *J. Phys. Chem.*, **1964**, *68*, 441–451.

[128] M. Korth; M. Pitonak; J. Rezac; P. Hobza. A transferable h-bonding correction for semiempirical quantum-chemical methods. *J. Chem. Theory Comput.*, **2010**, *6*, 344–352.

[129] X. Wu; B. R. Brooks. Self-guided Langevin dynamics simulation method. *Chem. Phys. Lett.*, **2003**, *381*, 512–518.

[130] X. Wu; A. Damjanovic; B. R. Brooks. Efficient and unbiased sampling of biomolecular systems in the canonical ensemble: a review of self-guided langevin dynamics. *Adv. Chem. Phys.*, **2012**, *150*, 255–326.

[131] X. Yu; X. Wu; G. A. Bermejo; B. R. Brooks; J. W. Taraska. Accurate high-throughput structure mapping and prediction with transition metal ion fret. *Structure*, **2013**, *21*, 9–19.

[132] X. Wu; B. R. Brooks. Self-guided langevin dynamics via generalized langevin equation. *J. Comput. Chem.*, **2016**, *37*, 595–601.

[133] X. Wu; B. R. Brooks. Reformulation of the self-guided molecular simulation method. *J. Chem. Phys.*, **2020**, *153*, 094112.

[134] X. Wu; B. R. Brooks. Toward canonical ensemble distribution from self-guided Langevin dynamics simulation. *J. Chem. Phys.*, **2011**, *134*, 134108.

[135] X. Wu; B. R. Brooks. Force-momentum-based self-guided Langevin dynamics: a rapid sampling method that approaches the canonical ensemble . *J. Chem. Phys.*, **2011**, *135*, 204101.

[136] I. Kolossváry; W. C. Guida. 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.

[137] I. Kolossváry; W. C. Guida. Low-mode conformational search elucidated: Application to $C_{39}H_{80}$ and flexible docking of 9-deazaguanine inhibitors into PNP. *J. Comput. Chem.*, **1999**, *20*, 1671–1684.

[138] I. Kolossváry; G. M. Keserü. 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.

[139] G. M. Keserü; I. Kolossváry. Fully flexible low-mode docking: Application to induced fit in HIV integrase. *J. Am. Chem. Soc.*, **2001**, *123*, 12708–12709.

[140] W. H. Press; B. P. Flannery; S. A. Teukolsky; W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, New York, 1989.

[141] D. C. Liu; J. Nocedal. On the limited memory method for large scale optimization. *Math. Programming B*, **1989**, *45*, 503–528.

[142] J. Nocedal; J. L. Morales. Automatic preconditioning by limited memory quasi-Newton updating. *SIAM J. Opt.*, **2000**, *10*, 1079–1096.

[143] P. Kollman. Free energy calculations: Applications to chemical and biochemical phenomena. *Chem. Rev.*, **1993**, *93*, 2395–2417.

[144] T. Simonson. in *Computational Biochemistry and Biophysics*, O. Becker; A. D. MacKerell; B. Roux; M. Watanabe, Eds. Marcel Dekker, New York, 2001.

[145] D. Frenkel; B. Smit. *Understanding Molecular Simulation: From Algorithms to Applications. Second edition*. Academic Press, San Diego, 2002.

[146] T. Steinbrecher; D. A. Case; A. Labahn. A multistep approach to structure-based drug design: Studying ligand binding at the human neutrophil elastase. *J. Med.. Chem.*, **2006**, *49*, 1837–1844.

[147] T. Steinbrecher; A. Hrenn; K. Dormann; I. Merfort; A. Labahn. Bornyl (3,4,5-trihydroxy)-cinnamate - An optimized human neutrophil elastase inhibitor designed by free energy calculations. *Bioorg. Med. Chem.*, **2008**, *16*, 2385–2390.

[148] G. Hummer; A. Szabo. Calculation of free-energy differences from computer simulations of initial and final states. *J. Chem. Phys.*, **1996**, *105*, 2004–2010.

[149] T. Steinbrecher; D. L. Mobley; D. A. Case. Non-linear scaling schemes for Lennard-Jones interactions in free energy calculations. *J. Chem. Phys.*, **2007**, *127*, 214108.

[150] J. Kaus; L. C. T. Pierce; R. C. Walker; J. A. McCammon. PMEMD TI: Placeholder. *J. Chem. Theory Comput.*, **2013**.

