nabc/sff

1 nabc and libsff

1.1 A little history

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.[1] 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 1.6.

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*.

1.2 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.
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.

keyword	*default*	*meaning*
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.[2] 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, in ps^{-1} . Values in the range 2-5 ps^{-1} often give acceptable temperature control, while allowing transitions to take place.[3] Values near 50 ps^{-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.[4]
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.
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().

keyword	*default*	*meaning*
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 [5] 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,[6, 7] 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,[8] 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.[9]
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.
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].

keyword	*default*	*meaning*

keyword	*default*	*meaning*
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 1.6 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 1.6 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.

1.3 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'**.

1.3.1 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;
```

1.3.2 netcdfClose

```
int netcdfClose(struct AmberNetcdf NC)
```

Close NetCDF file associated with **NC**.

1.3.3 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().

1.3.4 netcdfDebug

```
int netcdfDebug(struct AmberNetcdf NC)
```

Print debug information for NetCDF file associated with **NC**.

1.3.5 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).

1.3.6 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.

1.3.7 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).

1.3.8 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.

1.3.9 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.

1.3.10 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().

1.3.11 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().

1.3.12 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.
```

1.4 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.[1] 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.[10]

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);
```

1.5 Low-MODe (LMOD) optimization methods

István Kolossváry has contributed functions, which implement the LMOD methods for minimization, conformational searching, and flexible docking.[11–14] 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.

1.5.1 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,

$$Hv = [\nabla(x_{min} + h) - \nabla(x_{min})]/h \tag{1.1}$$

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.

1.5.2 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.

1.5.3 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 1.2; the parameters in the xmin_opt structure are described in Table 1.3; these should be preceded by "`xo.`", since they are members of an *xmod_opt* struct with that name; see the sample program below to see how this works.

There are three types of minimizers that can be used, specified by the *method* parameter:

`method`

1: PRCG Polak-Ribiere conjugate gradient method, similar to the *conjgrad()* function [16].

2: L-BFGS Limited-memory Broyden-Fletcher-Goldfarb-Shanno quasi-Newton algorithm [17]. 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 [16, 18]. 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.

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 1.2: *Arguments for xmin().*

1.5.4 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.

#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;
```

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. 1.1 in section 1.5.1). 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 [15](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 1.3: *Options for xmin_opt.*

```
    //    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.

1.5.5 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.
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 1.5.2.). 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.

keyword	*default*	*meaning*
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 1.5.7, 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.
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 1.5.2.
lmod_step_size_max	5.0	The maximum length of a single LMOD ZIG move in Å. See section 1.5.2.
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 1.5.2. 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.

keyword	*default*	*meaning*
lmod_relax_grms	1.0	The gradient RMS convergence criterion of structure relaxation, see ZAG move in section 1.5.2.
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.
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 ∼50-100 Arnoldi dimension provides the fastest net computational cost with very many iterations.

1.5.6 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.

```
1  //   LMOD reverse communication external minimization package.
2  //   Written by Istvan Kolossvary.
3
4  #include "xmin_opt.h"
5  #include "lmod_opt.h"
6
7  // M A I N  P R O G R A M  to carry out LMOD simulation on a molecule/complex:
8
9  struct xmin_opt xo;
10 struct lmod_opt lo;
11
```

```
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 1.5.2). 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 1.5.2) 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.

1.5.7 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.

1.6 The Generalized Born with Hierarchical Charge Partitioning (GB-HCP)

GB-HCP (and its latest version, GB-HCPO[19]) 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[19]. 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 long range electrostatic interactions by using N log N Hierarchical Charge Partitioning (HCP) approximation [20, 21]. 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[22]: 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.

1.6.1 Level 1 HCP approximation

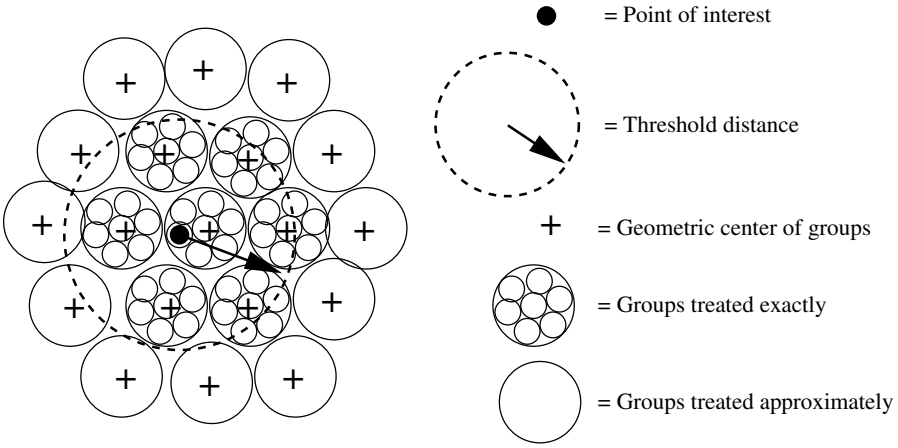
The HCP option can now be used with one level of approximation (groups) using NAB molecular dynamics scripts. No additional manipulation of the input structure files is required for one level of approximation. For an example see AmberTools/examples/hcp/2trx.nab. The level 1 approximation is recommended 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.

1.6.2 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:

Figure 1.1: *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.*



...
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

2 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.

2.1 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.

2.1.1 Conformation build-up procedures

MC-SYM[23–25] 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.*[26] 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[27, 28] 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.

2.1.2 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.[29] 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,[30] 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.*[31] 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[32] 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.

2.2 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.

2.2.1 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. 4.

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.

2.2.2 Distance geometry

nab's second initial structure-creation method is *metric matrix distance geometry,[33, 34]* 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. 5.

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[35] 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.

2.2.3 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,[36] calcyclin dimers,[37] HMG-protein/DNA complexes,[38] active sites of Rieske iron-sulfur proteins,[39] and supercoiled DNA.[40] The Examples chapter below provides a number of other sample applications.

2.3 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>
```

2.4 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.

2.5 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.

2.5.1 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.

2.5.2 Superimpose two molecules

Here is another simple nab program. It reads two DNA molecules and superimposes them using a rotation matrix made from a correspondence between their C1' atoms.

```
1  // Program 2 - Superimpose two DNA duplexes
2  molecule m, mr;
3  float r;
4
5  m = getpdb( "test.pdb" );
6  mr = getpdb( "gcg10.pdb" );
7  superimpose( m, "::C1'", mr, "::C1'" );
8  putpdb( "test.sup.pdb", m );
9  rmsd( m, "::C1'", mr, "::C1'", r );
10 printf( "rmsd = %8.3fn", r );
```

This program uses three variables—two molecules, m and mr and one float, r. An nab declaration can include any number of variables of the same type, but variables of different types must be in separate declarations. The builtin function getpdb() reads two molecules in PDB format from the files test.pdb and gcg10.pdb into the variables m and mr. The superimposition is done with the builtin function superimpose(). The arguments to superimpose() are two molecules and two "atom expressions". nab uses atom expressions as a compact way of specifying sets of atoms. Atom expressions and atom names are discussed in more detail below but for now an atom expression is a pattern that selects one or more of the atoms in a molecule. In this example, they select all atoms with names C1'.

superimpose() uses the two atom expressions to associate the corresponding C1' carbons in the two molecules. It uses these correspondences to create a rotation matrix that when applied to m will minimize the root mean square deviation between the pairs. It applies this matrix to m, "moving" it on to mr. The transformed molecule m is written out to the file test.sup.pdb in PDB format using the builtin function putpdb(). Finally the builtin function rmsd() is used to compute the actual root mean square deviation between corresponding atoms in the two superimposed molecules. It returns the result in r, which is written out using the C-like I/O function printf(). rmsd() also uses two atom expressions to select the corresponding pairs. In this example, they are the same pairs that were used in the superimposition, but any set of pairs would have been acceptable. An example of how this might be used would be to use different subsets of corresponding atoms to compute trial superimpositions and then use rmsd() over all atoms of both molecules to determine which subset did the best job.

2.5.3 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

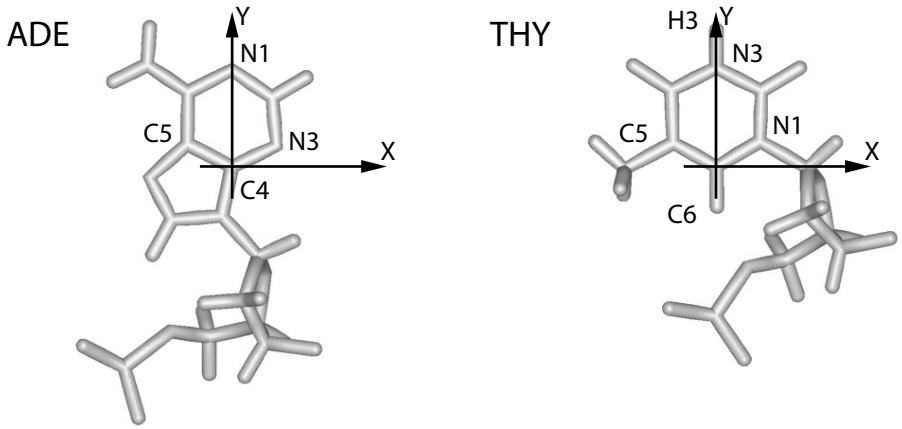


Figure 2.1: *ADE and THY after execution of Program 3.*

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 atoms as shown in Figure 2.1. 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.

2.6 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.

2.7 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.

