4/20/02

NAB Users' Manual

Thomas J. Macke and David A. Case

Department of Molecular Biology

The Scripps Research Institute

La Jolla, CA 92037

Some of the force field routines were adapted from similar routines in the MOIL program package: R. Elber, A. Roitberg, C. Simmerling, R. Goldstein, H. Li, G. Verkhivker, C. Keasar, J. Zhang and A. Ulitsky, "MOIL: A program for simulations of macromolecules" *Comp. Phys. Commun.* **91,** 159-189 (1995).

The "readparm" routine to access Amber topology and parameter files was adapted from code written by Bill Ross at the University of California, San Francisco.

The "trifix" routine for random pairwise metrization is based on an algorithm designed by Jay Ponder and was adapted from code in the Tinker package; see M.E. Hodsdon, J.W. Ponder, and D.P. Cistola, *J. Mol. Biol.* **264,** 585-602 (1996) and `http://dasher.wustl.edu/tinker/.`

The "molsurf" routines for computing molecular surface areas were adapted from routines written by Paul Beroza.

The "sasad" routine for computing derivatives of solvent accessible surface areas was kindly provided by S. Sridharan, A. Nicholls and K.A. Sharp. See *J. Computat. Chem.* **8,** 1038-1044 (1995).

The preprocessor (*ucpp*) was written by Thomas Pornin <thomas.pornin@ens.fr>, `http://www.di.ens.fr/~pornin/ucpp/`, and is distributed under a separate, BSD-style license. See `ucpp-0.7/README` for details.

The *cifparse* routines to deal with mmCIF formatted files were written by John Westbrook, and are distributed with permission. See `cifparse/README` for details.

The *teLeap* code comes from the Amber suite (see `http://www.amber.ucsf.edu/amber`); this portion is also distributed under the GNU General Public License.

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).

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 connection of NAB to AVS is described in B.S. Duncan, T.J. Macke and A.J. Olson, *J. Mol. Graphics* **13,** 271-282 (1995).

1. Installation and Getting Started.

1.1. Installation.

The nab package is available via anonymous ftp at ftp://ftp.scripps.edu/pub/case/nab-4.4.tar.gz as a compressed tar file. The first step in setting up the nab package is to unzip the tar file using the UNIX commands gunzip:

```
gunzip nab-4.4.tar.gz
tar xvf nab-4.4.tar.gz
```

The path to this new directory should be defined as the environment variable $NABHOME. The environment variables $NABHOME and $ARCH should be defined at this time, where $ARCH is the architecture type of the machine. For the most part, you can make up your own architecture name (it is just used to manage installations for different machines).

```
setenv NABHOME insertyourpathhere/nab-4.4

setenv ARCH yourarchitecture
```

Now, in the top-level ($NABHOME) directory, you should edit "config.h" to specify any variables particular to your site. There are sample files {config.h.windows, config.h.linux, config.h.generic, config.h.sgi, config.h.hp} that should get you started. For example, for many machines, the following will work:

```
cp config.h.generic config.h
```

(Instructions for what the options mean are in the config.h files) Then,

```
make
```

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

This can be followed by

```
make test
```

which will run tests and will report successes or failures.

Now, add the path to the binary executable of nab to your own path and rehash the search path, e.g.,

```
set path = ( $NABHOME/bin/$ARCH $path )

rehash
```

Now, you should be able to compile nab programs. Eventually, you may wish to define the environment variables $NABHOME and $ARCH and add the path to the binary executable of nab explicitly in your

1.2. 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] [-avs] [-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

`-avs` creates an `AVS` module

`-noassert` causes the compiler to ignore assert statements

`-nodebug` causes the compiler to ignore debug statements

`-o` *file* names the output file

`-D`*string* defines *string* to the C preprocessor

Linking FORTRAN and C object code with `nab` is accomplished simply by including the source files on the command line with the `nab` file. For instance, if a `nab` program *bar.nab* uses a C function defined in the file *foo.c*, compiling and linking optimized `nab` code would be accomplished by

```
nab -O bar.nab foo.c
```

The result is an executable `a.out` file.

1.3. Tested platforms

We have carried out the compilation and test programs on the following machines; for most of them we have also used NAB for several years.

(1) DEC AXP machines, under Digital Unix, with DEC compilers.

(2) SGI machines, with R4400 (32-bit) and R8000/R10000 (64-bit) architectures, using vendor-supplied compilers.

(3) Sun Sparc, under Solaris 2.5, using SunPRO compilers.

(4) Sun Sparc, under Solaris 2.5, using gcc 2.7.2, flex, and bison.

(5) HP 735 PA-RISC, under HP-UX 10, using vendor-supplied compilers.

(6) HP 735 PA-RISC, under HP-UX 10, using gcc 2.7.2 and flex.

(7) RedHat Linux on Intel Pentium, using gcc 2.95.2, flex, and bison.

(8) Windows 95/98/2000 on Intel Pentium, using the Cygwin deveopment kit, available from `http://sources.redhat/com`.

1.4. Contacting the developers

Please send suggestions and questions to `case@scripps.edu` or `macke@scripps.edu`. We would appreciate receiving a message if you use the program, so that we can send bug fixes and announcements of new versions.

2. General introduction and overview.

Nucleic acid builder (`nab`) is a high-level language that facilitates manipulations of macromolecules and their fragments. `nab` uses a C-like syntax for variables, expressions and control structures (`if`, `for`, `while`) and has extensions for operating on molecules (new types and a large number of builtins for providing the necessary operations). We expect `nab` to be useful in model building and coordinate manipulation of proteins and nucleic acids, ranging in size from fairly small systems to the largest systems for which an atomic level of description makes good computational sense. As a programming language, it is not a solution or program in itself, but rather provides an environment that eases many of the bookkeeping tasks involved in writing programs that manipulate three-dimensional structural models.

The current implementation is version 4.0, and incorporates the following main features:

(1) Objects such as points, atoms, residues, strands and molecules can be referenced and manipulated as named objects. The internal manipulations involved in operations like merging several strands into a single molecule are carried out automatically; in most cases the programmer need not be concerned about the internal data structures involved.

(2) Rigid body transformations of molecules or parts of molecules can be specified with a fairly high-level set of routines. This functionality includes rotations and translations about particular axis systems, least-squares atomic superposition, and manipulations of coordinate frames that can be attached to particular atomic fragments.

(3) Additional coordinate manipulation is achieved by a tight interface to distance geometry methods. This allows 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. You will need to have access to the *LEaP* module of AMBER to make full use of this facility.

(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, “hashed” arrays (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.

(8) Support is also present for compiling `nab` code into an AVS (Application Visualization System) module rather than to a stand-alone program. In combination with the AVS Geometry Viewer (or other AVS modules) this allows one to fairly easily build interactive programs that manipulate and display complex molecular transformations. We currently support AVS version 5; extensions to Data Explorer (openDX) are under development.

Our hope is that `nab` will serve to formalize the step-by-step process that is used to build complex model structures, and will facilitate the management and use of higher level symbolic constraints. Writing a program to create a structure forces more of the model’s assumptions to be explicit in the program itself. And an `nab` description can serve as a way to show a model’s salient features, much like helical parameters are used to characterize duplexes.

The first three chapters of this document form a “users’ manual” that both introduces the language through a series of sample programs, and illustrates the programming interfaces provided. The examples are chosen not only to show the syntax of the language, but also to illustrate potential approaches to the construction of some unusual nucleic acids, including DNA double- and triple-helices, RNA pseudoknots, four-arm junctions, and DNA-protein interactions. A separate “language reference manual” (in Chapter 4) gives a more formal and careful description of the requirements of the language itself.

The basic literature reference for the code is: T. Macke and D.A. Case. Modeling unusual nucleic acid structures. In *Molecular Modeling of Nucleic Acids*, N.B. Leontes and J. SantaLucia, Jr., eds. (Washington, DC: American Chemical Society, 1998), pp. 379-393. Users are requested to include this citation in papers that make use of NAB.

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 [1-3] is a high level molecular description language used to describe single stranded

---

1. F. Major, M. Turcotte, D. Gautheret, G. Lapalme, E. Fillon, and R. Cedergren, “The Combination of Symbolic and Numerical Computation for Three-Dimensional Modeling of RNA,” Science **253,** (5025)1255-1260 (1991).

2. D. Gautheret, F. Major, and R. Cedergren, “Modeling the three-dimensional structure of RNA using discrete nucleotide conformational sets,” J. Mol. Biol. **229,** 1049-1064 (1993).

3. W. Saenger, M. Turcotte, G. Lapalme, and F. Major, “Exploring the conformations of nucleic acids,” J. Funct. Program. **5,** 443-460 (1995). Springer-Verlag,

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.* [4] 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 [5,6] 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 [7]. 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

---

4. D.A. Erie, K.J. Breslauer, and W.K. Olson, "A Monte Carlo Method for Generating Structures of Short Single-Stranded DNA Sequenes," Biopolymers **33,** (1)75-105 (1993).

5. C.-S. Tung and E.S. Carter, II, "Nucleic acid modeling tool (NAMOT): an interactive graphic tool for modeling nucleic acid structures," CABIOS **10,** 427-433 (1994).

6. E.S. Carter, II and C.-S. Tung, "NAMOT2--a redesigned nucleic acid modeling tool: construction of non-canonical DNA structures," CABIOS **12,** 25-30 (1996).

7. V. B. Zhurkin, Yu. P. Lysov, and V. I. Ivanov, "Different Families of Double Stranded Conformations of DNA as Revealed by Computer Calculations," Biopolymers **17,** 277-312 (1978).

JUMNA [8], 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.* [9] 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 [10] 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 interated `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.

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 nonbonded geometry, or where it may not be obvious how to place the bases.

---

8. R. Lavery, K. Zakrzewska, and H. Skelnar, "JUMNA (junction minimisation of nucleic acids)," Comp. Phys. Commun. **91,** 135-158 (1995).

9. J. Gabarro-Arpa, J.A.H. Cognet, and 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. **10,** 166-173 (1992).

10. M. Le Bret, J. Gabarro-Arpa, J. C. Gilbert, and C. Lemarechal, "MORCAD an object-oriented molecular modeling package," J. Chim. Phys. **88,** 2489-2496 (1991).

`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 correspondance 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.

Distance geometry. `nab`'s second initial structure-creation method is *metric matrix distance geometry* [11,12], which can be a very powerful method of creating initial structures. It has two main strengths. First, since it uses internal coordinates, the initial position of atoms about which nothing is known may be left unspecified. This has the effect that distance geometry models use only the information the modeler considers valid. No assumptions are required concerning the positions of unspecified atoms. The second advantage is that much structural information is in the form of distances. These include constraints from NMR or fluorescence energy transfer experiments, implied propinquities from chemical probing and footprinting, and tertiary interactions inferred from sequence analysis. Distance geometry provides a way to formally incorporate this information, or other assumptions, into the model-building process.

Distance geometry converts a molecule represented as a set of interatomic distances into a 3-D structure. `nab` has several builtin functions that are used together to provide metric matrix distance geometry. A `bounds` object contains the molecule's interatomic distance bounds matrix and a list of its chiral centers and their volumes. The function `newbounds()` creates a `bounds` object containing a distance bounds matrix containing initial upper and lower bounds for every pair of atoms, and a list of the molecule's chiral centers and their volumes. Distance bounds for pairs of atoms involving only a single residue are derived from that residue's coordinates. The 1,2 and 1,3 distance bounds are set to the actual distance between the atoms. The 1,4 distance lower bound is set to the larger of the sum of the two atoms Van der Waals radii or their *syn* (torsion angle = 0°) distance, and the upper bound is set to their *anti* (torsion angle = 180°) distance. `newbounds()` also initializes the list of the

---

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

12. D.C. Spellmeyer, A.K. Wong, M.J. Bower, and J.M. Blaney, "Conformational analysis using distance geometry methods," J. Mol. Graph. Model. **15,** 18-36 (1997).