[151] T. Steinbrecher; I. Joung; D. A. Case. Soft-core potentials in thermodynamic integration: Comparing one- and two-step transformations. *J. Comp. Chem.*, **2011**, *32*, 3253–3263.

[152] V. Babin; C. Roland; C. Sagui. Adaptively biased molecular dynamics for free energy calculations. *J. Chem. Phys.*, **2008**, *128*, 134101.

[153] T. Huber; A. E. Torda; W. F. van Gunsteren. Local elevation: a method for improving the searching properties of molecular dynamics simulation. *J. Comput. Aided. Mol. Des.*, **1994**, *8*, 695–708.

[154] F. Wang; D. P. Landau. Efficient, multiple-range random walk algorithm to calculate the density of states. *Phys. Rev. Lett.*, **2001**, *86*, 2050–2053.

[155] A. Laio; M. Parrinello. Escaping free-energy minima. *Proc. Natl. Acad. Sci.*, **2002**, *99*, 12562–12566.

[156] M. Iannuzzi; A. Laio; M. Parrinello. Efficient exploration of reactive potential energy surfaces using car-parrinello molecular dynamics. *Phys. Rev. Lett.*, **2003**, *90*, 238302–1.

[157] T. Lelièvre; M. Rousset; G. Stoltz. Computation of free energy profiles with parallel adaptive dynamics. *J. Chem. Phys.*, **2007**, *126*, 134111.

[158] P. Raiteri; A. Laio; F. L. Gervasio; C. Micheletti; M. Parrinello. Efficient reconstruction of complex free energy landscapes by multiple walkers metadynamics. *J. Phys. Chem.*, **2006**, *110*, 3533–3539.

[159] Y. Sugita; A. Kitao; Y. Okamoto. Multidimensional replica-exchange method for free-energy calculations. *J. Chem. Phys.*, **2000**, *113*, 6042–6051.

[160] G. Bussi; F. L. Gervasio; A. Laio; M. Parrinello. Free-energy landscape for β hairpin folding from combined parallel tempering and metadynamics. *J. Am. Chem. Soc.*, **2006**, *128*, 13435–13441.

[161] S. Piana; A. Laio. A bias-exchange approach to protein folding. *J. Phys. Chem. B*, **2007**, *111*, 4553–4559.

[162] V. Babin; C. Roland; T. A. Darden; C. Sagui. The free energy landscape of small peptides as obtained from metadynamics with umbrella sampling corrections. *J. Chem. Phys.*, **2006**, *125*, 2049096.

[163] V. Babin; V. Karpusenka; M. Moradi; C. Roland; C. Sagui. Adaptively biased molecular dynamics: an umbrella sampling method with a time dependent potential. *Inter. J. Quantum Chem.*, **2009**, *109*, 3666–3678.

[164] V. Babin; C. Sagui. Conformational free energies of methyl-alpha-l-iduronic and methyl-beta-d-glucuronic acids in water. *J. Chem. Phys.*, **2010**, *132*, 104108.

[165] M. Moradi; V. Babin; C. Roland; T. Darden; C. Sagui. Conformations and free energy landscapes of polyproline peptides. *Proc. Natl. Aca. Sci. USA*, **2009**, *106*, 20746.

[166] M. Moradi; V. Babin; C. Roland; C. Sagui. A classical molecular dynamics investigation of the free energy and structure of short polyproline conformers. *J. Chem. Phys.*, **2010**, *133*, 125104.

[167] M. Moradi; V. Babin; C. Sagui; C. Roland. in *Proline: Biosynthesis, Regulation and Health Benefits*, B. Nedjimi, Ed., pp 67–110. Nova Publishers, 2013.

[168] M. Moradi; V. Babin; C. Sagui; C. Roland. A statistical analysis of the PPII propensity of amino acid guests in proline-rich peptides. *Biophysical J.*, **2011**, *100*, 1083 – 1093.

[169] M. Moradi; V. Babin; C. Sagui; C. Roland. PPII propensity of multiple-guest amino acids in a proline-rich environment. *J. Phys. Chem. B.*, **2011**, *115*, 8645–8656.

[170] V. Babin; C. Roland; C. Sagui. The alpha-sheet: A missing-in-action secondary structure? *Proteins –Structure Function and Bioinformatics*, **2011**, *79*, 937–946.