```
...
m = newmolecule();
addstrand( m, \fIstr-1\fC );
...
for( ... ){
...
res = getresidue( \fIres-name\fC, \fIres-lib\fC );
res = transformres( mat, res, NULL );
addresidue( m, \fIstr-name\fC, res );
...
```

```
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.

2.8 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:

```
1 // getres() - map 1 letter names into 3 letter names
2 residue getres( string rname, string rlib )
3 {
4     residue res;
```

```
5       string map1to3[ hashed ];          // convert residue names
6
7       map1to3["A"] = "ADE";     map1to3["C"] = "CYT";
8       map1to3["G"] = "GUA";     map1to3["T"] = "THY";
9       map1to3["U"] = "URA";
10
11      map1to3["a"] = "ADE";     map1to3["c"] = "CYT";
12      map1to3["g"] = "GUA";     map1to3["t"] = "THY";
13      map1to3["u"] = "URA";
14
15      if( r in map1to3 ) {
16          res = getresidue( map1to3[ r ], rlib );
17      }else{
18          fprintf( stderr, "undefined residue %s\\n", r );
19          exit( 1 );
20      }
21      return( res );
22  };
```

getres() is the first of several nab functions that are discussed in this User Manual. The following explanation will cover not just getres() but will serve as an introduction to user defined nab functions in general.

An nab function is a named group of declarations and statements that is executed as a unit by using the function's name in an expression. nab functions can have special variables called parameters that allow the same function to operate on different data. A function definition begins with a header that describes the function, followed by the function body which is a list of statements and declarations enclosed in braces ({}) and ends with a semicolon. The header to getres() is on line 2 and the body is on lines 3 to 22.

Every nab function header begins with the reserved word that specifies its type, followed by the function's name followed by its parameters (if any) enclosed in parentheses. The parentheses are always required, even if the function does not have parameters. nab functions may return a single value of any of the 10 nab types. nab functions can not return arrays. In symbolic terms every nab function header uses this template:

***type name*(*parameters?*)**

The parameters (if present) to an nab function are a comma separated list of type variable pairs:

***type1 variable1*, *type2 variable2*, ...**

An nab function may have any number of parameters, including none. Parameters may of any of the 10 nab types, but unlike function values, parameters can be arrays, including *hashed* arrays. The function getres() has two parameters, the two string variables resname and reslib.

Parameters to nab functions are "called by reference" which means that they contain the actual data—not copies of it—that the function was called with. When an nab function parameter is assigned, the actual data in the calling function is changed. The only exception is when an expression is passed as a parameter to an nab function. In this case, the nab compiler evaluates the expression into a temporary (and invisible to the nab programmer) variable and then operates on its contents.

Immediately following the function header is the function body. It is a list of declarations followed by a list of statements enclosed in braces. The list of declarations, the list of statements or both may be empty. getres() has several statements, and a single declaration, the variable res. This variable is a *local variables*. Local variables are defined only when the function is active. If a local variable has the same name as variable defined outside of a it the local variable hides the global one. Local variables can not be parameters.

The statement part of getres() begins on line 6. It consists of several if statements organized into a decision tree. The action of this tree is to translate one of the strings A, , , T, etc., or their lower case equivalents into the corresponding three letter standard nucleic acid residue name and then extract that residue from reslib using the low level residue library function getresidue(). The value returned by getresidue() is stored in the local variable res, except when the input string is not one of those listed above. In that case, getres() writes a message to stderr indicating that it can not translate the input string and sets res to the value NULL. nab uses NULL to represent

non-existent values of the types string, file, atom, residue, molecule and bounds. A value of NULL generally means that a variable is uninitialized or that an error occurred in creating it.

A function returns a value by executing a return statement, which is the reserved word return followed by an expression. The return statement evaluates the expression, sets the function value to it and returns control to the point just after the call. The expression is optional but if present the type of the expression must be the same as the type of the function or both must be numeric (int, float). If the expression is missing, the function still returns, but its value is undefined. getres() includes one return statements on line 20. A function also returns with an undefined value when it "runs off the bottom", i.e., executes the last statement before the closing brace and that statement is not a return.

2.9 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.

2.10 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().

2.11 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.

2.11.1 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.

2.11.2 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.

2.11.3 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.

2.12 Creating Watson Crick duplexes

Watson/Crick duplexes are fundamental components of almost all nucleic acid structures and nab provides several functions for use in creating them. They are

```
residue getres( string resname, string reslib );
molecule bdna( string seq );
molecule fd_helix( string helix_type, string seq, string acid_type );
string wc_complement( string seq, string reslib, string natype );
molecule wc_basepair( residue sres, residue ares );
```

```
molecule wc_helix( string seq, string rlib, string natype,
          string aseq, string arlib, string anatype, float xoff,
          float incl, float twist, float rise, string opts );
```

All of these functions are written in nab allowing the user to modify or extend them as needed without having to modify the nab compiler.

Note: If you just want to create a regular helical structure with a given sequence, use the "fiber-diffraction" routine fd_helix(), which is discussed in Section 3.13. The methods discussed next are more general, and can be extended to more complicated problems, but they are also much harder to follow and understand. The construction of "unusual" nucleic acids was the original focus of NAB; if you are using NAB for some other purpose (such as running Amber force field calculations) you should probably skip to Chapter **??** at this point.

2.12.1 bdna() and fd_helix()

The function bdna() which was used in the first example converts a string into a Watson/Crick DNA duplex using average DNA helical parameters.

```
1  // bdna() - create average B-form duplex
2  molecule bdna( string seq )
3  {
4          molecule m;
5          string cseq;
6          cseq = wc_complement( seq, "", "dna" );
7          m = wc_helix( seq, "", "dna",
8                    cseq, "", "dna",
9                    2.25, -4.96, 36.0, 3.38, "s5a5s3a3" );
10          return( m );
11 };
```

bdna() calls wc_helix() to create the molecule. However, wc_helix() requires both strands of the duplex so bdna() calls wc_complement() to create a string that represents the Watson/Crick complement of the sequence contained in its parameter seq. The string *"s5a5s3a3"* replaces both the *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 3.13.

2.12.2 wc_complement()