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 expermental measurements or parts of the model's hypothesis. The functions `andbounds()` and `orbounds()` allow logical manipulation of bounds. `setbounds_from_db()Fr 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 specifed 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 [13] 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.

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 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 is a good method for refining structures created either by rigid body transformations or distance geometry. `nab` has its own 3-D/4-D molecular mechanics package that implements several AMBER force fields and reads AMBER parameter and topology files. Solvation effects can also be modelled with generalized Born continuum models.

Our hope is that `nab` will serve to formalize the step-by-step process that is used to build complex model structures. It will facilitate the management and use of higher level symbolic constraints. Writing a program to create a structure forces one to make explicit more of the model's assumptions in the program itself. And an nab description can serve as a way to exhibit a model's salient features, much like helical parameters are used to characterize duplexes. So far, nab has been used to construct

---

13. M.E. Hodsdon, J.W. Ponder, and 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. **264,** 585-602 (1996).

14. T. Macke, S.-M. Chen, and W.J. Chazin, in *Structure and Function, Volume 1: Nucleic Acids,* R.H. Sarma and M.H. Sarma, Ed. (Adenine Press, Albany, 1992). pp. 213-227.

models for synthetic Holliday junctions [14], calcyclin dimers [15], HMG-protein/DNA complexes [16], active sites of Rieske iron-sulfur proteins [17], and supercoiled DNA [18]. The Examples chapter below provides a number of other sample applications.

2.3. 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.

2.3.1. 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 3.38Å rise and 36.0° 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"

15. B.C.M. Potts, J. Smith, M. Akke, T.J. Macke, K. Okazaki, H. Hidaka, D.A. Case, and W.J. Chazin, "The structure of calcyclin reveals a novel homodimeric fold S100 Ca^{2+}-binding proteins," Nature Struct. Biol. **2,** 790-796 (1995).

16. J.J. Love, X. Li, D.A. Case, K. Giese, R. Grosschedl, and P.E. Wright, "DNA recognition and bending by the architectural transcription factor LEF-1: NMR structure of the HMG domain complexed with DNA," Nature **376,** 791-795 (1995).

17. R.J. Gurbiel, P.E. Doan, G.T. Gassner, T.J. Macke, D.A. Case, T. Ohnishi, J.A. Fee, D.P. Ballou, and 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 **35,** 7834-7845 (1996).

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

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 readabilty.

2.3.2. 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 = %8.3f\n", 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.

2.3.3. Place residues in a standard orientation.

This is the last of the introductory examples. It places nucleic acid monomers in an orientation that is useful for building Watson/Crick base pairs. It uses several atom expressions to create a frame or handle attached to an nab molecule that permits easy movement along important "molecular directions". In a standard Watson/Crick base pair the C4 and N1 atoms of the purine base and the H3, N3 and C6 atoms of the pyrimdine 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 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 writen 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.

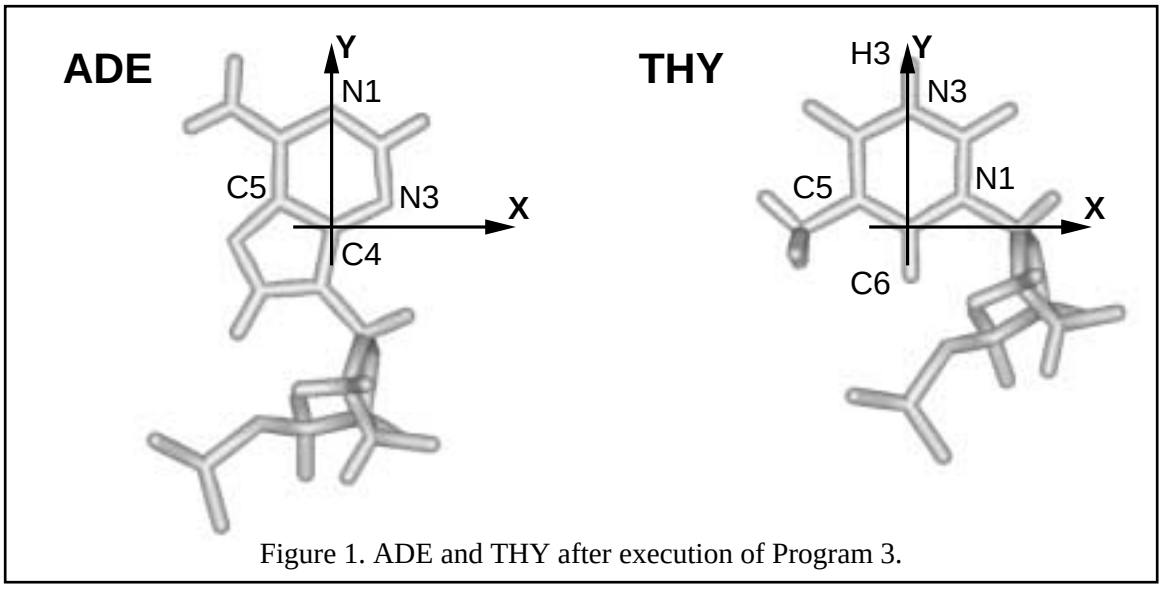


Figure 1. ADE and THY after execution of Program 3.

2.4. 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 accessable 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.

2.5. Creating Molecules.

The following functions are used to create molecules. Only an overview is given here; more details are in chapter 3.

```
molecule newmolecule();
int     addstrand( molecule m, string str );

residue getresidue( string rname, string rlib );
residue transformres( matrix mat, residue res, string aex );
int     addresidue( molecule m, string str, residue res );
```

```
int     connectres( molecule m, string str,
                    int rn1, string atm1, int rn2, string atm2 );

int     mergestr( molecule m1, string str1, string end1,
                  molecule m2, string str2, string end2 );
```

The general strategy for creating molecules with `nab` is to create a new (empty) molecule then build it one residue at a time. Each residue is fetched from a residue library, transformed to properly position it and added to a growing strand. A template showing this strategy is shown below. `mat`, `m` and `res` are respectively a `matrix`, `molecule` and `residue` variable declared elsewhere. Words in italics indicate general instances of things that would be filled in according to actual application.

```
1     ...
2     m = newmolecule();
3     addstrand( m, str-1 );
4     ...
5     for( ... ){
6         ...
7         res = getresidue( res-name, res-lib );
8         res = transformres( mat, res, NULL );
9         addresidue( m, str-name, res );
10        ...
11    }
12    ...
```

In line 2, the function `newmolecule()` creates a molecule and stores it in `m`. The new molecule is empty—no strands, residues or atoms. Next `addstrand()` is used to add a strand named *str-1*. Strand names may be up to 255 characters in length and can include any characters except white space. Each strand in a molecule must have a unique name. There is no limit on the number of strands a molecule may have.

The actual structure would be created in the loop on lines 5-11. Each time around the loop, the function `getresidue()` is used to extract the next residue with the name *res-name* from some residue library *res-lib* and stores it in the `residue` variable `res`. Next the function `transformres()` applies a transformation matrix, held in the `matrix` variable `mat` to the residue in `res`, which places it in the orientation and position it will have in the new molecule. Finally, the function `addresidue()` appends the transformed residue to the end of the chain of residues in the strand *str-name* of the new molecule.

Residues in each strand are numbered from 1 to *N*, where *N* is the number of residues in that strand. The residue order is the order in which they were inserted with `addresidue()`. While `nab` does not require it, nucleic acid chains are usually numbered from 5’ to 3’ and proteins chains from the N-terminus to the C-terminus. The residues in nucleic acid strands and protein chains are usually bonded with the outgoing end of residue *i* bonded to the incoming end of residue *i*+1. However, as this is not always the case, `nab` requires the user to explicitly make all interresidue bonds with the builtin `connectres()`.

`connectres()` makes bonds between two atoms in different residues of the same strand of a molecule. Only residues in the same strand can be bonded. `connectres()` takes six arguments.

They are a molecule, the name of the strand containing the residues to be bonded, and two pairs each of a residue number and the name of an atom in that residue. As an example, this call to `connectres()`,

```
connectres( m, "sense", i, "O3'", i+1, "P" );
```

connects an atom named `"O3'"` in residue `i` to an atom named `"P"` in residue `i+1`, creating the phosphate bond that joins two nucleic acid monomers.

The function `mergestr()` is used to either move or copy the residues in one strand into another strand. Details are provided in chapter 3.

2.6. Residues and Residue Libraries.

`nab` programs build molecules from residues that are parts of residue libraries. Residue libraries contain coordinates and bonding information for each of their entries. They may also contain additional information such as the type of the residues, (dna, rna, amino acid or unknown), the level of atomic detail (all atoms including hydrogens, united atom with only hydrogen bonding hydrogens or unknown), lists of chiral centers including those for enforcing planarity, atomic charges and radii. `nab` is distributed with four residue libraries—A- and B-DNA, RNA and an amino acid library that produces fully extended peptides. In addition to `nab` residue libraries (denoted by a `.rlb` suffix), `nab` can also read residues from the `LEaP` object file format (OFF) files such as `all_amino94.lib` (denoted by a A complete description of an `nab` residue library can be found in the `nab` Language Reference.

`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 `$NABHOME/reslib` unless it begins with a slash (`/`)

A common task of many `nab` programs is the translation of a string of characters into a structure where each letter in the string represents a residue. Generally, some mapping of one or two character names into actual residue names is required. `nab` supplies the function `getres()` that maps the single character names `a`, `c`, `g`, `t` and `u` and their 5' and 3' terminal analogues into the residues `ADE`, `CYT`, `GUA`, `THY` and `URA`. Here is its source:

```
1     // getres() - map 1-2 letter names into 3 letter names
```

```
2     residue getres( string rname, string rlib )
3     {
4             residue res;
5
6             if( r == "a" || r == "A" ){
7                     res = getresidue( "ADE", rlib );
8             }else if( r == "c" || r == "C" ){
9                     res = getresidue( "CYT", rlib );
10            }else if( r == "g" || r == "G" ){
11                    res = getresidue( "GUA", rlib );
12            }else if( r == "t" || r == "T" ){
13                    res = getresidue( "THY", rlib );
14            }else if( r == "u" || r == "U" ){
15                    res = getresidue( "URA", rlib );
16            }else{
17                    fprintf( stderr, "undefined residue %s0, r );
18                    exit( 1 );
19            }
20            return( res );
21    };
```

`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 21. Line 21 ends with the final semicolon.

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-existant values of the types string, file, atom, residue, molecule and bounds. A value of NULL generally means that a variable is unitialized or that an error ocurred 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.

2.7. 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 molcule.

An important property of `nab` atom expressions is that the order in which the strands, residues, and atoms are listed is unimportant. *i.e.*, the atom expression `"2,1:5,2,3:N1,C1'"` is the exact same atom expression as `"1,2:3,2,5:C1',N1"`. All atom expressions are reordered, internal to `nab`, in increasing atom number. So, in the above example, the selected atoms will be selected in the following sequence:

```
1:2:N1
1:2:C1'
1:3:N1
1:3:C1'
1:5:N1
1:5:C1'
2:2:N1
2:2:C1'
2:3:N1
2:3:C1'
2:5:N1
2:5:C1'
```

The order in which atoms are selected internal to a specific residue are the order in which they appear in a `nab` PDB file. As seen in the above example, `N1` appears before `C1'` in all `nab` nucleic acid residues and PDB files.

2.8. 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 resdiue 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, speicified 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 proton, all calculations involving that atom can be skipped. The `if` in line 10 tests to see if `ai` is a proton. If 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, the 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()`.

2.9. 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`.

2.9.1. 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.

2.9.2. 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 specifed 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.) Similary, rhe 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 coordindates of the input molecule. It returns a 0 on success and 1 on failure. In both functions, the special atom expression NULL selects all atoms in the input residue or molecule.

2.9.3. 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.