[171] M. Moradi; V. Babin; C. Roland; C. Sagui. Are long-range structural correlations behind the aggregation phenomena of polyglutamine diseases? *PLoS Comput. Biol.*, **2012**, *8*, e1002501.

[172] M. Moradi; V. Babin; C. Roland; C. Sagui. Reaction path ensemble of the B-Z-DNA transition: a comprehensive atomistic study. *Nucleic Acids Research*, **2013**, *41*, 33–43.

[173] F. Pan; V. H. Man; C. Roland; C. Sagui. Structure and Dynamics of DNA and RNA Double Helices of CAG and GAC Trinucleotide Repeats. *Biophys. J.*, **2017**, *113*, 19–36.

[174] F. Pan; Y. Zhang; V. H. Man; C. Roland; C. Sagui. E-motif formed by extrahelical cytosine bases in DNA homoduplexes of trinucleotide and hexanucleotide repeats. *Nucl. Acids Res.*, **2018**, *46*, 942–955.

[175] F. Pan; V. H. Man; C. Roland; C. Sagui. Structure and Dynamics of DNA and RNA Double Helices Obtained From the CCG and GGC Trinucleotide Repeats. *J. Phys. Chem. B*, **2018**, *122*, 4491–4512.

[176] P. Xu; F. Pan; C. Roland; C. Sagui; K. Weninger. Dynamics of strand slippage in DNA hairpins formed by CAG repeats: roles of sequence parity and trinucleotide interrupts. *Nucl. Acids Res.*, **2020**, *48*, 2232–2245.

[177] M. Moradi; J.-G. Lee; V. Babin; C. Roland; C. Sagui. Free energy and structure of polyproline peptides: an ab initio and classical molecular dynamics investigation. *Int. J. Quantum Chem.*, **2010**, *110*, 2865–2879.

[178] M. Moradi; C. Sagui; C. Roland. Calculating relative transition rates with driven nonequilibrium simulations. *Chem. Phys. Lett.*, **2011**, *518*, 109.

[179] M. Moradi; C. Sagui; C. Roland. Investigating rare events with nonequilibrium work measurements: I. Nonequilibrium transition paths. *J. Chem. Phys.*, **2014**, *140*, 034114.

[180] M. Moradi; C. Sagui; C. Roland. Investigating rare events with nonequilibrium work measurements: II. Transition and reaction rates. *J. Chem. Phys.*, **2014**, *140*, 034115.

[181] M. Moradi; E. Tajkorshid. Driven Metadynamics: Reconstructing equilibrium free energies from driven adaptive-bias simulations. *J. Phys. Chem. Lett.*, **2013**, *4*, 1882.

[182] A. C. Pan; D. Sezer; B. Roux. Finding transition pathways using the string method with swarms of trajectories. *J. Phys. Chem. B*, **2008**, *112*, 3432–3440.

[183] A. Barducci and G. Bussi and M. Parrinello. Well-tempered metadynamics: a smoothly converging and tunable free energy method. *Phys. Rev. Lett.*, **2008**, *100*, 020603.

[184] K. Minoukadeh and Ch. Chipot and T. Lelievre. Potential of Mean Force Calculations: A multiple-walker adaptive biasing force technique. *J. Chem. Theor. and Comput.*, **2010**, *6*, 1008.

[185] E. A. Coutsias; C. Seok; K. A. Dill. Using quaternions to calculate RMSD. *J. Comput. Chem.*, **2004**, *25*, 1849–1857.

[186] D. K. Coutsias EA, Seok C. Using quaternions to calculate rmsd. *J Comput Chem*, **2004 Nov 30**, *25*, 1849–57.

[187] G. Fiorin; M. L. Klein; J. Hénin. Using collective variables to drive molecular dynamics simulations. *Molecular Physics*, **2013**, *111*, 3345–3362.

[188] M. Moradi; E. Tajkhorshid. Mechanistic picture for conformational transition of a membrane transporter at atomic resolution. *Proc. Natl. Acad. Sci. U.S.A.*, **2013**, *110*, 18916–18921.

[189] M. Moradi; E. Tajkhorshid. Computational Recipe for Efficient Description of Large-Scale Conformational Changes in Biomolecular Systems. *J Chem Theory Comput*, **2014**, *10*, 2866–2880.