The function wc_complement() takes three strings. The first is a sequence using the standard one letter code, the second is the name of an nab residue library, and the third is the nucleic acid type (RNA or DNA). It returns a string that contains the Watson/Crick complement of the input sequence in the same one letter code. The input string and the returned complement string have opposite directions. If the left end of the input string is the 5' base then the left end of the returned string will be the 3' base. The actual direction of the two strings depends on their use.

```
1  // wc_complement() - create a string that is the W/C
2  // complement of the string seq
3  string wc_complement( string seq, string rlib, string rlt )
4  //  (note that rlib is unused: included only for backwards compatibility
```

```
5  {
6     string acbase, base, wcbase, wcseq;
7     int i, len;
8
9     if( rlt == "dna" )      acbase = "t";
10    else if( rlt == "rna" ) acbase = "u";
11    else{
12       fprintf( stderr,
13         "wc_complement: rlt (%s) is not dna/rna, no W/C comp.", rlt );
14       return( NULL );
15    }
16    len = length( seq );
17    wcseq = NULL;
18    for( i = 1; i <= len; i = i + 1 ){
19        base = substr( seq, i, 1 );
20        if( base == "a" || base == "A" )       wcbase = acbase;
21        else if( base == "c" || base == "C" )  wcbase = "g";
22        else if( base == "g" || base == "G" )  wcbase = "c";
23        else if( base == "t" || base == "T" )  wcbase = "a";
24        else if( base == "u" || base == "U" )  wcbase = "a";
25        else{
26            fprintf( stderr, "wc_complement: unknown base %sn", base );
27            return( NULL );
28        }
29        wcseq = wcseq + wcbase;
30    }
31    return( wcseq );
32 }
```

wc_complement() begins its work in line 9, where the nucleic acid type, as indicated by rlt as DNA or RNA is used to determine the correct complement for an a. The complementary sequence is created in the for loop that begins in line 18 and extends to line 30. The nab builtin substr() is used to extract single characters from the input sequence beginning with with position 1 and working from left to right until entire input sequence has been converted. The if-tree from lines 20 to 28 is used to set the character complementary to the current character, using the previously determined acbase if the input character is an a or A. Any character other than the expected a, c, g, t, u (or A, C, G, T, U) is an error causing wc_complement() to print an error message and return NULL, indicating that it failed. Line 29 shows how nab uses the infix + to concatenate character strings. When the entire string has been complemented, the for loop terminates and the complementary sequence now in wcseq is returned as the function value. Note that if the input sequence is empty, wc_complement() returns NULL, indicating failure.

2.12.3 wc_helix() Overview

wc_helix() generates a uniform helical duplex from a sequence, its complement, two residue libraries and four helical parameters: x-offset, inclination, twist and rise. By using two residue libraries, wc_helix() can generate RNA/DNA heteroduplexes. wc_helix() returns an nab molecule containing two strands. The string seq becomes the "sense" strand and the string aseq becomes the "anti" strand. seq and aseq are required to be complementary although this is not checked. wc_helix() creates the molecule one base pair at a time. seq is read from left to right, aseq is read from right to left and corresponding letters are extracted and converted to residues by getres(). These residues are in turn combined into an idealized Watson/Crick base pair by wc_basepair(). An AT created by wc_basepair() is shown in Figure 2.

A Watson/Crick duplex can be modeled as a set of planes stacked in a helix. The numbers that describe the relationships between the planes and between the planes and the helical axis are called helical parameters. Planes can be defined for each base or base pair. Six numbers (three displacements and three angles) can be defined for every pair of planes; however, helical parameters for nucleic acid bases are restricted to the six numbers describing the the relationship between the two bases in a base pair and the six numbers describing the relationship between

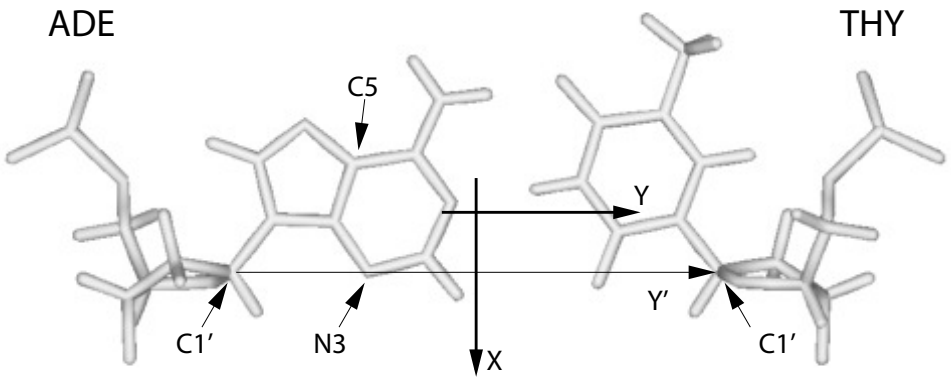


Figure 2.2: *ADE.THY from wc_basepair().*

adjacent base pairs. A complete description of helical parameters can be found in Dickerson.[41]

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.

2.12.4 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 2.2 is in the plane of page with origin on the C4 of the A. The calls to setframe() and alignframe() move the base pair so that the origin is at the intersection of the lines marked X and Y'.

```
1 // wc_basepair() - create Watson/Crick base pair
2 #define AT_SEP 8.29
3 #define CG_SEP 8.27
```

```
4
5  molecule wc_basepair( residue sres, residue ares )
6  {
7          molecule m, m_sense, m_anti;
8          float sep;
9          string srname, arname;
10         string xtail, xhead;
11         string ytail, yhead;
12         matrix mat;
13
14         m = newmolecule();
15         m_sense = newmolecule();
16         m_anti = newmolecule();
17         addstrand( m, "sense" );
18         addstrand( m, "anti" );
19         addstrand( m_sense, "sense" );
20         addstrand( m_anti, "anti" );
21
22         srname = getresname( sres );
23         arname = getresname( ares );
24         ytail = "sense::C1'";
25         yhead = "anti::C1'";
26         if( ( srname == "ADE" ) || ( srname == "DA" ) ||
27             ( srname == "RA" ) || ( srname =~ "[DR]A[35]" ) ){
28                 sep = AT_SEP;
29                 xtail = "sense::C5";
30                 xhead = "sense::N3";
31                 setframe( 2, m_sense,
32                         "::C4", "::C5", "::N3", "::C4", "::N1" );
33         }else if( ( srname == "CYT" ) || ( srname =~ "[DR]C[35]*" ) ){
34                 sep = CG_SEP;
35                 xtail = "sense::C6";
36                 xhead = "sense::N1";
37                 setframe( 2, m_sense,
38                         "::C6", "::C5", "::N1", "::C6", "::N3" );
39         }else if( ( srname == "GUA" ) || ( srname =~ "[DR]G[35]*" ) ){
40                 sep = CG_SEP;
41                 xtail = "sense::C5";
42                 xhead = "sense::N3";
43                 setframe( 2, m_sense,
44                         "::C4", "::C5", "::N3", "::C4", "::N1" );
45         }else if( ( srname == "THY" ) || ( srname =~ "DT[35]*" ) ){
46                 sep = AT_SEP;
47                 xtail = "sense::C6";
48                 xhead = "sense::N1";
49                 setframe( 2, m_sense,
50                         "::C6", "::C5", "::N1", "::C6", "::N3" );
51         }else if( ( srname == "URA" ) || ( srname =~ "RU[35]*" ) ){
52                 sep = AT_SEP;
53                 xtail = "sense::C6";
54                 xhead = "sense::N1";
55                 setframe( 2, m_sense,
56                         "::C6", "::C5", "::N1", "::C6", "::N3" );
57         }else{
58                 fprintf( stderr,
59                         "wc_basepair : unknown sres %s\\n",srname );
60                 exit( 1 );
```

```
61          }
62          addresidue( m_sense, "sense", sres );
63          if( ( arname == "ADE" ) || ( arname == "DA" ) ||
64              ( arname == "RA" ) || ( arname =~ "[DR]A[35]" ) ){
65                  setframe( 2, m_anti,
66                          "::C4", "::C5", "::N3", "::C4", "::N1" );
67          }else if( ( arname == "CYT" ) || ( arname =~ "[DR]C[35]*" ) ){
68                  setframe( 2, m_anti,
69                          "::C6", "::C5", "::N1", "::C6", "::N3" );
70          }else if( ( arname == "GUA" ) || ( arname =~ "[DR]G[35]*" ) ){
71                  setframe( 2, m_anti,
72                          "::C4", "::C5", "::N3", "::C4", "::N1" );
73          }else if( ( arname == "THY" ) || ( arname =~ "DT[35]*" ) ){
74                  setframe( 2, m_anti,
75                          "::C6", "::C5", "::N1", "::C6", "::N3" );
76          }else if( ( arname == "URA" ) || ( arname =~ "RU[35]*" ) ){
77                  setframe( 2, m_anti,
78                          "::C6", "::C5", "::N1", "::C6", "::N3" );
79          }else{
80                  fprintf( stderr,
81                          "wc_basepair : unknown ares %s\\n",arname );
82                  exit( 1 );
83          }
84          addresidue( m_anti, "anti", ares );
85
86          alignframe( m_sense, NULL );
87          alignframe( m_anti, NULL );
88          mat = newtransform( 0., 0., 0., 180., 0., 0. );
89          transformmol( mat, m_anti, NULL );
90          mat = newtransform( 0., sep, 0., 0., 0., 0. );
91          transformmol( mat, m_anti, NULL );
92          mergestr( m, "sense", "last", m_sense, "sense", "first" );
93          mergestr( m, "anti", "last", m_anti, "anti", "first" );
94
95          freemolecule( m_sense ); freemolecule( m_anti );
96
97          setframe( 2, m, "::C1'", xtail, xhead, ytail, yhead );
98          alignframe( m, NULL );
99          return( m );
100 };
```