2.10. 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. `getres()` which maps one letter residue names into actual residue names was discussed in the section **Residues and Residue Libraries**.

2.10.1. 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;
7             cseq = wc_complement( seq, "dna.amber94.rlb", "dna" );
```

```
8             m = wc_helix( seq, "dna.amber94.rlb", "dna",
9                       cseq, "dna.amber94.rlb", "dna",
10                      2.25, -4.96, 36.0, 3.38, "s5a5s3a3" );
11             return( m );
12    };
```

`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 phospates 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 occured 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 library (in this case, *dna.amber94.rlb*) 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 the Language Reference chapter below.

2.10.2. 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     {
5        string acbase, base, wcbase, wcseq;
6        int i, len;
8
9        if( rlt == "dna" )
10               acbase = "t";
11       else if( rlt == "rna" )
12               acbase = "u";
13       else{
14          fprintf( stderr,
15            "wc_complement: rlib is not dna/rna, no W/C comp.",
16                 rlib );
```

```
17          return( NULL );
18      }
20      len = length( seq );
21      wcseq = NULL;
22      for( i = 1; i <= len; i = i + 1 ){
23          base = substr( seq, i, 1 );
24          if( base == "a" || base == "A" )
25                  wcbase = acbase;
26          else if( base == "c" || base == "C" )
27                  wcbase = "g";
28          else if( base == "g" || base == "G" )
29                  wcbase = "c";
30          else if( base == "t" || base == "T" )
31                  wcbase = "a";
32          else if( base == "u" || base == "U" )
33                  wcbase = "a";
34          else{
35              fprintf( stderr,
36                 "wc_complement: unknown base %sn",
37                  base );
38              return( NULL );
39          }
40          wcseq = wcseq + wcbase;
41      }
42      return( wcseq );
43  }
```

`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`. If the residue library is not nucleic acid, the complementary sequence can not be created and `wc_complement()` returns the value `NULL` indicating failure. The complementary sequence is created in the `for` loop that begins in line 22 and extends to line 41. 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 24 to 39 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 40 shows how `nab` uses the infix `+` to concatentate 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.

2.10.3. 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 adjacent base pairs. A complete description of helical parameters can be found in Dickerson [19]

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.

2.10.4. 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) 180° about the X-axis. After applying this transformation, the two bases are still occupying the same space but ares is now antiparallel to sres. Line 90 creates a transformation matrix that displaces m_anti and ares along the Y-axis by sep Å. The properly positioned molecules containing sres and ares are merged into a single molecule, m, completing the base pair. Lines 95-96 move this base pair to a more convenient

19. R. E. Dickerson, "Definitions and Nomenclature of Nucleic Acid Structure Parameters," J. Biomol. Struct. Dyn. **6,** (4)627-634 (1989).

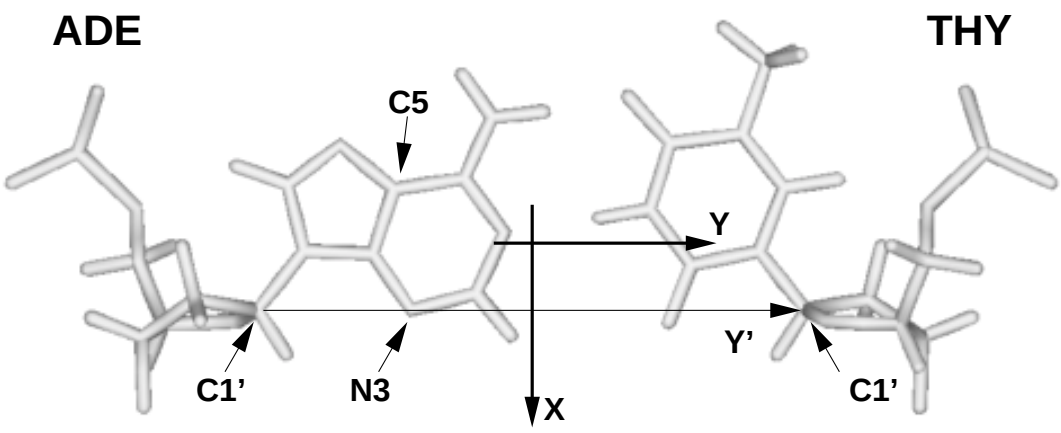


Figure 2. ADE:THY from wc_basepair().

orientation for helix generation. Initially the base as shown in Figure 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     molecule wc_basepair( residue sres, residue ares )
5     {
6             residue r;
7             molecule m;
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 %s0,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 %s0,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              setframe( 2, m, "::C1'", xtail, xhead, ytail, yhead );
96              alignframe( m, NULL );
97              return( m );
98      };
```

2.10.5. 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 19-34 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 38-87. 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 getres() and assembled into a base pair by wc_base-pair(). 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 95-118. substr() is used to extract the appropriate letters from seq and aseq which are converted into another base pair by getres() 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 con-nectres() 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 lines 108-109 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 184-194 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    residue sres, ares;
15    int     has_s, has_a;
16    int i, slen;
17    float   ttwist, trise;
18
```

```
19      has_s = 1; has_a = 1;
20      if( sreslib == "" ) sreslib_use = "dna.amber94.rlb";
21              else sreslib_use = sreslib;
22      if( areslib == "" ) areslib_use = "dna.amber94.rlb";
23              else areslib_use = areslib;
24
25      if( seq == NULL && aseq == NULL ){
26              fprintf( stderr, "wc_helix: no sequence0 );
27              return( NULL );
28      }else if( seq == NULL ){
29              seq = wc_complement( aseq, areslib_use, snatype );
30              has_s = 0;
31      }else if( aseq == NULL ){
32              aseq = wc_complement( seq, sreslib_use, anatype );
33              has_a = 0;
34      }
35
36      slen = length( seq );
37
38      srname = substr( seq, 1, 1 );
39      setreslibkind( sreslib, snatype );
40      if ( substr( sreslib_use, length(sreslib_use)-2,
41              length(sreslib_use ) ) == "rlb" ){
42              if( opts =~ "s5" )
43                      sres = getres( srname, substr( sreslib_use, 1,
44                              length(sreslib_use)-3 ) + "5.rlb");
45              else if( opts =~ "s3" && slen == 1 )
46                      sres = getres( srname, substr( sreslib_use, 1,
47                              length(sreslib_use)-3 ) + "3.rlb");
48              else sres = getres( srname, sreslib_use );
49      }else if ( substr( sreslib_use, length(sreslib_use)-2,
50              length(sreslib_use ) ) == "lib" ){
51              if( opts =~ "s5" )
52                      sres = getres( srname + "5", sreslib_use );
53              else if( opts =~ "s3" && slen == 1 )
54                      sres = getres( srname + "3", sreslib_use );
55              else sres = getres( srname, sreslib_use );
56      }else{
57              fprintf(stderr,
58              "wc_helix : unknown sense residue library : %s0,
59                      sreslib_use );
60              exit( 1 );
61      }
62
63      arname = substr( aseq, 1, 1 );
64      setreslibkind( areslib, anatype );
65      if ( substr( areslib_use, length(areslib_use)-2,
```

```
66              length(areslib_use ) ) == "rlb" ){
67              if( opts =~ "a3" )
68                      ares = getres( arname, substr( areslib_use, 1,
69                      length(areslib_use)-3 ) + "3.rlb");
70              else if( opts =~ "a5" && slen == 1 )
71                      ares = getres( arname, substr( areslib_use, 1,
72                      length(areslib_use)-3 ) + "5.rlb");
73              else ares = getres( arname, areslib_use );
74      }else if ( substr( areslib_use, length(areslib_use)-2,
75              length(areslib_use ) ) == "lib" ){
76              if( opts =~ "a3" )
77                      ares = getres( arname + "3", areslib_use );
78              else if( opts =~ "a5" && slen == 1 )
79                      ares = getres( arname + "5", areslib_use );
80              else ares = getres( arname, areslib_use );
81      }else{
82              fprintf(stderr,
83              "wc_helix : unknown anti residue library : %s0,
84                      areslib_use );
85              exit( 1 );
86      }
87      m1 = wc_basepair( sres, ares );
88      freeresidue( sres ); freeresidue( ares );
88
89      xomat = newtransform(xoff, 0., 0., 0., 0., 0. );
90      transformmol( xomat, m1, NULL );
91      inmat = newtransform( 0., 0., 0., incl, 0., 0.);
92      transformmol( inmat, m1, NULL );
93
94      trise = rise; ttwist = twist;
95      for( i = 2; i <= slen-1; i = i + 1 ){
96              srname = substr( seq, i, 1 );
97              setreslibkind( sreslib, snatype );
98              sres = getres( srname, sreslib_use );
99              arname = substr( aseq, i, 1 );
100             setreslibkind( areslib, anatype );
101             ares = getres( arname, areslib_use );
102             m2 = wc_basepair( sres, ares );
102             freeresidue( sres ); freeresidue( ares );
103             transformmol( xomat, m2, NULL );
104             transformmol( inmat, m2, NULL );
105             mat = newtransform( 0., 0., trise,
106                     0., 0., ttwist );
107             transformmol( mat, m2, NULL );
108             mergestr( m1, "sense", "last",
109                     m2, "sense", "first" );
110             connectres( m1, "sense",
```

```
111                 i-1, "O3'", i, "P" );
112           mergestr( m1, "anti", "first",
113                 m2, "anti", "last" );
114           connectres( m1, "anti",
115                 1, "O3'", 2, "P" );
116           trise = trise + rise;
117           ttwist = ttwist + twist;
117           freemolecule( m2 );
118     }
119
120     i = slen;         // add in final residue pair
121     if( i > 1 ){
122
123           srname = substr( seq, i, 1 );
124           setreslibkind( sreslib, snatype );
125           if ( substr( sreslib_use, length(sreslib_use)-2,
126                 length(sreslib_use ) ) == "rlb" ){
127                 if( opts =~ "s3" )
128                       sres = getres( srname, substr( sreslib_use, 1,
129                       length(sreslib_use)-3 ) + "3.rlb");
130                 else
131                       sres = getres( srname, sreslib_use );
132           }else if ( substr( sreslib_use, length(sreslib_use)-2,
133                 length(sreslib_use ) ) == "lib" ){
134                 if( opts =~ "s3"  )
135                       sres = getres( srname + "3", sreslib_use );
136                 else
137                       sres = getres( srname, sreslib_use );
138           }else{
139                 fprintf(stderr,
140                 "wc_helix : unknown sense residue library : %s0,
141                       sreslib_use );
142                 exit( 1 );
143           }
144           arname = substr( aseq, i, 1 );
145           setreslibkind( areslib, anatype );
146           if ( substr( areslib_use, length(areslib_use)-2,
147                 length(areslib_use ) ) == "rlb" ){
148                 if( opts =~ "a5" )
149                       ares = getres( arname, substr( areslib_use, 1,
150                       length(areslib_use)-3 ) + "5.rlb");
151                 else
152                       ares = getres( arname, areslib_use );
153           }else if ( substr( areslib_use, length(areslib_use)-2,
154                 length(areslib_use ) ) == "lib" ){
155                 if( opts =~ "a5" )
156                       ares = getres( arname + "5", areslib_use );
```

```
157                     else
158                             ares = getres( arname, areslib_use );
159             }else{
160                     fprintf(stderr,
161                     "wc_helix : unknown anti residue library : %s0,
162                             areslib_use );
163                     exit( 1 );
164             }
165
166             m2 = wc_basepair( sres, ares );
166             freeresidue( sres ); freeresidue( ares );
167             transformmol( xomat, m2, NULL );
168             transformmol( inmat, m2, NULL );
169             mat = newtransform( 0., 0., trise,
170                     0., 0., ttwist );
171             transformmol( mat, m2, NULL );
172             mergestr( m1, "sense", "last",
173                     m2, "sense", "first" );
174             connectres( m1, "sense",
175                     i-1, "O3'", i, "P" );
176             mergestr( m1, "anti", "first",
177                     m2, "anti", "last" );
178             connectres( m1, "anti",
179                     1, "O3'", 2, "P" );
180             trise = trise + rise;
181             ttwist = ttwist + twist;
181             freemolecule( m2 );
182     }
183
184     m3 = newmolecule();
185     addstrand( m3, "sense" );
186     addstrand( m3, "anti" );
187     if( has_s )
188            mergestr( m3, "sense", "last",
189            m1, "sense", "first" );
190     if( has_a )
191            mergestr( m3, "anti", "last",
192                    m1, "anti", "first" );
193     freemolecule( m1 );
194
195     return( m3 );
196     };
```

2.11. Structure Quality and Energetics.

Up to this point, all the structures in the examples have been built using only transformations. These transformations properly place the purine and pyrimidine rings. However, since they are rigid

body transformations, they will create distorted sugar/backbone geometry if any internal sugar/backbone rearrangements are required to accomodate the base geometry. The amount of this distortion depends on both the input residues and transformations applied and can vary from trivial to so severe that the created structures are useless. `nab` offers two methods for fixing bad sugar/backbone geometry. They are molecular mechanics and distance geometry. `nab` provides distance geometry routines and has its own molecular mechanics package. The latter requires the *LEaP* program, which is part of the *AMBER* suite of programs developed at the University of California, San Francisco and at The Scripps Research Institute. Information about how to obtain this program is available on the Internet at `http://www.amber.ucsf.edu/amber/amber.html`. Details on the routines involved are given in the **Language Reference** chapter, and some examples are given below.