[190] S. Park; F. Khalili-Araghi; E. Tajkhorshid; K. Schulten. Free energy calculation from steered molecular dynamics simulations using Jarzynski's equality. *J. Chem. Phys.*, **2003**, *119*, 3559–3566.

[191] M. Matsumoto; T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, **1998**, *8*, 3–30.

[192] L. Maragliano; A. Fischer; E. Vanden-Eijnden; G. Ciccotti. String method in collective variables: Minimum free energy paths and isocommittor surfaces. *J. Chem. Phys.*, **2006**, *125*, 024106.

[193] B. M. Duggan; G. B. Legge; H. J. Dyson; P. E. Wright. SANE (Structure Assisted NOE Evaluation): An automated model-based approach for NOE assignment. *J. Biomol. NMR*, **2001**, *19*, 321–329.

[194] A. Kalk; H. J. C. Berendsen. Proton magnetic relaxation and spin diffusion in proteins. *J. Magn. Reson.*, **1976**, *24*, 343–366.

[195] E. T. Olejniczak; M. A. Weiss. Are methyl groups relaxation sinks in small proteins? *J. Magn. Reson.*, **1990**, *86*, 148–155.

[196] K. J. Cross; P. E. Wright. Calibration of ring-current models for the heme ring. *J. Magn. Reson.*, **1985**, *64*, 220–231.

[197] K. Ösapay; D. A. Case. A new analysis of proton chemical shifts in proteins. *J. Am. Chem. Soc.*, **1991**, *113*, 9436–9444.

[198] D. A. Case. Calibration of ring-current effects in proteins and nucleic acids. *J. Biomol. NMR*, **1995**, *6*, 341–346.

[199] L. Banci; I. Bertini; G. Gori-Savellini; A. Romagnoli; P. Turano; M. A. Cremonini; C. Luchinat; H. B. Gray. Pseudocontact shifts as constraints for energy minimization and molecular dynamics calculations on solution structures of paramagnetic metalloproteins. *Proteins*, **1997**, *29*, 68.

[200] C. R. Sanders, II; B. J. Hare; K. P. Howard; J. H. Prestegard. Magnetically-oriented phospholipid micelles as a tool for the study of membrane-associated molecules. *Prog. NMR Spectr.*, **1994**, *26*, 421–444.

[201] V. Tsui; L. Zhu; T. H. Huang; P. E. Wright; D. A. Case. Assessment of zinc finger orientations by residual dipolar coupling constants. *J. Biomol. NMR*, **2000**, *16*, 9–21.

[202] D. A. Case. Calculations of NMR dipolar coupling strengths in model peptides. *J. Biomol. NMR*, **1999**, *15*, 95–102.

[203] G. P. Gippert; P. F. Yip; P. E. Wright; D. A. Case. Computational methods for determining protein structures from NMR data. *Biochem. Pharm.*, **1990**, *40*, 15–22.

[204] D. A. Case; P. E. Wright. in *NMR in Proteins*, G. M. Clore; A. Gronenborn, Eds., pp 53–91. MacMillan, New York, 1993.

[205] D. A. Case; H. J. Dyson; P. E. Wright. Use of chemical shifts and coupling constants in nuclear magnetic resonance structural studies on peptides and proteins. *Meth. Enzymol.*, **1994**, *239*, 392–416.

[206] R. Brüschweiler; D. A. Case. Characterization of biomolecular structure and dynamics by NMR cross-relaxation. *Prog. NMR Spectr.*, **1994**, *26*, 27–58.

[207] D. A. Case. The use of chemical shifts and their anisotropies in biomolecular structure determination. *Curr. Opin. Struct. Biol.*, **1998**, *8*, 624–630.

[208] A. E. Torda; R. M. Scheek; W. F. VanGunsteren. Time-dependent distance restraints in molecular dynamics simulations. *Chem. Phys. Lett.*, **1989**, *157*, 289–294.

[209] D. A. Pearlman; P. A. Kollman. Are time-averaged restraints necessary for nuclear magnetic resonance refinement? A model study for DNA. *J. Mol. Biol.*, **1991**, *220*, 457–479.

[210] A. E. Torda; R. M. Brunne; T. Huber; H. Kessler; W. F. van Gunsteren. Structure refinement using time-averaged J-coupling constant restraints. *J. Biomol. NMR*, **1993**, *3*, 55–66.