2.12.5 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";
22        else sreslib_use = sreslib;
23 if( areslib == "" ) areslib_use = "all_nucleic94.lib";
24        else areslib_use = areslib;
25
26 if( seq == NULL && aseq == NULL ){
27        fprintf( stderr, "wc_helix: no sequence\\n" );
28        return( NULL );
29 }else if( seq == NULL ){
30        seq = wc_complement( aseq, areslib_use, snatype );
```

```
       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 );
}

```

```
88
89  i = slen;          // add in final residue pair:
90
91  if( i > 1 ){
92         srname = substr( seq, i, 1 );
93         srname = "D" + loup[ substr( seq, i, 1 ) ];
94         setreslibkind( sreslib, snatype );
95         if( opts =~ "s3"  )
96            sres = getres( srname + "3", sreslib_use );
97         else
98            sres = getres( srname, sreslib_use );
99         arname = "D" + loup[ substr( aseq, i, 1 ) ];
100        setreslibkind( areslib, anatype );
101        if( opts =~ "a5" )
102           ares = getres( arname + "5", areslib_use );
103        else
104           ares = getres( arname, areslib_use );
105
106        m2 = wc_basepair( sres, ares );
107        freeresidue( sres ); freeresidue( ares );
108        transformmol( xomat, m2, NULL );
109        transformmol( inmat, m2, NULL );
110        mat = newtransform( 0., 0., trise, 0., 0., ttwist );
111        transformmol( mat, m2, NULL );
112        mergestr( m1, "sense", "last", m2, "sense", "first" );
113        connectres( m1, "sense", i-1, "O3'", i, "P" );
114        mergestr( m1, "anti", "first", m2, "anti", "last" );
115        connectres( m1, "anti", 1, "O3'", 2, "P" );
116        trise = trise + rise;
117        ttwist = ttwist + twist;
118        freemolecule( m2 );
119 }
120
121 m3 = newmolecule();
122 addstrand( m3, "sense" );
123 addstrand( m3, "anti" );
124 if( has_s )
125        mergestr( m3, "sense", "last", m1, "sense", "first" );
126 if( has_a )
127        mergestr( m3, "anti", "last",  m1, "anti", "first" );
128 freemolecule( m1 );
129
130 return( m3 );
131 };
```

3 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.

3.1 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.

3.1.1 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

3.1.2 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

3.1.3 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.

3.1.4 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

3.1.5 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.

3.2 Higher-level constructs

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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.

3.2.7 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.

3.3 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.

3.3.1 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.
```

3.3.2 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.

3.3.3 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;
```

3.3.4 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;
}
```

3.3.5 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.

3.3.6 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
      ...
}
```

3.3.7 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
    ...
}
```

3.3.8 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.
```

3.3.9 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.

3.4 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 { ... };
```

3.5 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.

3.5.1 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
};
```

3.5.2 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;
```

3.6 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.

3.7 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".

3.8 Math Functions

nab provides the builtin mathematical functions shown in Table 3.1. 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.

3.9 System Functions

```
int exit( int i );
int system( string cmd );
```

The function exit() terminates the calling nab program with return status i. system() invokes a subshell to execute cmd. The subshell is always /bin/sh. The return value of system() is the return value of the subshell and not the command it executed.

3.10 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.

3.10.1 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.

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 $\lvert x \rvert$.
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 3.1: *NAB built-in mathematical functions*

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.

3.10.2 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.

3.11 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

3.12 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 **??**.

3.13 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.

3.14 Reduced Representation DNA Modeling Functions

nab provides several functions for creating the reduced representation models of DNA described by R. Tan and S. Harvey.[42] 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.

3.15 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 3.2.

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
```

3.16 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 );
int sugarpuckeranal( molecule mol, int strandnum, int startres, int endres );
```

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 3.2: *Options for getpdb and putpdb.*

```
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..[43]*

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.

3.17 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.

3.18 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.

3.19 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.

4 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.

4.1 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.

4.2 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).

4.3 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.

4.4 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.

4.4.1 Matrix Creation Functions

```
int MAT_cube( point pts[3], matrix mats[24] )
int MAT_ico( point pts[3], matrix mats[60] )
int MAT_octa( point pts[3], matrix mats[24] )
int MAT_tetra( point pts[3], matrix mats[12] )
int MAT_dihedral( point pts[3], int nfold, matrix mats[1] )
int MAT_cyclic( point pts[2], float ang, int cnt, matrix mats[1] )
int MAT_helix( point pts[2], float ang, float dst, int cnt, matrix mats[1] )
int MAT_orient( point pts[4], float angs[3], matrix mats[1] )
int MAT_rotate( point pts[2], float ang, matrix mats[1] )
int MAT_translate( point pts[2], float dst, matrix mats[1] )
```

These two groups of functions produce arrays of matrices that can be applied to objects to generate point group symmetries (first group) or useful transformations (second group). The operations are defined with respect to a center and a set of axes specified by the points in the array pts[]. Every function requires a center and one axis which are pts[1] and the vector pts[1]→pts[2]. The other two points (if required) define two additional directions: pts[1]→pts[3] and pts[1]→pts[4]. How these directions are used depends on the function.

The point groups generated by the functions MAT_cube(), MAT_ico(), MAT_octa() and MAT_tetra() have three internal 2-fold axes. While these 2-fold are orthogonal, the 2 directions specified by the three points in pts[] need only be independent (not parallel). The 2-fold axes are constructed in this fashion. Axis-1 is along the direction pts[1]→pts[2]. Axis-3 is along the vector pts[1]→pts[2] $\times$ pts[1]→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]→pts[1] and the second axis is created by the same orthogonalization process described above. Unlike the previous point group functions the number of matrices created by MAT_dihedral() is not fixed but is equal to $2 \times nfold$.

MAT_cyclic() creates cnt matrices that produce uniform rotations about the axis pts[1]→pts[2]. The rotations are in multiples of the angle ang beginning with 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]→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]→pts[2]. A zero ang or dst is acceptable in which case the function creates an identity matrix. Except for a different user interface these two functions are equivalent to the nab builtins rot4p() and tran4p().

MAT_orient() creates a matrix that rotates a object about the three axes pts[1]→pts[2], pts[1]→pts[3] and pts[1]→pts[4]. The rotations are specified by the values of the array angs[], with ang[1] the rotation about axis-1 etc. The rotations are applied in the order axis-3, axis-2, axis-1. The axes remained fixed throughout the operation and zero angle values are acceptable. If all three angles are zero, MAT_orient() creates an identity matrix.

4.4.2 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.

4.5 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.

4.5.1 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:	$IM_1 \times R_1$, $IM_2 \times R_1$, $IM_3 \times R_1$, $IM_1 \times R_2$, $IM_2 \times R_2$, $IM_3 \times R_2$

4.5.2 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	-

4.5.3 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.

4.5.4 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.

4.5.5 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.

4.5.6 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.

5 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.

5.1 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.[33, 44] 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:

$$g_{ij} \equiv \mathbf{x}_i \cdot \mathbf{x}_j \tag{5.1}$$

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

$$g_{ij} = 2(d_{i0}^2 + d_{j0}^2 - d_{ij}^2) \tag{5.2}$$

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:

$$g_{ij} = \sum_{k=1}^{3} x_{ik} x_{jk} = \sum_{k=1}^{3} w_{ik} w_{jk} \lambda_k \tag{5.3}$$

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:

$$x_{ik} = \lambda_k^{1/2} w_{ik} \tag{5.4}$$

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.[45, 46]

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.[35] 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.[47] 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,[48] 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.

5.2 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

```
-name=value
-name to select the default value if it exists.
```

The options to newbounds() are listed in Table 5.1.

The next five routines use atom expressions aex1 and aex2 to select two sets of atoms. Each of these four routines returns the number of bounds set or changed. For each pair of atoms ($a1$ in aex1 and $a2$ in aex2)

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 5.1: *Options to newbounds.*

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(^3). In this case, a more efficient triangle smoothing routine, geodesics() is used. geodesics() smoothes the bounds matrix via the triangle inequality using a sparse matrix version of a shortest path algorithm.

The embed routine takes a bounds object as input, and returns a four-dimensional array of coordinates; (values of the 4-th coordinate may be nearly zero, depending on the value of *k*4*d*, 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.
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.

keyword	*default*	*meaning*
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:

```
1  molecule m;
2  bounds b;
3  float fret, xyz[ 10000 ];
4  int ier;
5
6  m = getpdb( argv[2] );
7  b = newbounds( m, "" );
8  tsmooth( b, 0.0005 );
9
10 dg_options( b, "gdist=1, ntpr=50, k4d=2.0, randpair=10." );
11 embed( b, xyz );
12 ier = conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 200 );
13 printf( "conjgrad returns %d\\n", ier );
14
15 setmol_from_xyzw( m, NULL, xyz );
16 putpdb( "new.pdb", m );
```

In lines 6-8, the molecule is created by reading in a pdb file, then bounds are created and smoothed for it. The embed options (established in line 10) include 10% random pairwise metrization, use of Gaussian distance selection, squeezing out the 4-th dimension with a force constant of 2.0, and printing every 50 steps. The coordinates developed in the *embed* step (line 11) are passed to a conjugate gradient minimizer (see the description below), which will minimize for 200 steps, using the bounds-violation routine *db_viol* as the target function. Finally, in lines 15-16, the *setmol_from_xyzw* routine is used to put the coordinates from the *xyz* array back into the molecule, and a new pdb file is written.

More complex and representative examples of distance geometry are given in the **Examples** chapter below.

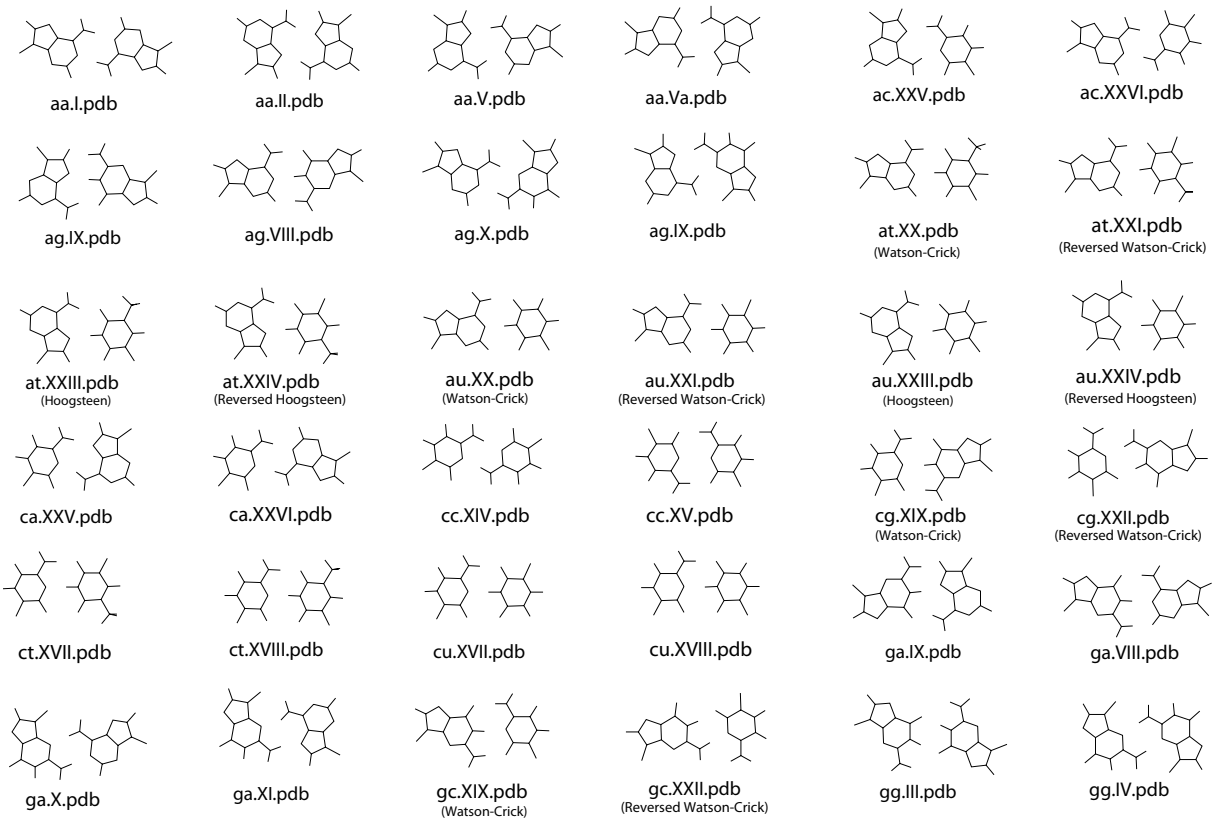


Figure 5.1: *Basepair templates for use with useboundsfrom(), (aa-gg)*

5.3 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.[49] 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 5.1 and 5.2.

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.

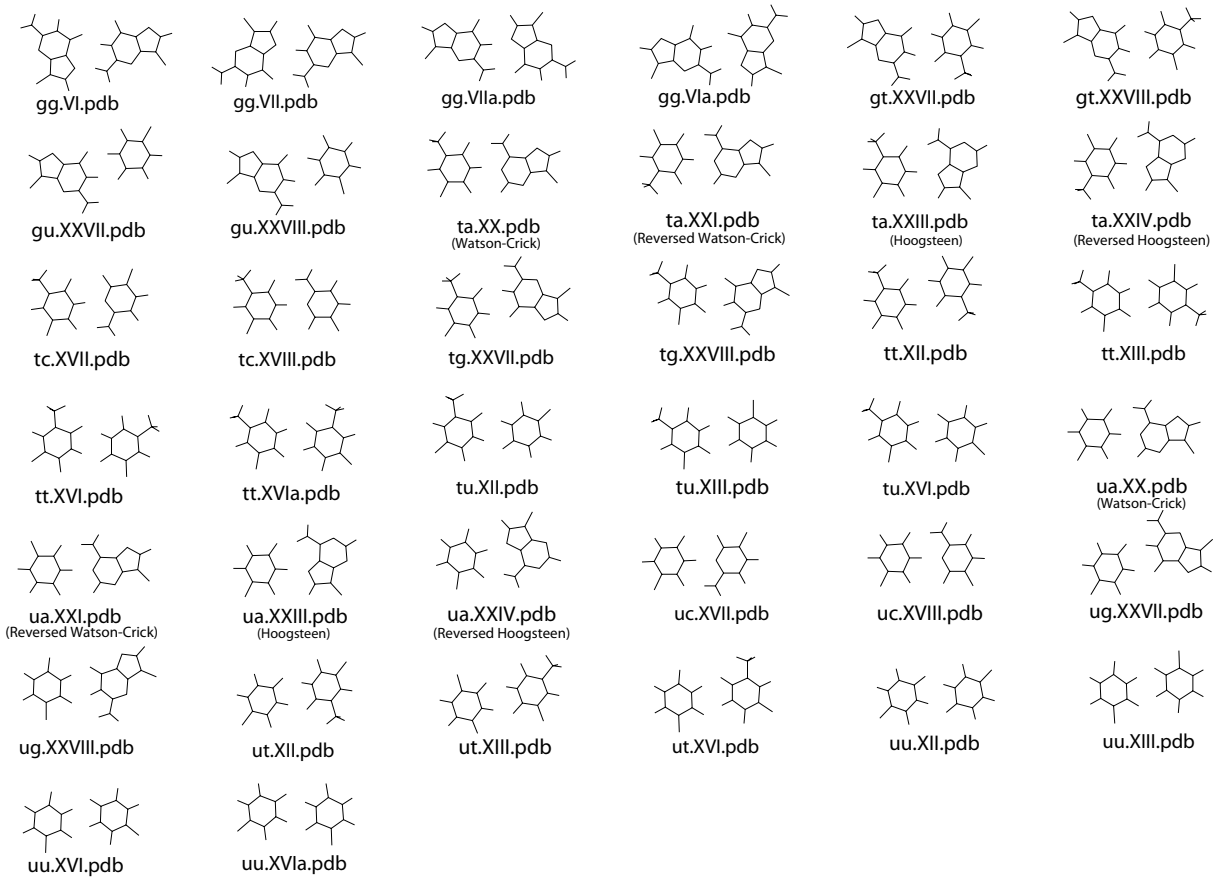


Figure 5.2: *Basepair templates for use with useboundsfrom(), (gg-uu)*

5.4 Bounds databases

In addition to canonical templates, it is also possible to specify bounds information from a database of known molecular structures. This provides the option to use data obtained from actual structures, rather than from an idealized, canonical conformation.

The function setboundsfromdb() sets the bounds of all pairs of atoms between the two residues selected by aex1 and aex2 to a statistically averaged distance calculated from known structures plus or minus a multiple of the standard deviation. The statistical information is kept in database files. Currently, there are three types of database files - Those containing bounds information between Watson-Crick basepairs, those containing bounds information between helically stacked residues, and those containing intra-residue bounds information for residues in any conformation. The standard deviation is multiplied by the parameter *mul* and subtracted from the average distance to determine the lower bound and similarly added to the average distance to determine the upper bound of all base-base atom distances. Base-backbone bounds, that is, bounds between pairs of atoms in which one atom is a base atom and the other atom is a backbone atom, are set to be looser than base-base atoms. Specifically, the lower bound between a base-backbone atom pair is set to the smallest measured distance of all the structures considered in creating the database. Similarly, the upper bound between a base-backbone atom pair is set to the largest measured distance of all the structures considered. Base-base, and base-sugar bounds are set in a similar manner. This was done to avoid imposing false constraints on the atomic bounds, since Watson-Crick basepairing and stacking does not preclude any specific backbone and sugar conformation. setboundsfromdb() first searches the current directory for *dbase* before checking the default database location, $AMBERHOME/dat/dgdb.

Each entry in the database file has six fields: The atoms whose bounds are to be set, the number of separate structures sampled in constructing these statistics, the average distance between the two atoms, the standard deviation, the minimum measured distance, and the maximum measured distance. For example, the database bdna.basepair.db has the following sample entries:

```
A:C2-T:C1' 424 6.167 0.198 5.687 6.673
A:C2-T:C2 424 3.986 0.175 3.554 4.505
A:C2-T:C2' 424 7.255 0.304 5.967 7.944
A:C2-T:C3' 424 8.349 0.216 7.456 8.897
A:C2-T:C4 424 4.680 0.182 4.122 5.138
A:C2-T:C4' 424 8.222 0.248 7.493 8.800
A:C2-T:C5 424 5.924 0.168 5.414 6.413
A:C2-T:C5' 424 9.385 0.306 8.273 10.104
A:C2-T:C6 424 6.161 0.163 5.689 6.679
A:C2-T:C7 424 7.205 0.184 6.547 7.658
```

The first column identifies the atoms from the adenosine C2 atom to various thymidine atoms in a Watson-Crick basepair. The second column indicates that 424 structures were sampled in determining the next four columns: the average distance, the standard deviation, and the minimum and maximum distances.

The databases were 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.

5.5 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.

6 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.

6.1 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.

6.2 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.

6.2.1 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, "" );
13
14 b = newbounds(m, "");
15 allocate xyz[ 4*m.natoms ];
16
17 useboundsfrom(b, m, "::??,H?[^T']", m, "::??,H?[^T']", 0.0 );
18 for ( i = 1; i < m.nresidues/2 ; i = i + 1 ){
19        setbounds(b,m, sprintf("1:%d:O3'",i),
20                       sprintf("1:%d:P",i+1),   1.595,1.595);
21        setbounds(b,m, sprintf("1:%d:O3'",i),
22                       sprintf("1:%d:O5'",i+1), 2.469,2.469);
```

```
        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 );
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.

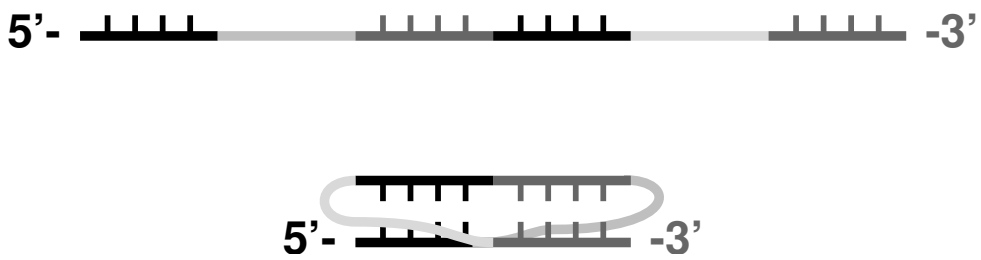


Figure 6.1: *Single-stranded RNA* (top) *folded into a pseudoknot* (bottom). *The black and dark grey base pairs can be stacked.*

6.2.2 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 6.1. 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, "");

for ( i = 1; i <= seqlen; i = i + 1) {
        useboundsfrom(b, m, sprintf("1:%d:??,H?[^'T]", i), m,
        sprintf("1:%d:??,H?[^'T]", i), 0.0 );
}

setboundsfromdb(b, m, "1:1:",  "1:2:",  "arna.stack.db",    1.0);
setboundsfromdb(b, m, "1:2:",  "1:3:",  "arna.stack.db",    1.0);
setboundsfromdb(b, m, "1:3:",  "1:18:", "arna.stack.db",    1.0);
setboundsfromdb(b, m, "1:18:", "1:19:", "arna.stack.db",    1.0);
setboundsfromdb(b, m, "1:19:", "1:20:", "arna.stack.db",    1.0);

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.

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 6.2. 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.

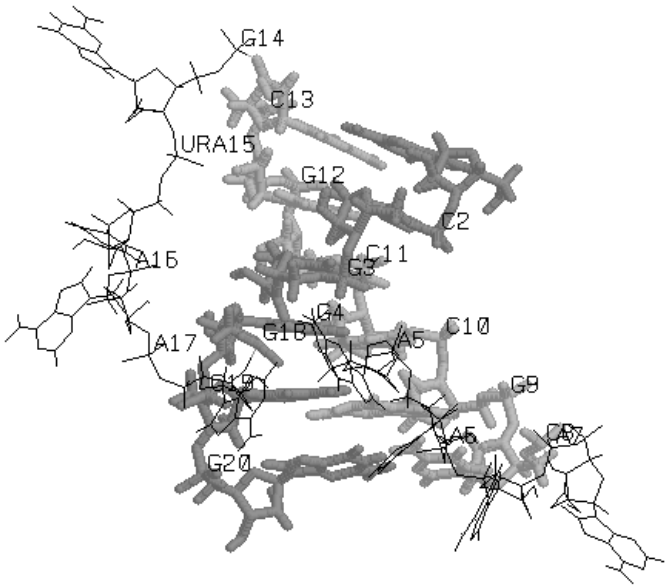


Figure 6.2: *Folded RNA pseudoknot.*

6.2.3 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.

```
1  //  Program 8a. General driver routine to do distance geometry \fC
2  //    on proteins, with DYANA-like distance restraints.\fC
3
4  #define MAXCOORDS 12000
5
6  molecule m;
7  atom    a;
8  bounds    b;
9  int    ier,i, numstrand, ires,jres;
10 float    fret, rms, ub;
11 float    xyz[ MAXCOORDS ], f[ MAXCOORDS ], v[ MAXCOORDS ];
12 file    boundsf;
13 string  iresname,jresname,iat,jat,aex1,aex2,aex3,aex4,line,dgopts,seq;
14
15 //  sequence of the mrf2 protein:
16 seq = "RADEQAFLVALYKYMKERKTPIERIPYLGFKQINLWTMFQAAQKLGGYETITARRQWKHIY"
17     + "DELGGNPGSTSAATCTRRHYERLILPYERFIKGEEDKPLPPIKPRK";
```