2.11.1. Creating a Parallel DNA Triplex.

Parallel DNA triplexes are thought to be intermediates in homologous DNA recombination. These triplexes, investigated by Zhurkin *et al.* [20] are called R-form DNA, and are believed to exist in two distinct conformations. In the presence of recombination proteins (eg. RecA), they adopt an extended conformation that is underwound with respect to standard helices (a twist of 20°) and very large base stacking distances (a rise of 5.1Å). However, in the absence of recombination proteins, R-form DNA exists in a "collapsed" form that resembles conventional triplexes but with two very important differences—the two parallel strands have the same sequence and the triplex can be made from any Watson/Crick duplex irregardless of its base composition. The remainder of this section discusses how this triplex could be modeled and two `nab` programs that implement that strategy.

If the degrees of freedom of a triplex are specified by the helicoidal parameters requried to place the bases, then a triplex of N bases has $6(N - 1)$ degrees of freedom, an impossibly large number for any but trivial N. Fortunately, the nature of homologous recombination allows some simplifying assumptions. Since the recombination must work on *any* duplex, the overall shape of the triplex must be sequence independent. This implies that each helical step uses the same set of transformational parameters which reduces the size of the problem to six degrees of freedom once the individual base triads have been created.

The individual triads are created by assuming that they are planar, that the third base is hydrogen bonded on the major groove side of the base pair as it appears in a standard Watson/Crick duplex, that the original Watson Crick base pair pair is essentially undisturbed by the insertion of the third base and finally that the third base belongs at the point that maximizes its hydrogen bonding with respect to the original Watson/Crick base pair. After the optimized triads have been created, they are assembled into dimers. The dimers assume that the helical axis passes through the center of the circle defined by the positions of the three `C1'` atoms. Several instances of a two parameter family (rise, twist) of dimers are created for each of the 16 pairs of triads and minimized.

2.11.2. Creating Base Triads.

Here is an `nab` program that computes the vacuum energy of `XY:X` base triads as a function of the position and orientation of the `X` (non-Watson/Crick) base. A minimum energy `AU:A` found by the program along with the potential energy surface keyed to the position of the second `A` is shown in

---

20. V. B. Zhurkin, G Raghunathan, N. B. Ulynaov, R. D. Camerini-Otero, and R. L. Jernigan, "A Parallel DNA Triplex as a Model for the Intermediate in Homologous Recombination," Journal of Molecular Biology **239,** 181-200 (1994).

Figure 3. The program creates a single Watson/Crick DNA base pair and then computes the energy of a third DNA base at each position of a user defined rectangular grid. Since hydrogen bonding is both distance and orientation dependent the program allows the user to specify a range of orientations to try at each grid point. The orientation giving the lowest energy at each grid point and its associated energy are written to a file. The position and orientation giving the lowest overall energy is saved and is used to *recreate* the best triad after the search is completed.

```
1     // Program 5 - Investigate energies of base triads
2     molecule m;
3     residue tr;
4     string sb, ab, tb;
5     matrix rmat, tmat;
6
7     file ef;
8     string mfnm, efnm;
9     point txyz[ 35 ];
10    float x, lx, hx, xi, mx;
11    float y, ly, hy, yi, my;
12    float rz, lrz, hrz, rzi, urz, mrz, brz;
13
14    int prm;
15    point xyz[ 100 ], force[ 100 ];
16    float me, be, energy;
17
18    scanf( "%s %s %s", sb, ab, tb );
19    scanf( "%lf %lf %lf", lx, hx, xi );
20    scanf( "%lf %lf %lf", ly, hy, yi );
21    scanf( "%lf %lf %lf", lrz, hrz, rzi );
22
23    mfnm = sprintf( "%s%s%s.triad.min.pdb", sb, ab, tb );
24    efnm = sprintf( "%s%s%s.energy.dat", sb, ab, tb );
25
26    m = wc_helix(sb, "dna.amber94.rlb", "dna", ab,
27            "dna.amber94.rlb", "dna", 2.25, 0.0, 0.0, 0.0 );
28
29    addstrand( m, "third" );
30    tr = getres( tb, "dna.amber94.rlb" );
31    addresidue( m, "third", tr );
32    setxyz_from_mol( m, "third::", txyz );
33
34    leap( m, "", "" ); readparm( m, "prmtop" );
35    mme_init( m, NULL, "::ZZZ", xyz, NULL );
36
37    ef = fopen( efnm, "w" );
38
39    mrz = urz = lrz - 1;
```

```
40      for( x = lx; x <= hx; x = x + xi ){
41          for( y = ly; y <= hy; y = y + yi ){
42              brz = urz;
43              for( rz = lrz; rz <= hrz; rz = rz + rzi ){
44                  setmol_from_xyz( m, "third::", txyz );
45                  rmat=newtransform( 0., 0., 0., 0., 0., rz );
46                  transformmol( rmat, m, "third::" );
47                  tmat=newtransform( x, y, 0., 0., 0., 0. );
48                  transformmol( tmat, m, "third::" );
49
50                  setxyz_from_mol( m, NULL, xyz );
51                  energy = mme( xyz, force, 1 );
52
53                  if( brz == urz ){
54                      brz = rz; be = energy;
55                  }else if( energy < be ){
56                      brz = rz; be = energy;
57                  }
58                  if( mrz == urz ){
59                      me = energy;
60                      mx = x; my = y; mrz = rz;
61                  }else if( energy < me ){
62                      me = energy;
63                      mx = x; my = y; mrz = rz;
64                  }
65              }
66              fprintf( ef, "%10.3f %10.3f %10.3f %10.3fn",
67                      x, y, brz, be );
68          }
69      }
70      fclose( ef );
71
72      setmol_from_xyz( m, "third::", txyz );
73      rmat = newtransform( 0.0, 0.0, 0.0, 0.0, 0.0, mrz );
74      transformmol( rmat, m, "third::" );
75      tmat = newtransform( mx, my, 0.0, 0.0, 0.0, 0.0 );
76      transformmol( tmat, m, "third::" );
77      putpdb( mfnm, m );
```

Program 5 begins by reading in a description of the desired triad and data defining the location and granularity of the search area. It does this with the calls to the `nab` builtin `scanf()` on lines 18-21. `scanf()` uses its first argument as a “format” string which directs the conversion of text versions of `int`, `float` and `string` values into their internal formats. The first call to `scanf()` reads the three letters that specify the bases, the next two calls read the X and Y location, extent and granularity of the the search rectangle and the last call reads in the first, last and increment values that will be used specify the orientation of the third base at each point on the search grid.

Lines 23 and 24 respectively, create the names of the files that will hold the best structure found and the values of the potential energy surface. The file names are created using the builtin `sprintf()`. Like `scanf()` this function also uses its first argument as a format string, used here to construct a string from the data values that follow it in the parameter list. The action of these calls is to replace the each format descriptor (`%s`) with the values of the corresponding string variable in the parameter list. The file names created for the `AU:A` shown in Figure 3 were `AUA.triad.min.pdb` and `AUA.energy.dat`. Format expressions and formatted I/O including the I/O like `sprintf()` are discussed in the sections **Format Expressions** and **Ordinary I/O Functions** of the **`nab` Language Reference**.

The triad is created in two major steps in lines 26-32. First a Watson/Crick base pair is created with `wc_helix()`. The base pair has an X-offset of 2.25Å and an inclination of 0.0 meaning it lies in the XY plane. Twist and rise although they are not used in creating a single base pair are also set to 0.0. The X-offset which is that of standard B-DNA was chosen to faciliate extension of triplexes made from the triads created here with standard duplex DNA. Absent this consideration any X-offset including 0.0 would have been satisfactory. A third strand (`"third"`) is added to `m`, the string `tb` is converted into a DNA residue and this residue is added to the new strand. Finally in the coordinates of the third strand are saved in the `point` array `txyz`. Referring to Figure 3, the third base is located directly on top of the Watson/Crick pair. A purine would have its `C4` atom at the origin and its `C4-N1` vector along the Y axis; a pyrimidine its `C6` at the origin and its `C6-N3` vector along the Y axis. Obviously this is not a real structure; however, as will be seen in the next section, this initial placement greatly simplifies the transformations required to explore the search area.

2.11.3. Finding the lowest energy triad.

The energy calculation begins in line 34 and extends to line 69. Elements of the general molecular mechanics code skeleton discussed in the **Language Reference** chapter are seen at lines 34-35 and lines 50-51. Initialization takes place in lines 34 and 35 with the call to `leap()` to prepare the `prmtop` that contains the information needed to compute molecular mechanics energies. This is followed by the call to `readparm` which reads back in the newly created `prmtop` file, and creates an internal data structure. The force field routine is initialized in line 35, asking that all atoms be allowed to move. The actual energy calculation is done in lines 50 and 51. `setxyz_from_mol()` copies the current conformation of `mol` into the `point` array `xyz` and then `mme()` evaluates the energy of this conformation. Note that the energy evaluation is in a loop, in this case nested inside the three loops that control the conformational search.

The search area shown in Figure 3 is on the left side of the Watson/Crick base pair. This corresponds to inserting the third base into the major groove of the duplex. Now as the third base is initially positioned at the origin with its hydrogen bonding edge pointing towards the top of the page, it must be both moved to the left or in the -X direction and rotated approximately -90° so that its hydrogen bonding sites can interact with those on the left side of the Watson/Crick pair.

The search is executed by the three nested `for` loops in lines 40, 41 and 43. They control the third base's X and Y position and its orientation in the XY plane. Two transformations are used to place the base. The first step of the placement process is in line 44 where the `nab` builtin `setmol_from_xyz()` is used to restore the original (untransformed) coordinates of the base. The call to `newtransform()` in line 45 creates a transformation matrix that will point the third base so that its hydrogen bonding sites are aimed in the positive X direction. A second transformation matrix created on line 47 is used to move the properly oriented third base to a point on the search area. The call to `setxyz_from_mol()` extracts the coordinates of this conformation into `xyz` and `mme()` computes and returns its energy.