[211] D. A. Pearlman. How well to time-averaged J-coupling restraints work? *J. Biomol. NMR*, **1994**, *4*, 279–299.

[212] D. A. Pearlman. How is an NMR structure best defined? An analysis of molecular dynamics distance-based approaches. *J. Biomol. NMR*, **1994**, *4*, 1–16.

[213] X. Wu; J. L. Milne; M. J. Borgnia; A. V. Rostapshov; S. Subramaniam; B. R. Brooks. A core-weighted fitting method for docking atomic structures into low-resolution maps: application to cryo-electron microscopy. *J Struct Biol*, **2003**, *141*, 63–76.

[214] J. L. Milne; X. Wu; M. J. Borgnia; J. S. Lengyel; B. R. Brooks; D. Shi; R. N. Perham; S. Subramaniam. Molecular structure of a 9-MDa icosahedral pyruvate dehydrogenase subcomplex containing the E2 and E3 enzymes using cryoelectron microscopy. *J Biol Chem*, **2006**, *281*, 4364–70.

[215] X. Wu; B. R. Brooks. Modeling of Macromolecular assemblies with map objects. *Proc. 2007 Int. Conf. Bioinform. Comput. Biol.*, **2007**, *II*, 411–417.

[216] C. M. Khursigara; X. Wu; P. Zhang; J. Lefman; S. Subramaniam. Role of HAMP domains in chemotaxis signaling by bacterial chemoreceptors. *PNAS*, **2008**, *105*, 16555–60.

[217] J. S. Lengyel; K. M. Stott; X. Wu; A. Brooks, B. R.and Balbo; P. Schuck; R. N. Perham; S. Subramaniam; J. L. Milne. Extended polypeptide linkers establish the spatial architecture of a pyruvate dehydrogenase multienzyme complex. *Structure*, **2008**, *16*, 93–103.

[218] C. M. Khursigara; X. Wu; ; S. Subramaniam. Chemoreceptors in Caulobacter crescentus: trimers of receptor dimers in a partially ordered hexagonally packed array. *J Bacteriol*, **2008**, *190*, 6805–10.

[219] J. Elegheert; A. Desfosses; A. V. Shkumatov; X. Wu; N. Bracke; K. Verstraete; K. Van Craenenbroeck; B. R. Brooks; D. I. Svergun; B. Vergauwen; I. Gutsche; S. N. Savvides. Extracellular complexes of the hematopoietic human and mouse CSF-1 receptor are driven by common assembly principles . *Structure*, **2011**, *19*, 1762–72.

[220] C. M. Khursigara; G. Lan; S. Neumann; X. Wu; S. Ravindran; M. J. Borgnia; V. Sourjik; J. Milne; Y. Tu; S. Subramaniam. Lateral density of receptor arrays in the membrane plane influences sensitivity of the E. coli chemotaxis response. *Embo J*, **2011**, *30*, 1719–29.

[221] X. Wu; B. R. Brooks. in *Microscopy: advances in scientific research and education*, A. Mendez-Vilas, Ed., pp 39–47. Formatex Research Center, Spain, 2014.

[222] X. Wu; B. R. Brooks. in *Modern electron microscopy in physical and life science*, M. Janecek, Ed., chapter 12, pp 243–262. InTech, 2016.

[223] A. Bartesaghi; A. Merk; S. Banerjee; D. Matthies; X. Wu; J. L. S. Milne; S. Subramaniam. 2.2 a resolution cryo-em structure of beta-galactosidase in complex with a cell-permeant inhibitor. *Science*, **2015**, *348*, 1147–1151.

[224] D. S. Cerutti; D. A. Case. Molecular dynamics simulations of macromolecular crystals. *Wires Comput. Mol. Sci.*, **2018**, *8*, e1402.

[225] P. A. Janowski; D. S. Cerutti; J. M. Holton; D. A. Case. Peptide crystal simulations reveal hidden dynamics. *J. Am. Chem. Soc.*, **2013**, *135*, 7938–7948.

[226] N. Moriarty; P. Janowski; J. Swails; H. Nguyen; J. Richardson; D. Case; P. Adams. Improved chemistry restraints for crystallographic refinement by integrating the Amber force field into Phenix. *Acta Cryst. D*, **2020**, *76*, 51–62.