```
18
19  //  build this sequence in an extended conformation, and construct a bounds
20  //     matrix just based on the covalent structure:
21  m = linkprot( "A", seq, "" );
22  b = newbounds( m, "" );
23
24  //  read in constraints, updating the bounds matrix using "andbounds":
25
26  //  distance constraints are basically those from Y.-C. Chen, R.H.  Whitson
27  //    Q. Liu, K. Itakura and Y. Chen, "A novel DNA-binding motif shares
28  //    structural homology to DNA replication and repair nucleases and
29  //    polymerases,"  Nature Sturct. Biol. 5:959-964 (1998).
30
31  boundsf = fopen( "mrf2.7col", "r" );
32  while( line = getline( boundsf ) ){
33      sscanf( line, "%d %s %s %d %s %s %lf", ires, iresname, iat,
34          jres, jresname, jat, ub );
35
36  //   translations for DYANA-style pseudoatoms:
37      if( iat == "HN" ){ iat = "H"; }
38      if( jat == "HN" ){ jat = "H"; }
39
40      if( iat == "QA" ){ iat = "CA"; ub += 1.0; }
41      if( jat == "QA" ){ jat = "CA"; ub += 1.0; }
42      if( iat == "QB" ){ iat = "CB"; ub += 1.0; }
43      if( jat == "QB" ){ jat = "CB"; ub += 1.0; }
44      if( iat == "QG" ){ iat = "CG"; ub += 1.0; }
45      if( jat == "QG" ){ jat = "CG"; ub += 1.0; }
46      if( iat == "QD" ){ iat = "CD"; ub += 1.0; }
47      if( jat == "QD" ){ jat = "CD"; ub += 1.0; }
48      if( iat == "QE" ){ iat = "CE"; ub += 1.0; }
49      if( jat == "QE" ){ jat = "CE"; ub += 1.0; }
50      if( iat == "QQG" ){ iat = "CB"; ub += 1.8; }
51      if( jat == "QQG" ){ jat = "CB"; ub += 1.8; }
52      if( iat == "QQD" ){ iat = "CG"; ub += 1.8; }
53      if( jat == "QQD" ){ jat = "CG"; ub += 1.8; }
54      if( iat == "QG1" ){ iat = "CG1"; ub += 1.0; }
55      if( jat == "QG1" ){ jat = "CG1"; ub += 1.0; }
56      if( iat == "QG2" ){ iat = "CG2"; ub += 1.0; }
57      if( jat == "QG2" ){ jat = "CG2"; ub += 1.0; }
58      if( iat == "QD1" ){ iat = "CD1"; ub += 1.0; }
59      if( jat == "QD1" ){ jat = "CD1"; ub += 1.0; }
60      if( iat == "QD2" ){ iat = "ND2"; ub += 1.0; }
61      if( jat == "QD2" ){ jat = "ND2"; ub += 1.0; }
62      if( iat == "QE2" ){ iat = "NE2"; ub += 1.0; }
63      if( jat == "QE2" ){ jat = "NE2"; ub += 1.0; }
64
65      aex1 = ":" + sprintf( "%d", ires) + ":" + iat;
66      aex2 = ":" + sprintf( "%d", jres) + ":" + jat;
67      andbounds( b, m, aex1, aex2, 0.0, ub );
68  }
69  fclose( boundsf );
70
71  //  add in helical chirality constraints to force right-handed helices:
72  //    (hardwire in locations 1-16, 36-43, 88-92)
73  for( i=1; i<=12; i++){
74      aex1 = ":" + sprintf( "%d", i ) + ":CA";
```

```
    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
//  itself, and print out the violations:
setmol_from_xyzw( m, NULL, xyz );
dumpboundsviolations( stdout, b, 0.5 );

//  do a final short molecular-mechanics "clean-up":
putpdb( m, "temp.pdb" );
m = getpdb_prm( "temp.pdb", "leaprc.protein.ff14SB", "", 0 );
setxyz_from_mol( m, NULL, xyz );

mm_options( "cut=10.0" );
mme_init( m, NULL, "::ZZZ", xyz, NULL );
conjgrad( xyz, 3*m.natoms, fret, mme, 0.02, 100., 200 );
setmol_from_xyz( m, NULL, xyz );
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.

6.3 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.

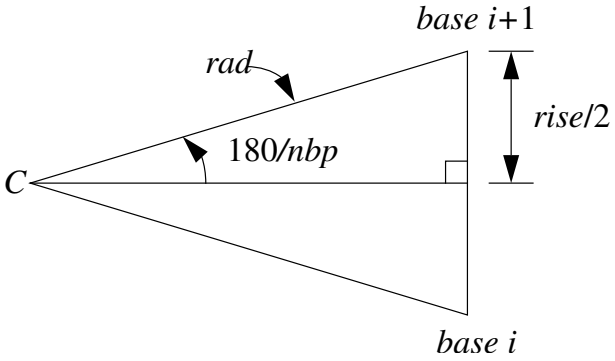
However, this method is inadequate for the more complicated curves that can be handled by example 3. Here each base is placed on the curve so that its helical axis is aligned correctly, but its helical orientation with respect to the previous base is arbitrary. It is then rotated about its helical axis so that it has the correct twist with respect to the previous base.

6.3.1 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.

```
1  // Program 9 - Create closed circular DNA.
2  #define RISE    3.38
3
4  int     b, nbp, dlk;
5  float       rad, twist, ttw;
6  molecule    m, m1;
7  matrix      matdx, mattw, matry;
8  string      sbase, abase;
9  int     getbase();
10
11 if( argc != 3 ){
12     fprintf( stderr, "usage: %s nbp dlk\\n", argv[ 1 ] );
```

```
    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 );

    if( b > 1 ){
        matry = newtransform(
            0.,0.,0.,0.,-360.*(b-1)/nbp,0. );
        transformmol( matry, 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 = ttw + twist;
    if( ttw >= 360.0 )
        ttw = ttw - 360.0;
}

connectres( m, "A", nbp, "O3'", 1, "P" );
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 theΔ*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 Δ*linkingnumber*(dlk) is set in line 24. The helical twist and circle radius are computed in lines 26 and 27 in accordance with the formulas developed above. Line 29 creates a transformation matrix, matdx, that is used to move each base from the global origin along the X-axis to the point where its center intersects the circle.

The circular DNA is built in the molecule variable m, which is initialized and given two strands, "A" and "B" in lines 30-32. The variable ttw in line 34 holds the total twist applied to each base pair The molecule is created in the loop from lines 35-66. The base pair number (b) is converted to the appropriate strings specifying the two nucleotides in this pair. This is done by the function getbase(). This source of this function must be provided by the user who is creating the circles as only he or she will know the actual DNA sequence of the circle. Once the two bases are specified they are passed to the nab builtin wc_helix() which returns a single base pair in the XY plane with its center at the origin. The helical axis of this base pair is on the Z-axis with the 5'-3' direction oriented in the positive Z-direction.

One or three transformations is required to position this base in its correct place in the circle. It must be rotated about the Z-axis (its helical axis) so that it is one additional unit of twist beyond the previous base. This twist is done in lines 43-46. Since the first base needs 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.

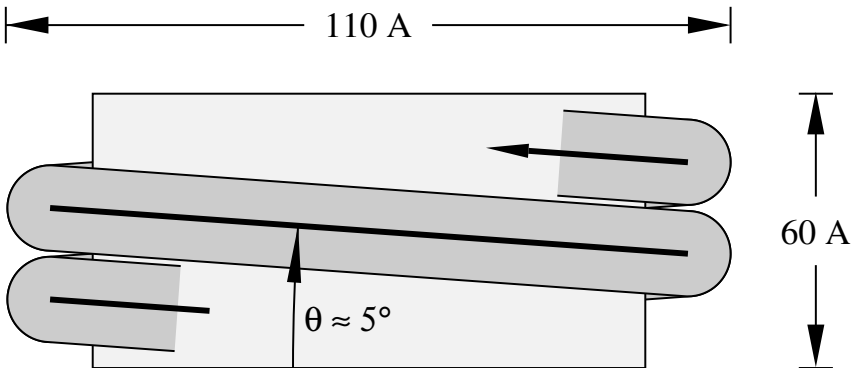
6.3.2 Nucleosome Model

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:

$$d = 3.38\sin(\theta);\ \phi = 360t/(n-1) \tag{6.1}$$

$$r = \frac{3.38(n-1)\cos(\theta)}{2\pi t} \tag{6.2}$$

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 );
```

```
20 matrx = newtransform( 0.0, 0.0, 0.0, -theta, 0.0, 0.0 );
21
22 m = newmolecule();
23 addstrand( m, "A" ); addstrand( m, "B" );
24 ttw = 0.0;
25 for( b = 1; b <= nbp; b = b + 1 ){
26     getbase( b, sbase, abase );
27     m1 = wc_helix( sbase, "", "dna", abase, "", "dna",
28         2.25, -4.96, 0.0, 0.0 );
29     mattw = newtransform( 0., 0., 0., 0., 0., ttw );
30     transformmol( mattw, m1, NULL );
31     transformmol( matrx, m1, NULL );
32     transformmol( matdx, m1, NULL );
33     maty = newtransform( 0.,dy*(b-1),0., 0.,-phi*(b-1),0.);
34     transformmol( maty, m1, NULL );
35
36     mergestr( m, "A", "last", m1, "sense", "first" );
37     mergestr( m, "B", "first", m1, "anti", "last" );
38     if( b > 1 ){
39         connectres( m, "A", b - 1, "O3'", b, "P" );
40         connectres( m, "B", 1, "O3'", 2, "P" );
41     }
42     ttw += TWIST; if( ttw >= 360.0 ) ttw -= 360.0;
43 }
44 putpdb( "nuc.pdb", m );
```

Finding the radius of the superhelix is a little tricky. In general a single turn of the helix will not contain an integral number of base pairs. For example, using typical numbers of 1.75 turns and 145 base pairs requires 82.9 base pairs to make one turn. An approximate solution can be found by considering the ideal superhelix that the DNA duplex is wrapped around. Let L be the arc length of this helix. Then $L\cos(\theta)$ is the arc length of its projection into the XZ plane. Since this projection is an overwound circle, L is also equal to $2\pi rt$, where t is the number of turns and r is the unknown radius. Now L is not known but is approximately $3.38(n-1)$. Substituting and solving for $r$gives Eq. 6.2.