The remainder of the loop determines if this is either the best overall energy or the best energy for this grid point. Lines 53-57 compute the best energy at this point and lines 58-64 compute the best overall energy. The complexity arises from the fact that the energy returned by `mme()` can be any `float` value. Thus it is not possible to to pick a value that is guaranteed to be higher than any value returned during the search. The solution is to use the value from the first iteration of the loop as the value to test against. The two variables `mrz` and `brz` are used to indicate the very first iteration and the first iteration of the rz loop. The gray rectangle of Figure 3 shows the vacuum energy of the best `AU:A` triad found when the origin of the X' Y' axes are at that point on the rectangle. Darker grays are lower energies. Figure 3 shows the best `AU:A` found.

2.11.4. Assembling the Triads into Dimers.

Once the minimized base triads have been created, they must be assembled into triplexes. Since these triplexes are believed to be intermediates in homologous recombination, their structure should be nearly sequence independent. This means that they can be assembled by applying the same set of helical parameters to each optimized triad. However, several things still need to be determined. These are the location of the helical axis and just what helical parameters are to be applied. This code assumes that the three backbone strands are roughly on the surface of a cylinder whose axis is the global helical axis. In particular the helical axis is the center of the circle defined by the three `C1'` atoms in each triad. While the four circles defined by the four minimized triads are not exactly the same, their radii are within XÅ of each other with the XY:X triad having the largest offset of YÅ. The code makes two additional assumptions. The sugar rings are all in the C2'-endo conformation and the triads are not

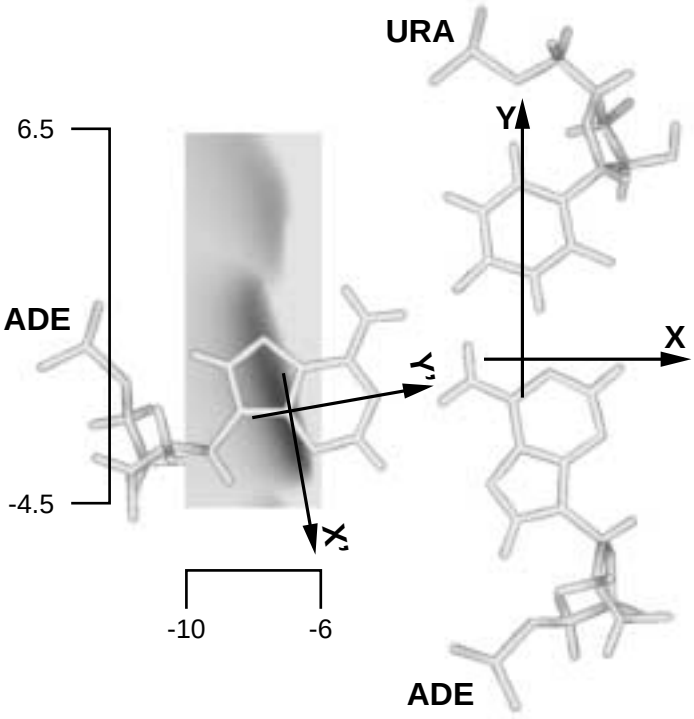


Figure 3. Minimum energy AUA triad and Potential Energy Surface.

inclined with respect to the helical axis. The program that creates and evaluates the dimers is shown below. A detailed explanation of the program follows the listing.

```
1    // Program 6 - Assemble triads into dimers
2    molecule    gettriad( string mname )
3    {
4        molecule    m;
5        point       p1, p2, p3, pc;
6        matrix      mat;
7
8        if( mname == "a" ){
9            m = getpdb( "ata.triad.min.pdb" );
10           setpoint( m, "A:ADE:C1'", p1 );
11           setpoint( m, "B:THY:C1'", p2 );
12           setpoint( m, "C:ADE:C1'", p3 );
13       }else if( mname == "c" ){
14           m = getpdb( "cgc.triad.min.pdb" );
15           setpoint( m, "A:CYT:C1'", p1 );
16           setpoint( m, "B:GUA:C1'", p2 );
17           setpoint( m, "C:CYT:C1'", p3 );
```

```
18        }else if( mname == "g" ){
19            m = getpdb( "gcg.triad.min.pdb" );
20            setpoint( m, "A:GUA:C1'", p1 );
21            setpoint( m, "B:CYT:C1'", p2 );
22            setpoint( m, "C:GUA:C1'", p3 );
23        }else if( mname == "t" ){
24            m = getpdb( "tat.triad.min.pdb" );
25            setpoint( m, "A:THY:C1'", p1 );
26            setpoint( m, "B:ADE:C1'", p2 );
27            setpoint( m, "C:THY:C1'", p3 );
28        }
29        circle( p1, p2, p3, pc );
30        mat = newtransform( -pc.x, -pc.y, -pc.z, 0.0, 0.0, 0.0 );
31        transformmol( mat, m, NULL );
32        setreskind( m, NULL, "DNA" );
33        return( m );
34    };
35
36    int mk_dimer( string ti, string tj )
37    {
38        molecule    mi, mj;
39        matrix      mat;
40        int     sid;
41        float       ri, tw;
42        string      ifname, sfname, mfname;
43        file        idx;
44
45        int     natoms;
46        float       dgrad, fret;
47        float       box[ 3 ];
48        float       xyz[ 1000 ];
49        float       fxyz[ 1000 ];
50        float       energy;
51
52        sid = 0;
53        mi = gettriad( ti );
54        mj = gettriad( tj );
55        mergestr( mi, "A", "3'", mj, "A", "5'" );
56        mergestr( mi, "B", "5'", mj, "B", "3'" );
57        mergestr( mi, "C", "3'", mj, "C", "5'" );
58        connectres( mi, "A", 1, "O3'", 2, "P" );
59        connectres( mi, "B", 1, "O3'", 2, "P" );
60        connectres( mi, "C", 1, "O3'", 2, "P" );
61
62        leap( mi, "", "" );
63        readparm( mi, "prmtop" );
64
```

```
65          ifname = sprintf( "%s%s3.idx", ti, tj );
66          idx = fopen( ifname, "w" );
67          for( ri = 3.2; ri <= 4.4; ri = ri + .2 ){
68              for( tw = 25; tw <= 45; tw = tw + 5 ){
69                  sid = sid + 1;
70                  fprintf( idx, "%3d %5.1f %5.1f", sid, ri, tw );
71
72                  mi = gettriad( ti );
73                  mj = gettriad( tj );
74
75                  mat = newtransform( 0.0, 0.0, ri, 0.0, 0.0, tw );
76                  transformmol( mat, mj, NULL );
77
78                  mergestr( mi, "A", "3'", mj, "A", "5'" );
79                  mergestr( mi, "B", "5'", mj, "B", "3'" );
80                  mergestr( mi, "C", "3'", mj, "C", "5'" );
81                  connectres( mi, "A", 1, "O3'", 2, "P" );
82                  connectres( mi, "B", 1, "O3'", 2, "P" );
83                  connectres( mi, "C", 1, "O3'", 2, "P" );
84
85                  sfname = sprintf( "%s%s3.%03d.pdb", ti, tj, sid );
86                  putpdb( sfname, mi );   // starting coords
87
88                  natoms = getmolyz( mi, NULL, xyz );
89                  mme_init( m, NULL, "::ZZZ", xyz, NULL );
90
91                  dgrad = 0.1;
92                  conjgrad( xyz, 3*natoms, fret, mme, dgrad, 10, 100 );
93                  energy = mme( xyz, fxyz, 1 );
94
95                  setmol_from_xyz( mi, NULL, xyz );
96                  mfname = sprintf( "%s%s3.%03d.min.pdb", ti, tj, sid );
97                  putpdb( mfname, mi );   // minimized coords
98              }
99          }
100         fclose( idx );
101     };
102
103     int i, j;
104     string  ti, tj;
105     for( i = 1; i <= 4; i = i + 1 ){
106         for( j = 1; j <= 4; j = j + 1 ){
107             ti = substr( "acgt", i, 1 );
108             tj = substr( "acgt", j, 1 );
109             mk_dimer( ti, tj );
110         }
111     }
```

Program 6 assembles, minimizes and writes the final energies of a family of dimers for each of the 16 pairs of optimized triads. The program is long but straightforward. It is organized into two subroutines followed by a main program. The first subroutine `gettriad()` is defined in lines 2-34, the second subroutinue `mk_dimer()` in lines 36-101 and the main program in lines 103-111. The overall organization is that the main program controls the sequence of the dimers beginning with `AA` and continuing with `AC`, `AG`, ... and on up to `TT`. Each time it selects the sequence of the dimer, it calls `mk_dimer()` to explore the family of structures defined by variation in the rise and twist. `mk_dimer()` in turn calls `gettriad()` to fetch and orient the specified base triples.

The function `gettriad()` (lines 2-34) takes a string with one of the four values `"a"`, `"c"`, `"g"` or `"t"`. The `if`-tree in lines 8-28 uses this string to select the coordinates of the corresponding optimized triad. The `if`-tree sets the value of the three points `p1`, `p2` and `p3` that will be used to define the circle whose center will intersect the global helical axis. Once these points are defined, the `nab` builtin `circle()` (line 29) returns the center of the circle they define in `pc`. The builtin `circle()` returns a `1` if the three points do not define a circle and a `0` if they do. In this case it is known that the positions of the three `C1'` atoms are well behaved, so the return value is ignored. The selected triad is properly centered in lines 30-31. Each residue of the triad is set to be of type `"DNA"` via the call to `setreskind()` in line 32 so that its atomic charges and forcefield potentials can be set correctly to perform the minimization. The new molecule is returned as the function's value in line 33.

The dimers are created by the function `mk_dimers()` that is defined in lines 36-101. The process uses two stages. The molecule is first prepared for molecular mechanics in lines 53-63 and then dimers are created and minimized in the two nested loops in lines 67-99. The results of the minimizations are stored in a file whose name is derived from the name of the triads in the dimer. For example, the results for an `AA` would be in the file `"aa3.idx"`. There is one file for each of the 16 dimers. The file name is created in line 65 and opened for writing in line 66. It is closed just before the function returns in line 100. Each line of the file contains a number that identifies the dimer's parameters followed by its rise, twist and final (minimized) energy.

In order to perform molecular on a molecule the `nab` program must create a "parameter" structure for it. This structure contains the topology of the molecule and parameters for the various terms of forcefield—things like bond lengths and angles, torsions, chirality and planarity. This is done in lines 53-63. The particular dimer is created. The function `gettriad()` is called twice to return the two properly centered triads in the molecules `mi` and `mj`. Next the three strands of `mj` are merged into the three strands of `mi` to create a triplex of length 2. The `"A"` and `"B"` strands form the Watson/Crick pairs of the triplex and the `"C"` strand contains the strand that is parallel to the `"A"` strand. The three' calls to `connectres()` create an `O3'-P` bond between the newly added residue and the existing residues in each of the three strands. After all this is done, the call to `leap()` in line 64 builds the parameter structure, returning `1` on failure and `0` on success.

This section of code seems simple enough except for one thing—the two triads in the dimer are obviously directly on top of each other. However, this is not a problem because `leap()` ignores the molecule's coordinates. Instead it uses the molecule's residue names to get each residue's internal coordinates and other information from a library which it uses to up the parameter and topology structure required by the minimization routines.

The dimers are built and minimized in the two nested loops in lines 69-104. The outer loop varies the rise from 3.2Å to 4.4Å by 0.2Å and the inner loop varies the twist from 25° to 45° in steps of 5° creating 35 different starting dimers. The variable `sid` is a number that identifies each (rise,twist) pair. It is inserted into the file names of the starting coordinates (lines 85-86) and minimized coordinates (lines 96-97) to make it easy to identify them.

Each dimer is created in lines 72-83. The two specifiifed triads are returned by the calls to `get-triad()` as the molecule's `mi` and `mj`. Next the triad in `mj` is transformed to give it the current rise and twist with respect to the triad in `mi`. The transformed triad in `mj` is merged into `mi` and bonded to `mi`. These starting coordinates are written to a file whose name contains both the dimer sequence and `sid`. For example, the first dimer for `AA` would be `"aa3.01.pdb"`, the `01` indicating that this dimer used a rise of 3.2Å and a twist of 25°.