[227] P. Afonine; A. Urzhumtsev. On a fast calculation of structure factors at a subatomic resolution. *Acta Cryst. A*, **2004**, *60*, 19–32.

[228] J. Jiang; A. Brünger. Protein hydration observed by X-ray diffraction: solvation properties of penicillopepsin and neuraminidase crystal structures. *J. Mol. Biol.*, **1994**, *243*, 100–115.

[229] A. Fokine; A. Urzhumtsev. Flat bulk-solvent model: obtaining optimal parameters. *Acta Cryst. D*, **2002**, *58*, 1387–1392.

[230] R. Grosse-Kunstleve; N. Sauter; N. Moriarty; P. Adams. The Computational Crystallography Toolbox: crystallographic algorithms in a reusable software framework. *J. Appl. Crystallogr.*, **2002**, *35*, 126–136.

[231] P. Afonine; R. Grosse-Kunstleve; P. Adams; A. Urzhumtsev. Bulk-solvent and overall scaling revisited: faster calculations, improved results. *Acta Cryst. D*, **2013**, *69*, 625–634.

[232] V. Lunin; T. Skovoroda. R-free likelihood-based estimates of errors for phases calculated from atomic models. *Acta Cryst. A*, **1995**, *51*, 880–887.

[233] P. V. Afonine; R. W. Grosse-Kunstleve; P. D. Adams. A robust bulk-solvent correction and anisotropic scaling procedure. *Acta Cryst. D*, **2005**, *61*, 850–855.

[234] Y. Xue; N. Skrynnikov. Ensemble MD simulations restrained via crystallographic data: accurate structure leads to accurate dynamics. *Prot. Sci.*, **2014**, *23*, 488–507.

[235] A. Raval; S. Piana; M. Eastwood; R. Dror; D. Shaw. Refinement of protein structure homology models via long, all-atom molecular dynamics simulations. *Proteins*, **2012**, *80*, 2071–2079.

[236] O. Mikhailovskii; Y. Xue; N. R. Skrynnikov. Modeling a unit cell: crystallographic refinement procedure using the biomolecular MD simulation platform *Amber*. *IUCrJ*, **2022**, *9*, 114–133.

[237] J. Schmit; N. Kariyawasam; V. Needham; P. Smith. SLTCAP: A Simple Method for Calculating the Number of Ions Needed for MD Simulation. *J. Chem. Theory Comput.*, **2018**, *14*, 1823–1827.

[238] M. Machado; S. Pantano. Split the Charge Difference in Two! A Rule of Thumb for Adding Proper Amounts of Ions in MD Simulations. *J. Chem. Theory Comput.*, **2020**, *16*, 1367–1372.

[239] J. Wang; R. M. Wolf; J. W. Caldwell; P. A. Kollamn; D. A. Case. Development and testing of a general Amber force field. *J. Comput. Chem.*, **2004**, *25*, 1157–1174.

[240] J. Wang; W. Wang; P. A. Kollman; D. A. Case. Automatic atom type and bond type perception in molecular mechanical. *J. Mol. Graphics Model.*, **2006**, *25*, 247–260.

[241] A. Jakalian; B. L. Bush; D. B. Jack; C. I. Bayly. Fast, efficient generation of high-quality atomic charges. AM1-BCC model: I. Method. *J. Comput. Chem.*, **2000**, *21*, 132–146.

[242] A. Jakalian; D. B. Jack; C. I. Bayly. Fast, efficient generation of high-quality atomic charges. AM1-BCC model: II. Parameterization and Validation. *J. Comput. Chem.*, **2002**, *23*, 1623–1641.

[243] X. He; V. H. Man; W. Wang; T. S. Lee; J. Wang. A fast and high-quality charge model for the next generation general AMBER force field. *J. Chem. Phys.*, **2020**, *153*, 114502.

[244] Y. Sun; X. He; T. Hou; L. Cai; V. H. Man; J. Wang. Development and test of highly accurate endpoint free energy methods. 1: Evaluation of ABCG2 charge model on solvation free energy prediction and optimization of atom radii suitable for more accurate solvationÂ freeÂ energy prediction by the PBSA method. *J Comput Chem*, **2023**, *44*, 1334–1346.