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().

6.4 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.

6.4.1 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().

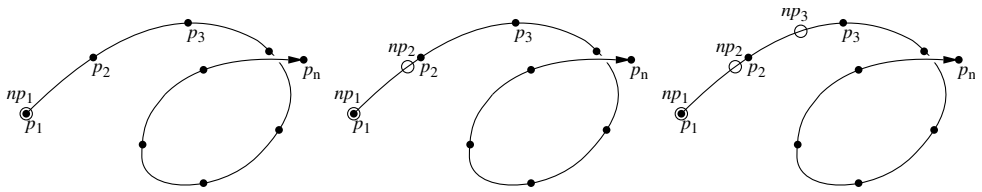
What is unusual about this interpolation is that the points at which the interpolation is to be performed are unknown. Instead, these points are chosen so that the distance between each point and its successor is the constant value RISE, set here to 3.38 which is the rise of an ideal B-DNA duplex. Thus, 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;

int spline();
int splint();

for( npts = 0; line = getline( stdin ); ){
    npts = npts + 1;
    a[ npts ] = npts;
    sscanf( line, "%lf %lf %lf",
        x[ npts ], y[ npts ], z[ npts ] );
}

spline( a, x, npts, 1e30, 1e30, x2, tmp );
spline( a, y, npts, 1e30, 1e30, y2, tmp );
spline( a, z, npts, 1e30, 1e30, z2, tmp );

li = 1; la = 1.0; lx = x[1]; ly = y[1]; lz = z[1];
printf( "%8.3f %8.3f %8.3f\\n", lx, ly, lz );

while( li < npts ){
    ni = li + 1;
    na = a[ ni ];
    nx = x[ ni ]; ny = y[ ni ]; nz = z[ ni ];
    dx = nx - lx; dy = ny - ly; dz = nz - lz;
    d = sqrt( dx*dx + dy*dy + dz*dz );
    if( d > RISE ){
        tfrac = frac = .5;
        for( i = 1; i <= MAXI; i = i + 1 ){
            na = la + tfrac * ( a[ni] - la );
            splint( a, x, x2, npts, na, nx );
            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.

6.4.2 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;
```

```
        sscanf( line, "%lf %lf %lf",
            pts[ npts ].x, pts[ npts ].y, pts[ npts ].z );
    }
}

if( pf != stdin )
    fclose( pf );

putdna( "dna.path", pts, npts );
```

6.4.3 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,
            "::ORG", "::ORG", "::SXT", "::ORG", "::CYT" );
        axis2frame( m_path, pts[ p ], pts[ p + 1 ] );
        alignframe( m_ax, m_path );
        mergestr( m_path, "A", "last", m_ax, "A", "first" );
        if( p > 1 ){
            setpoint( m_path, sprintf( "A:%d:CYT",p-1 ), p1 );
            setpoint( m_path, sprintf( "A:%d:ORG",p-1 ), p2 );
            setpoint( m_path, sprintf( "A:%d:ORG",p ), p3 );
            setpoint( m_path, sprintf( "A:%d:CYT",p ), p4 );
            tw = 36.0 - torsionp( p1, p2, p3, p4 );
            mat = rot4p( p2, p3, tw );
            aex = sprintf( ":%d:", p );
            transformmol( mat, m_path, aex );
            setpoint( m_path, sprintf( "A:%d:ORG",p ), p1 );
            setpoint( m_path, sprintf( "A:%d:SXT",p ), p2 );
            setpoint( m_path, sprintf( "A:%d:CYT",p ), p3 );
            setframep( 1, m_path, p1, p1, p2, p1, p3 );
        }

        getbase( p, sbase, abase );
        m_bp = wc_helix( sbase, "", "dna",
            abase, "", "dna",
            2.25, -5.0, 0.0, 0.0 );
        alignframe( m_bp, m_path );
        mergestr( m, "A", "last", m_bp, "sense", "first" );
        mergestr( m, "B", "first", m_bp, "anti", "last" );
        if( p >  1 ){
            connectres( m, "A", p - 1, "O3'", p, "P" );
            connectres( m, "B", 1, "P", 1, "O3'" );
        }
    }

    putpdb( mname + ".pdb", m );
    putbnd( mname + ".bnd", m );
};
```

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.

6.5 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] R. Brown; D. Case. Second derivatives in generalized Born theory. *J. Comput. Chem.*, **2006**, *27*, 1662–1675.

[2] 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.

[3] 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.

[4] C. Brooks; A. Brünger; M. Karplus. Active site dynamics in protein molecules: A stochastic boundary molecular-dynamics approach. *Biopolymers*, **1985**, *24*, 843–865.

[5] 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.

[6] A. Onufriev; D. Bashford; D. A. Case. Modification of the generalized Born model suitable for macromolecules. *J. Phys. Chem. B*, **2000**, *104*, 3712–3720.

[7] 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.

[8] 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.

[9] 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.

[10] D. T. Nguyen; D. A. Case. On finding stationary states on large-molecule potential energy surfaces. *J. Phys. Chem.*, **1985**, *89*, 4020–4026.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Z. J. Shi; J. Shen. New inexact line search method for unconstrained optimization. *J. Optim. Theory Appl.*, **2005**, *127*, 425–446.

[16] 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.

[17] D. C. Liu; J. Nocedal. On the limited memory method for large scale optimization. *Math. Programming B*, **1989**, *45*, 503–528.

[18] J. Nocedal; J. L. Morales. Automatic preconditioning by limited memory quasi-Newton updating. *SIAM J. Opt.*, **2000**, *10*, 1079–1096.

[19] 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.

[20] 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.

[21] R. Anandakrishnan; M. Daga; A. V. Onufriev. An n log n Generalized Born Approximation. *J. Chem. Theory Comput.*, **2011**, *7*, 544–559.

[22] 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.

[23] 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.

[24] 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.

[25] M. Turcotte; G. Lapalme; F. Major. Exploring the conformations of nucleic acids. *J. Funct. Program.*, **1995**, *5*, 443–460.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] R. Lavery; K. Zakrzewska; H. Skelnar. JUMNA (junction minimisation of nucleic acids). *Comp. Phys. Commun.*, **1995**, *91*, 135–158.

[31] 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.

[32] 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.

[33] G. M. Crippen; T. F. Havel. *Distance Geometry and Molecular Conformation*. Research Studies Press, Taunton, England, 1988.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] T. J. Macke. *NAB, a Language for Molecular Manipulation*. Ph.D. thesis, The Scripps Research Institute, 1996.

[41] R. E. Dickerson. Definitions and Nomenclature of Nucleic Acid Structure Parameters. *J. Biomol. Struct. Dyn.*, **1989**, *6*, 627–634.

[42] R. Tan; S. Harvey. Molecular Mechanics Model of Supercoiled DNA. *J. Mol. Biol.*, **1989**, *205*, 573–591.

[43] M. S. Babcock; E. P. D. Pednault; W. K. Olson. Nucleic Acid Structure Analysis. *J. Mol. Biol.*, **1994**, *237*, 125–156.

[44] T. F. Havel; I. D. Kuntz; G. M. Crippen. The theory and practice of distance geometry. *Bull. Math. Biol.*, **1983**, *45*, 665–720.

[45] 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.

[46] 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.

[47] 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.

[48] D. K. Agrafiotis. Stochastic Proximity Embedding. *J. Comput. Chem.*, **2003**, *24*, 1215–1221.

[49] W. Saenger. in *Principles of Nucleic Acid Structure*, p 120. Springer-Verlag, New York, 1984.