The minimization is performed in lines 88-95. The call to `setxyz_from_mol()` extracts the current atom positions of `mi` into the array `xyz`. The coordinates are passed to `mme_init()` which initializes the molecular mechanics system. The actual minimization is done with the call to `conj-grad()` which performs `100` cycles of conjugate gradient minimization, printing the results every `10` cycles. The final energy is written to the file `idx` and the molecule's original coordinates are updated with the minimized coordinates by the call to `setmol_from_xyz()`. Once all dimers have been made for this sequence the loops terminate. The last thing done by `mk_dimer()` before it returns to the main program is to close the file containing the energy results for this family of dimer.

The very simple main program follows `mk_dimer()`. It consists of two nested loops that produce the pairs of strings (`"a"`,`"a"`), (`"a"`,`"c"`),...,(`"t"`,`"t"`) calling `mk_dimer()` for each pair.

3. NAB Language Reference.

3.1. Introduction.

nab is a computer language used to create, modify and describe models of macromolecules, especially those of unusual nucleic acids. The following sections provide a complete description of the nab language. The discussion begins with its lexical elements, continues with sections on expressions, statements and user defined functions and concludes with an explanation of each of nab's builtin functions. Two appendices contain a more detailed and formal description of the lexical and syntactic elements of the language including the actual lex and yacc input used to create the compiler. Two other appendices describe nab's internal data structures and the C code generated to support some of nab's higher level operations.

3.2. Language Elements.

An nab program is composed of several basic lexical elements: identifiers, reserved words, literals, operators and special characters. These are discussed in the following sections.

3.2.1. Identifiers.

An identifier is a sequence of letters, digits and underscores beginning with a letter. Upper and lower case letters are distinct. Identifiers are limited to 255 characters in length. The underscore (_) is a letter. Identifiers beginning with underscore must be used carefully as they may conflict with operating system names and nab created temporaries. Here are some nab identifiers.

```
mol   i3   twist   TWIST   Watson_Crick_Base_Pair
```

3.2.2. Reserved Words.

Certain identifiers are reserved words, special symbols used by nab to denote control flow and program structure. Here are the nab reserved words:

allocate	assert	atom	bounds	break
continue	deallocate	debug	delete	dynamic
else	file	for	float	hashed
if	in	int	matrix	molecule
point	residue	return	string	while

3.2.3. Literals.

Literals are self defining terms used to introduce constant values into expressions. nab provides three types of literals: integers, floats and character strings. Integer literals are sequences of one or more decimal digits. Float literals are sequences of decimal digits that include a decimal point and/or are followed by an exponent. An exponent is the letter e or E followed by an optional + or - followed by one to three decimal digits. The exponent is interpreted as "times 10 to the power of *exp*" where *exp* is the number following the e or E. All numeric literals are base 10. Here are some integer and float literals:

```
1   3.14159   5   .234   3.0e7   1E-7
```

String literals are sequences of characters enclosed in double quotes ("). A double quote is placed into a string literal by preceding it with a backslash (\). A backslash is inserted into a string by preceding it with a backslash. Strings of zero length are permitted.

```
""   "a string"   "string with a \""   "string with a \\"
```

Non-printing characters are inserted into strings via escape sequences: one to three characters following a backslash. Here are the nab string escapes and their meanings:

\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 backslash.
\ooo	character with value ooo where ooo is 1 to 3 octal digits (0-7).
\xhh	character with value hh where hh is 1 or 2 hex digits (0-9,a-f,A-F).

Here are some strings with escapes:

"Molecule\tResidue\tAtom\n"	Two tabs and a newline.
"\252Real quotes\272"	Octal values, \252, the left double quote “ and \272 the right double quote ”.

3.2.4. Operators.

nab uses several additional 1 or 2 character symbols as operators. Operators combine literals and identifiers into expressions.

Operator	Meaning	Precedence	Associates
`( )`	Expression grouping	9	
`[ ]`	Array indexing	9	
`.`	Select attribute	8	
Unary `-`	Negation	8	Right to left
`!`	Not	8	
`^`	Cross product	7	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`	Member of hashed array, or atom in a molecule	4	
`&&`	And	3	
`||`	Or	2	
`=`	Assignment	1	Right to left

3.2.5. Special Characters.

`nab` uses braces (`{}`) to group statements into compound statements and statements and declarations into function bodies. The semicolon (`;`) is used to terminate statements. The comma (`,`) separates items in parameter lists and declarations. The sharp (#) used in column 1 designates a preprocessor directive, which invokes the standard C preprocessor to provide constants, macros and file inclusion. A # in any other column, except in a comment or a literal string is an error. Two consecutive forward slashes (`//`) indicate that the rest of the line is a comment which is ignored. All other characters except white space (spaces, tabs, newlines and formfeeds) are illegal except in literal strings and comments.

3.3. Higher-level constructs.

3.3.1. Variables.

A variable is a name given to a part of memory that is used to hold data. Every `nab` variable has type which determines how the computer interprets the variable's contents. `nab` provides 10 data types. They are the numeric types `int` and `float` which are translated into the underlying C com-

piler'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 `string`s are atomic objects. The `atom`, `residue`, `molecule` and `bounds` types become pointers to the appropriate C `struct`s. `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`.

3.3.2. Attributes.

Four `nab` types—`atom`, `residue`, `molecule` and `point`—have attributes which are elements of their internal structure directly accessible at the `nab` level. Attributes are accessed via the select operator (`.`) which takes a variable as its left hand operand and an attribute name (an identifier) as its right. The general form is

var`.`*attr*

Most attributes behave exactly like ordinary variables of the same type. However, some attributes are read only. They are not permitted to appear as the left hand side of an assignment. When a read only attribute is passed to an `nab` function, it is copied into temporary variable which in turn is passed to the function. Read only attributes are not permitted to appear as destination variables in `scanf()`

*This translation of `float` to `double` is new at version 3.0. Previous versions of the code used (single-precision) `float` variables in both C and NAB programs. Carrying out manipulations in double-precision generally helps numerical stability, especially for distance geometry and molecular mechanics calculations. The earlier behavior can be re-obtained by changing the *defreal.h* header file.

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`	`float`	Yes	The atom's X coordinate.
`y`	`float`	Yes	The atom's Y coordinate.
`z`	`float`	Yes	The atom's Z coordinate.
`charge`	`float`	Yes	
`radius`	`float`	Yes	
`int1`	`int`	Yes	User settable int value.
`float1`	`float`	Yes	User settable float value.

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 starting at `1`. `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 for each strand. 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.

3.3.3. Arrays.

`nab` supports two kinds of arrays—ordinary arrays where the selector is a comma separated list of integer expressions and associative or “hashed” arrays where the selector is a character string. The set of character strings that is associated with data in a hashed array is called its keys. Array elements may be of any `nab` type. All the dimensions of an ordinary array are indexed from `1` to N_d, where N_d is the size of the d^{th} dimension. Non parameter array declarations are similar to scalar declarations except the variable name is followed by either a comma separated list of integer constants surrounded by square brackets (`[]`) for ordinary arrays or the reserved word `hashed` in square brackets for associative arrays. Associative arrays have no predefined size.

```
float    energy[ 20 ], surface[ 13,13 ];
int      attr[ dynamic, dynamic ];
molecule structs[ hashed ];
```

The syntax for multi-dimensional arrays like that for Fortran, not C. The *nab2c* compiler linearizes all index references, and the underlying C code sees only single-dimension arrays. Arrays are stored in "column-order", so that the most-rapidly varying index is the first index, as in Fortran. Multi-dimensional `int` or `float` arrays created in *nab* can generally be passed to Fortran routines expecting the analogous construct.

Dynamic arrays are not allocated space upon program startup, but are created and freed by the `allocate` and `deallocate` statements:

```
allocate  attr[ i, j ];
 ....
deallocate attr;
```

Here `i` and `j` must be integer expressions that may be evaluated at run-time. It is an error (generally fatal) to refer to the contents of such an array before it has been allocated or after it has been deallocated.

3.3.4. Expressions.

Expressions use operators to combine variables, constants and function values into new values. `nab` uses standard algebraic notation (`a+b*c`, etc) for expressions. Operators with higher precedence are evaluated first. Parentheses are used to alter the evaluation order. The complete list of `nab` operators with precedence levels and associativity is listed under **Operators**.

`nab` permits mixed mode arithmetic in that `int` and `float` data may be freely combined in expressions as long as the operation(s) are defined. The only exceptions are that the modulus operator (`%`) does not accept `float` operands, and that subscripts to ordinary arrays must be integer valued. In all other cases except parameter passing and assignment, when an `int` and `float` are combined by an operator, the `int` is converted to `float` then the operation is executed. In the case of parameter passing, `nab` requires (but does not check) that actual parameters passed to functions have the same type as the corresponding formal parameters. As for assignment (`=`) the right hand side is converted to the type of the left hand side (as long as both are numeric) and then assigned. `nab` treats assignment like any other binary operator which permits multiple assignments (`a=b=c`) as well as "embedded" assignments like:

```
if( mol = newmolecule() ) ...
```

`nab` relational operators are strictly binary. Any two objects can be compared provided that both are numeric, both are `string` or both are the same type. Comparisons for objects other than `int`, `float` and `string` are limited to tests for equality. Comparisons between `file`, `atom`, `residue`, `molecule` and `bounds` objects test for "pointer" equality, meaning that if the pointers are the same, the objects are same and thus equal, but if the pointers are different, no inference about the actual objects can be made. The most common comparison on objects of these types is against `NULL` to see if the object was correctly created. Note that as `nab` considers `NULL` to be false the following expressions are equivalent.

`if(` *var* `== NULL )...`	is the same as	`if( !`*var*` )...`
`if(` *var* `!= NULL )...`	is the same as	`if(` *var* `)...`

The Boolean operators `&&` and `||` evaluate only enough of an expression to determine its truth value. `nab` considers the value `0` to be false and *any* non-zero value to be true. `nab` supports direct assignment and concatenation of string values. The infix `+` is used for `string` concatenation.

`nab` provides several infix vector operations for `point` values. They can be assigned and `point` valued functions are permitted. Two `point` values can be added or subtracted. A `point` can be multiplied or divided by a `float` or an `int`. The unary minus can be applied to a `point` which has the same effect as multiplying it by `-1`. Finally, the at sign (`@`) is used to form the dot product of two `point`s and the circumflex (`^`) is used to form their cross product.

3.3.5. 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. These are not documented here, but they should be, and we will try to work on that for the next version of this document.

3.3.6. Atom Expressions.

An atom expression is a character string that contains one or more patterns that match a set of atom names in a molecule. Atom expressions contain three substrings separated by colons (`:`). They represent the strand, residue and atom parts of the atom expression. Each subexpression consists of a comma (`,`) separated list of patterns, or for the residue part, patterns and/or number ranges. Several atom expressions may be placed the 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 1.

3.3.7. Format Expressions.

A format expression is a special character string that is used to direct the conversion between the computer's internal data representations and their character equivalents. `nab` uses the underlying C compiler's `printf()`/`scanf()` system to provide formatted I/O. This section provides a short introduction to this system. For the complete description, consult any standard C reference. Note that since `nab` supports fewer types than its underlying C compiler, formatted I/O options pertaining to the data subtypes (`h`,`l`,`L`) are not applicable to `nab` format expressions.

An input format string is a mixture of ordinary characters, *spaces* and format descriptors. An output format string is mixture of ordinary characters including spaces and format descriptors. Each format descriptor begins with a percent sign (%) followed by several optional characters describing the format and ends with single character that specifies the type of the data to be converted. Here are the most common format descriptors. The `...` represent optional characters described below.