[245] C. I. Bayly; P. Cieplak; W. D. Cornell; P. A. Kollman. A well-Behaved electrostatic potential based method using charge restraints for determining atom-centered charges: The RESP model. *J. Phys. Chem.*, **1993**, *97*, 10269–10280.

[246] J. Wang; P. A. Kollman. Automatic parameterization of force field by systematic search and genetic algorithms. *J. Comput. Chem.*, **2001**, *22*, 1219–1228.

[247] A. P. Graves; D. M. Shivakumar; S. E. Boyce; M. P. Jacobson; D. A. Case; B. K. Shoichet. Rescoring docking hit lists for model cavity sites: Predictions and experimental testing. *J. Mol. Biol.*, **2008**, *377*, 914–934.

[248] B. Jojart; T. A. Martinek. Performance of the general amber force field in modeling aqueous POPC membrane bilayers. *J. Comput. Chem.*, **2007**, *28*, 2051–2058.

[249] L. Rosso; I. R. Gould. Structure and dynamics of phospholipid bilayers using recently developed general all-atom force fields. *J. Comput. Chem.*, **2008**, *29*, 24–37.

[250] J. Wang; T. Hou. Application of Molecular Dynamics Simulations in Molecular Property Prediction. 1. Density and Heat of Vaporization. *J. Chem. Theory Comput.*, **2011**, *7*, 2151–2165.

[251] X. He; B. Walker; W. H. Man; P. Ren; J. Wang. Recent progress in general force fields of small molecules. *Curr. Opin. Struc. Biol.*, **2022**, *72*, 187–193.

[252] R. Brown; D. Case. Second derivatives in generalized Born theory. *J. Comput. Chem.*, **2006**, *27*, 1662–1675.

[253] H. J. C. Berendsen; J. P. M. Postma; W. F. van Gunsteren; A. DiNola; J. R. Haak. Molecular dynamics with coupling to an external bath. *J. Chem. Phys.*, **1984**, *81*, 3684–3690.

[254] C. Brooks; A. Brünger; M. Karplus. Active site dynamics in protein molecules: A stochastic boundary molecular-dynamics approach. *Biopolymers*, **1985**, *24*, 843–865.

[255] D. T. Nguyen; D. A. Case. On finding stationary states on large-molecule potential energy surfaces. *J. Phys. Chem.*, **1985**, *89*, 4020–4026.

[256] Z. J. Shi; J. Shen. New inexact line search method for unconstrained optimization. *J. Optim. Theory Appl.*, **2005**, *127*, 425–446.

[257] S. Izadi; R. Anandakrishnan; A. V. Onufriev. Implicit solvent model for Million-Atom atomistic simulations: Insights into the organization of 30-nm chromatin fiber. *J. Chem. Theory Comput.*, **2016**, *12*, 5946–5959.

[258] R. Anandakrishnan; A. V. Onufriev. An N log N approximation based on the natural organization of biomolecules for speeding up the computation of long range interactions. *J. Comput. Chem.*, **2010**, *31*, 691–706.

[259] R. Anandakrishnan; M. Daga; A. V. Onufriev. An n log n Generalized Born Approximation. *J. Chem. Theory Comput.*, **2011**, *7*, 544–559.

[260] R. Anandakrishnan; C. Baker; S. Izadi; A. V. Onufriev. Point charges optimally placed to represent the multipole expansion of charge distributions. *PloS one*, **2013**, *8*, e67715.

[261] F. Major; M. Turcotte; D. Gautheret; G. Lapalme; E. Fillon; R. Cedergren. The Combination of Symbolic and Numerical Computation for Three-Dimensional Modeling of RNA. *Science*, **1991**, *253*, 1255–1260.

[262] D. Gautheret; F. Major; R. Cedergren. Modeling the three-dimensional structure of RNA using discrete nucleotide conformational sets. *J. Mol. Biol.*, **1993**, *229*, 1049–1064.

[263] M. Turcotte; G. Lapalme; F. Major. Exploring the conformations of nucleic acids. *J. Funct. Program.*, **1995**, *5*, 443–460.

[264] D. A. Erie; K. J. Breslauer; W. K. Olson. A Monte Carlo Method for Generating Structures of Short Single-Stranded DNA Sequences. *Biopolymers*, **1993**, *33*, 75–105.