`% ... c`	convert a character
`% ... d`	convert an 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, irregardless of their alignment. Left aligned conversions are also always filled with spaces. However, right aligned string and numeric conversions can use a `0` to indicate that left fill should be zeroes instead of spaces. In addition numeric conversions can also specify an optional + to indicate that non-negative numbers should be preceded by a plus sign. The default action for numeric conversions is that negative numbers are preceded by a minus, and other numbers have no sign. If both `0` and `+` are specified, their order does not matter.

Output field width and precision are last and are specified by one or two integers or stars (`*`) separated by a period (`.`). The first number (or star) is the field width, the second is its precision. If the precision is not specified, a default precision is chosen based on the conversion type. For floats (`%f`), it is six decimal places and for strings it is the entire string. Precision is not applicable to character or integer conversions and is ignored if specified. Precision may be specified without the field width by use of single integer (or star) preceded by a period. Again, the action is conversion type dependent. For strings (`%s`), the action is to print the first *N* characters of the string or the entire string, whichever is shorter. For floats (`%f`), it will print *N* decimal places but will extend the field to whatever size if required to print the whole number part of the float. The use of the star (`*`) as an output width or precision indicates that the width or precision is specified as the next argument in the conversion list which allows for runtime widths and precisions.

Output Format Options	
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.

3.4. Statements.

`nab` statements describe the action the `nab` program is to perform. The expression statement evaluates expressions. The `if` statement provides a two way branch. The `while` and `for` statements provide loops. The `break` statement is used to "short circuit" or exit these loops. The `continue` statement advances a `for` loop to its next iteration. The `return` statement assigns a function's value and returns control to the caller. Finally a list of statements can be enclosed in braces (`{}`) to create a compound statement.

3.4.1. Expression Statement.

An expression statement is an expression followed by a semicolon. It evaluates the expression. Many expression statements include an assignment operator and its evaluation will update the values of those variables on the left hand side of the assignment operator. These kinds of expression statements are usually called "assignment statements" in other languages. Other expression statements consist of a single function call with its result ignored. These statements take the place of "call statements" in other languages. Note that an expression statement can contain *any* expression, even ones that have no lasting effect.

```
mref = getpdb( "5p21.pdb" );          // "assignment" stmt
m = getpdb( "6q21.pdb" );
superimpose( m,"::CA",mref,"::CA" );  // "call" stmt

0;                                    // expression stmt.
```

3.4.2. Delete Statement.

`nab` provides the `delete` statement to remove elements of `hashed` arrays. The syntax is

```
delete h_array[ str ];
```

where *h_array* is a `hashed` array and *str* is a `string` valued expression. If the specified element is in *h_array* it is removed; if not, the statement has no effect.

3.4.3. If Statement.

The `if` statement is used to choose between two options based on the value of the `if` expression. There are two kinds of `if` statements—the simple `if` and the `if-else`. The simple `if` contains an expression and a statement. If the expression is true (any non-zero value), the statement is executed. If the expression is false (`0`), the statement is skipped.

```
if( expr )
        true_stmt;
```

The `if-else` statement places two statements under control of the `if`. One is executed if the expression is true, the other if it is false.

```
if( expr )
        true_stmt;
else
        false_stmt;
```

The single statement in a simple `if` or the two statements in an `if-else` can be any `nab` statement(s) including other `if` statements. This can introduce ambiguity as to which `if` is associated with an `else`:

```
if( expr_1 )
        if( expr_2 )
                stmt_1;
        else
                stmt_2;
```

Which `if` has the `else`, the `if` on the first line or the `if` on the second? The rule is that an `else` is associated with the nearest unpaired `if`. In this example, the `else` is associated with the `if` on the second line. To associate the `else` with the `if` on line 1 would require hiding the inner `if` inside braces:

```
if( expr_1 )
{
        if( expr_2 )
                stmt_1;
}
else
       stmt_2;
```

The braces convert the inner `if` into a compound statement removing the ambiguity.

3.4.4. While Statement.

The `while` statement is used to execute the statement under its control as long as the the `while` expression is true (non-zero). A compound statement is required to place more than one statement under the `while` statement's control.

```
while( expr )
        stmt;
```

```
while( expr )
{
        stmt_1;
        stmt_2;
        ...
        stmt_N;
}
```

3.4.5. For Statement.

The `for` statement is a loop statement that allows the user to include initialization and an increment as well as a loop condition in the loop header. The single statement under the control of the `for` statement is executed as long as the condition is true (non-zero). A compound statement is required to place more than one statement under control of a `for`. The general form of the `for` statement is

```
for( expr_1; expr_2; expr_3 )
        stmt;
```

which behaves like

```
expr_1;
while( expr_2 )
{
        stmt;
        expr_3;
}
```

expr_3 is generally an expression that computes the next value of the loop index. Any or all of *expr_1*, *expr_2* or *expr_3* can be omitted. An omitted *expr_2* is considered to be true, thus giving rise to an "infinite" loop. Here are some `for` loops.

```
for( i = 1; i <= 10; i = i + 1 )
        printf( "%3d\n", i );       // print 1 to 10

for( ; ; )                          // "infinite" loop
{
        getcmd( cmd );              // Exit better be in
        docmd( cmd );               // getcmd() or docmd().
}
```

`nab` also includes a special kind of `for` statement that is used to range over all the entries of a hashed

array or all the atoms of a molecule. The forms are

```
// hashed version
for( str in h_array )
        stmt;

// molecule version
for( a in mol )
        stmt;
```

In the first code fragment, *str* is string and *h_array* is a hashed array. This loop sets *str* to each key or string associated with data in *h_array*. Keys are returned in increasing lexical order. In the second code fragment *a* is an atom and *mol* is a molecule. This loop sets *a* to each atom in *mol*. The first atom is the first atom in the first residue of the first strand. Once all the atoms in this residue have been visited, it moves to the first atom of the next residue in the first strand. Once all atoms in all residues in the first strand have been visited, the process is repeated on the second and subsequent strands in mol until all atoms have been visited. The order of the strands of molecule is the order in which they were created using `addstrand()`. Residues in each strand are numbered from 1 to *N*. The order of the atoms in a residue is the order in which the atoms were listed in the reslib entry or pdbfile that that residue derives from.

3.4.6. Break Statement.

Execution of a `break` statement exits the immediately enclosing `for` or `while` loop. By placing the `break` under control of an `if` conditional exits can be created. `break` statements are only permitted inside `while` or `for` loops.

```
for( expr_1; expr_2; expr_3 )
{
        ...
        if( expr )
                break;              // "break" out of loop
        ...
}
```

3.4.7. Continue Statement.

Execution of a `continue` statement causes the immediately enclosing `for` loop to skip to its next value. If the next value causes the loop control expression to be false, the loop is exited. `continue` statements are permitted only inside `while` and `for` loops.

```
for( expr_1; expr_2; expr_3 )
{
        ...
```

```
        if( expr )
                continue;           // "continue" with next value
        ...
}
```

3.4.8. Return Statement.

The `return` statement has two uses. It terminates execution of the current function returning control to the point immediately following the call and when followed by an optional expression, returns the value of the expression as the value of the function. A function's execution also ends when it "runs off the bottom". When a function executes the last statement of its definition, it returns even if that statement is not a `return`. The value of the function in such cases is undefined.

```
return expr;        // return the value expr
return;             // return, function value undefined.
```

3.4.9. Compound Statement.

A compound statement is a list of statements enclosed in braces. Compound statements are required when a loop or an `if` has to control more than one statement. They are also required to associate an `else` with an `if` other than the nearest unpaired one. Compound statements may include other compound statements. Unlike C, `nab` compound statements are not blocks and may not include declarations.

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` `string`s are indexed from `1` to *N* where *N* is the number of characters in the string.

```
int     length( string s );

int     index( string s, string t );
int     match( string s, string r, int rlength );

string  substr( string s, int pos, int len );

int     split( string s, string fields[], string fsep );

int     sub( string r, string s, string t );
int     gsub( string r, string s, string t );
```

`length()` returns the length of the string `s`. Both `""` and `NULL` have length `0`. `index()` returns the position of the left most occurrence of `t` in `s`. If `t` is not in `s`, `index()` returns `0`. `match` returns the position of the longest leftmost substring of `s` that matches the regular expression `r`. The length of this substring is returned in `rlength`. If no substring of `s` matches `r`, `match()` returns `0` and `rlength` is set to `0`. `substr()` extracts the substring of length `len` from `s` beginning at position `pos`. If `len` is greater than the rest of the string beginning at `pos`, return the substring from `pos` to *N* where *N* is the length of the `string`. If `pos` is < `1` or > *N*, return `""`.

`split()` partitions `s` into fields separated by `fsep`. These field strings are returned in the array `fields`. The number of fields is returned as the function value. The array `fields` must be allocated before `split()` is called and must be large enough to hold all the field strings. The action of `split()` depends on the value of `fsep`. If `fsep` is a string containing one or more blanks, the fields of `s` are considered to be separated by *runs* of white space. Also, leading and trailing white space in `s` do not indicate an empty initial or final field. However, if `fsep` contains any value but blank, then fields are considered to be delimited by *single* characters from `fsep` and initial and/or trailing `fsep` characters do represent initial and/or trailing fields with values of `""`. `NULL` and the empty string `""`

have `0` fields. If both `s` and `fsep` are composed of only white space then `s` also has `0` fields. If `fsep` is not white space and `s` consists of nothing but characters from `fsep`, `s` will have *N* + 1 fields of `""` where *N* is the number of characters of `s`.

`sub()` replaces the leftmost longest substring of `t` that matches the regular expression `r`. `gsub()` replaces all non overlapping substrings of `t` that match the regular expression `r` with the string `s`.

3.8. Math Functions.

`nab` provides the following builtin mathematical functions. Since `nab` is intended for chemical structure calculations which always measure angles in degrees, the argument to the trig functions—`cos()`, `sin()` and `tan()`— and the return value of the inverse trig functions—`acos()`, `asin()`, `atan()` and `atan2()`—are in degrees instead of radians as they are in other languages.

`nab` **Builtin Mathematical Functions**	
`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 y, float x );`	Return $\tan^{-1}$(y / x) in degrees. By keeping x and y separate, 90° can be returned without encountering a zero divide. Also, `atan2` will return an angle in the full range [-180°, 180°].
`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( int iseed );`	Return random number in (0,1) and update seed.
`float ceil( float x );`	Return $\lceil x \rceil$.
`float cosh( float x );`	Return the hyberbolic cosine of x.
`float exp( float x );`	Return e^x.
`float fabs( float x );`	Return | x |.
`float floor( float x );`	Return $\lfloor x \rfloor$.
`float fmod( float x, float y );`	Return r, the remainder of x with respect to y. | r | < | 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 sqrt( float x );`	Return positive square root of x, x >= 0.
`float tanh( float x );`	Return the hyperbolic tangent of x.

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.

`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` `string`s are allocated when needed. However, in the case of any kind of `scanf()` to a `string` or the implied (and hidden) writing to a string with `sprintf()`, the number of characters to be written to the string is unknown until the string has been written. `nab` automatically allocates strings of length 256 to hold such data with the idea that 256 is usually big enough. However, there will be cases where it is not big enough and this will cause the program to die or behave strangely as it will overwrite other data.

Also note that the default precision for floats in *nab* is double precision (see `$NABHOME/src/defreal.h`, since this could be changed, or may be different on your system.) Formats for floats for the *scanf* functions then need to be "%lf" rather than "%f".

The `getline()` function returns a string that has the next line from file `f`. The end-of-line character has been stripped off.