[265] C.-S. Tung; E. S. Carter, II. Nucleic acid modeling tool (NAMOT): an interactive graphic tool for modeling nucleic acid structures. *CABIOS*, **1994**, *10*, 427–433.

[266] E. S. Carter, II; C.-S. Tung. NAMOT2–a redesigned nucleic acid modeling tool: construction of non-canonical DNA structures. *CABIOS*, **1996**, *12*, 25–30.

[267] V. B. Zhurkin; Y. P. Lysov; V. I. Ivanov. Different Families of Double Stranded Conformations of DNA as Revealed by Computer Calculations. *Biopolymers*, **1978**, *17*, 277–312.

[268] R. Lavery; K. Zakrzewska; H. Skelnar. JUMNA (junction minimisation of nucleic acids). *Comp. Phys. Commun.*, **1995**, *91*, 135–158.

[269] J. Gabarro-Arpa; J. A. H. Cognet; M. Le Bret. 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.

[270] M. Le Bret; J. Gabarro-Arpa; J. C. Gilbert; C. Lemarechal. MORCAD an object-oriented molecular modeling package. *J. Chim. Phys.*, **1991**, *88*, 2489–2496.

[271] G. M. Crippen; T. F. Havel. *Distance Geometry and Molecular Conformation*. Research Studies Press, Taunton, England, 1988.

[272] D. C. Spellmeyer; A. K. Wong; M. J. Bower; J. M. Blaney. Conformational analysis using distance geometry methods. *J. Mol. Graph. Model.*, **1997**, *15*, 18–36.

[273] M. E. Hodsdon; J. W. Ponder; D. P. Cistola. 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.

[274] T. Macke; S.-M. Chen; W. J. Chazin. in *Structure and Function, Volume 1: Nucleic Acids*, R. H. Sarma; M. H. Sarma, Eds., pp 213–227. Adenine Press, Albany, 1992.

[275] B. C. M. Potts; J. Smith; M. Akke; T. J. Macke; K. Okazaki; H. Hidaka; D. A. Case; W. J. Chazin. The structure of calcyclin reveals a novel homodimeric fold S100 Ca^{2+}-binding proteins. *Nature Struct. Biol.*, **1995**, *2*, 790–796.

[276] J. J. Love; X. Li; D. A. Case; K. Giese; R. Grosschedl; P. E. Wright. 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.

[277] R. J. Gurbiel; P. E. Doan; G. T. Gassner; T. J. Macke; D. A. Case; T. Ohnishi; J. A. Fee; D. P. Ballou; B. M. Hoffman. 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.

[278] T. J. Macke. *NAB, a Language for Molecular Manipulation*. Ph.D. thesis, The Scripps Research Institute, 1996.

[279] R. E. Dickerson. Definitions and Nomenclature of Nucleic Acid Structure Parameters. *J. Biomol. Struct. Dyn.*, **1989**, *6*, 627–634.

[280] R. Tan; S. Harvey. Molecular Mechanics Model of Supercoiled DNA. *J. Mol. Biol.*, **1989**, *205*, 573–591.

[281] M. S. Babcock; E. P. D. Pednault; W. K. Olson. Nucleic Acid Structure Analysis. *J. Mol. Biol.*, **1994**, *237*, 125–156.

[282] T. F. Havel; I. D. Kuntz; G. M. Crippen. The theory and practice of distance geometry. *Bull. Math. Biol.*, **1983**, *45*, 665–720.

[283] T. F. Havel. 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.

[284] J. Kuszewski; M. Nilges; A. T. Brünger. Sampling and efficiency of metric matrix distance geometry: A novel partial metrization algorithm. *J. Biomolec. NMR*, **1992**, *2*, 33–56.

[285] B. L. deGroot; D. M. F. van Aalten; R. M. Scheek; A. Amadei; G. Vriend; H. J. C. Berendsen. Prediction of protein conformational freedom from distance constraints. *Proteins*, **1997**, *29*, 240–251.

[286] D. K. Agrafiotis. Stochastic Proximity Embedding. *J. Comput. Chem.*, **2003**, *24*, 1215–1221.

[287] W. Saenger. in *Principles of Nucleic Acid Structure*, p 120. Springer-Verlag, New York, 1984.

[288] F. Hirata, Ed. *Molecular Theory of Solvation*. Kluwer Academic Publishers, 2003.

Index