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 Bioplymers

```
molecule   linkprot( string strandname, string seq, string reslib );
molecule   link_na( string strandname, string seq, string reslib,
            string natype, string opts );
int        getseq_from_pdb( string filename, int numstrand,
            string seq[], string strandname[], string type[] );
int        getxyz_from_pdb( string filename, molecule m, string naexp,
             int add_protons );
molecule   getpdb_rlb( string pdbfile, string reslib[], string strandname[],
             string seq[], string type[] );
```

Although many nab functions don't care what kind of molecule they operate on, specific support for proteins is currently somewhat limited. linkprot() takes a strand identifier and a sequence, and returns a molecule with this sequence. The molecule has an extended structure, so that the ϕ, ψ and ω angles are all 180^o. 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 getseq_from_pdb() routine can be used to extract an appropriate single letter sequence from an input pdb file. For each strand in the input file (separated by TER cards) the variable type

returns "protein", "dna" or "rna" depending on how it interpreted the sequence; the strandname returned is the value of the chain-id in the input pdb file. The function return is the number of residues that it could not identify; these are placed in `seq` as "?", and this return value should be zero for a successful invocation.

The `getxyz_from_pdb()` routine will read the pdb file given in `filename`, extract the coordinates, and put them into the corresponding positions in molecule `m`. Atom order within residues in the pdb file does not need to match that in the molecule. `getxyz_from_pdb()` also returns `naexp`, an atom expression string identifying atoms found in the pdb file but not in the molecule. If `add_protons` is not 0, then missing protons are built in plausible (but probably not optimal) positions.

One typical use of these routines would be as follows: *(1)* use `getseq_from_pdb()` to extract the sequence from a pdb file you got from somewhere; *(2)* use `linkprot()` or `link_na()` (or both) to create an `nab` version of the molecule; *(3)* use `getxyz_from_pdb()` to the molecular pdb coordinates into your molecule and to build in missing hydrogens if necessary. These steps are encapsulated in the relatively high-level `getpdb_rlb()` routine. Creating a molecule by `getpdb_rlb()` guarantees that the atoms and their order in each residue will be the same as in the residue library used. Each atom will therefore have a charge, and the resulting molecule should be ready to send to the `leap()` routine, if desired. On the other hand, `getpdb_rlb()` will fail for molecules that contain residues not in the standard residue libraries, whereas `getpdb()` is designed to work on "almost anything." The second argument to `getpdb_rlb` is an array of three strings, giving the resiudue libraries to be used for proteins, DNA and RNA, respectively. The `strandname`, `seq` and `type` arrays are populated on return, with one entry for each strand (or "chain" or "molecule") found in the pdb file. Strands must be separated by TER cards in the input pdb file.

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 [21]. 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

---

21. R. Tan and S. Harvey, "Molecular Mechanics Model of Supercoiled DNA," J. Mol. Biol. **205,** 573-591 (1989).

`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 `dna.amber94.rlb`. 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 using this rule:

$$\text{bond}(atom_i, atom_j) \text{ if } \text{dist}(atom_i, atom_j) < \begin{cases} 1.20 \text{ Angstroms if either atom is a hydrogen} \\ 2.20 \text{ Angstroms if either atom is a sulphur} \\ 1.85 \text{ Angstroms otherwise} \end{cases}$$

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 options. Currently, only a single option is recognized: if *-pqr* is found in the options string, the routine will read in atomic charges and radii immediately following the xyz coordinates (using eight columns for each). Since these columns would ordinarily be used for occupancy and B-factors, the latter are set to 1.0 and 0.0, respectively. Alternatively, if *-pqr* does not appear in the `options` string, occupancies and B-factors are read from the input pdb file, radii are set to default values (see the code for details), and charges are set to zero.

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.

There are several options available for `putpdb`:

Options flags for putpdb

keyword	*meaning*
`-pqr`	Put charges and radii into the columns following the xyz coordinates.
`-nobocc`	Do not put occupancy and b-factor into the columns following the xyz coordinates. than occupancies and charges. 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 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.
`-nocid`	Do not put the chain-id (see the description of *getpdb*, above) in the output (*i.e.* if this flag is present, the chain-id column will be blank).
`-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.

`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 );
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 corrspoinding 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 responsibilty 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 `point`s. 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.* [22] `plane()` 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.

---

22. M.S. Babcock, E.P.D. Pednault, and W.K. Olson, "Nucleic Acid Structure Analysis," J. Mol. Biol. **237,** 125-156 (1994).

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 molcular 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, resdiue res, int datom, int dbond );
int     dumpatom( file f, residue res, int anum, int dbond );

int     assert( condition );
int     debug( expression(s) );
```

`dumpmatrix()` writes the 16 `float` values of `mat` to the file `f`. The matrix is written as four rows of four numbers. `dumpbounds()` writes the distance bounds information contained in `b` to the file `f` using this eight column format:

```
atom-number1  atom-number2  lower  upper
```

If `binary` is set to a non-zero value, equivalent information is writen 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()` statment will evaluate and print a comma-separated expression list along with the source file(s) and line number(s). Continuing the above example, the statment

```
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 );
float      second();
```

The `date()` routine returns a string in the format "03/08/1999", and the `timeofday()` routine returns the current time as "13:45:00". If you need access to more sophisticated time and date functions, the `ftime()` routine is just a wrapper for the standard C routine `strftime`, where the format string is used to determine what is output; see standard C doumentation for how this works.

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

```
t1 = second();
t2 = second();
elapsed = t2 - t1;
```

3.19. nab and AVS.

The `nab` compiler can generate code to convert some `nab` functions into AVS modules. The function type is limited to `int`, `float`, `string` and `molecule`. The function value will be placed on an AVS output port. The function's name must have the form `AVS_`*ident*, where *ident* becomes the

name of the created module. All parameters to the function are either mapped onto AVS widgets or other input and output ports. Details of the mapping are specified using special comments or pragmas with this form:

```
//AVSinfo  parm-class  parm-name  parm-options
```

A detailed description of `nab`'s AVS capabilities is provided in the Chapter "`nab` and AVS".

4. 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 `point`s.

`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]` × `pts[1]`→`pts[3]` and axis-2 is recreated along the vector axis-3 × axis-1. Each of these four functions creates a fixed number of matrices.

Diehedral 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 × `nfold`.

`MAT_cyclic()` creates `cnt` matrices that produce uniform rotations about the axis `pts[1]`→`pts[2]`. The rotations are in multiples of the angle `ang` beginning with 0°, and increasing by `ang` until `cnt` matrices have been created. `cnt` is required to be > 0, but `ang` can be 0, in which case `MAT_cyclic` returns `cnt` copies of the identity matrix.

`MAT_helix()` creates `cnt` matrices that produce a uniform helical twist about the axis `pts[1]`→`pts[2]`. The rotations are in multiples of `ang` and the translations in multiples of `dst`. `cnt` must be > 0, but either `ang` or `dst` or both may be zero. If `ang` is not 0, but `dst` is, `MAT_helix()` produces a uniform plane rotation and is equivalent to `MAT_cyclic()`. An `ang` of 0 and a non-zero `dst` produces matrices that generate a uniform translation along the axis. If both `ang` and `dst` are 0, the `MAT_helix()` creates `cnt` copies of the identity matrix.

The three functions `MAT_orient()`, `MAT_rotate()` and `MAT_translate()` are not really symmetry operations but are auxilliary 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 convered in the section on “Searching Transformation Spaces”. A complete list of keywords, their accepatable values and defaults is shown below.

Keyword	Possible Values	Default Value
`symmetry`	`cube`, `cyclic`, `dihedral`, `dodeca`, `helix`, `ico`, `octa`, `tetra`.	None
`transform`	`orient`, `rotate`, `translate`.	None
`name`	Any string of nonblank characters.	`m`*Pid*
`noid`	`true`, `false`.	`false`
`axestype`	`absolute`, `relative`.	`relative`

`center`
`axis`, `axis1`[1]
`axis2`
`axis3`
	Any three numbers separated by tabs or spaces.	None
		None
		None
		None

`angle`, `angle1`[1]
`angle2`
`angle3`
`dist`
	Any number.	0
		0
		0
		0

`count`	Any integer.	1

1. `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 seaarch 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 0°, 120° and 240° 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 defualt 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. All of this is described in the section "Searching Transformation Spaces".

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 this command

```
matmul 3.mat 5.mat
```

would result in the third matrix of `3.mat` mulitplying 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. 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 structure that satisfy distance constraints is based on a metric matrix or "distance geometry" approach [23,24]. 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:

$$g_{ij} \equiv \mathbf{x}_i \cdot \mathbf{x}_j \tag{1}$$

These matrix elements can be expressed in terms of the distances d_{ij}, d_{i0}, and d_{j0}:

$$g_{ij} = \tfrac{1}{2}\,(d_{i0}^2 + d_{j0}^2 - d_{ij}^2) \tag{2}$$

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_k, the metric matrix can be written in two ways:

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

The first equality follows from the definition of the metric tensor, Eq. (1); the upper limit of three in the second summation reflects the fact that a rank three matrix has only three non-zero eigenvalues. Eq. (3) then provides an expression for the coordinates x_{ik} in terms of the eigenvalues and eigenvectors of the metric matrix:

$$x_{ik} = \lambda_k^{\frac{1}{2}}\, w_{ik} \tag{4}$$

---

23. T.F. Havel, I.D. Kuntz, and G.M. Crippen, “The theory and practice of distance geometry,” Bull. Math. Biol. **45,** 665-720 (1983).

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

If the input distances are not exact, then in general the metric matrix will have more than three non-zero eigenvalues, but an approximate scheme can be made by using Eq. (4) with the three largest eigenvalues. Since information is lost by discarding the remaining eigenvectors, the resulting distances will not agree with the input distances, but will approximate them in a certain optimal fashion. A further "refinement" of these structures in three-dimensional space can then be used to improve agreement with the input distances.

In practice, even approximate distances are not known for most atom pairs; rather, one can set upper and lower bounds on acceptable distances, based on the covalent structure of the protein and on the observed NOE cross peaks. Then particular instances can be generated by choosing (often randomly) distances between the upper and lower bounds, and embedding the resulting metric matrix.

Considerable attention has been paid recently to improving the performance of distance geometry by examining the ways in which the bounds are "smoothed" and by which distances are selected between the bounds [25,26]. 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 [27]. Methods like these are important especially for underconstrainted 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 [28]. 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.

5.2. Creating and manipulating bounds, embeding 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 );
```

25. 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. **56,** 43-78 (1991).

26. J. Kuszewski, M. Nilges, and A.T. Brünger, “Sampling and efficiency of metric matrix distance geometry: A novel partial metrization algorithm,” J. Biomolec. NMR **2,** 33-56 (1992).

27. M.E. Hodsdon, J.W. Ponder, and 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. **264,** 585-602 (1996).

28. B.L. deGroot, D.M.F. van Aalten, R.M. Scheek, A. Amadei, G. Vriend, and H.J.C. Berendsen, “Prediction of protein conformational freedom from distance constraints,” Proteins **29,** 240-251 (1997).

```
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 below.

Option	type	Default	Action
`newbounds()`			
`-rbm`	`string`	None	The value of the option is the name of a file containing the bounds matrix for this molecule. This file would ordinarliy be made by the dumpbounds 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.

The next five routines use atom expressions `aex1` and `aex2` to select two sets of atoms. Each of these four routines returns the number of bounds set or changed. For each pair of atoms (*a1* in `aex1` and *a2* in `aex2`) `andbounds()` sets the lower bound to max (*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 contraints 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 refinments to proceed even when the underlying cause of the inconsistency is not corrected.

For larger systems, the `tsmooth()` routine becomes quite time-consuming as it scales O(N^3). In this case, a more efficient triangle smoothing routine, `geodesics()` is used. `geodesics()` smoothes the bounds matrix via the triangle inequality using a sparse matrix version of a shortest path algorithm.

The `embed` routine takes a bounds object as input, and returns a four-dimensional array of coordinates; (values of the 4-th coordinate may be nearly zero, depending on the value of *k4d*, see below.) Options for how the embed is done are passed in throught the *dg_options* routine, whose option string

has *name=value* pairs, separated by commas or whitespace. Allowed options are listed in the following table.

Options parameters for dg_options

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 non-zero value, use a Gaussian distribution for selecting distances; this will have a mean at the center of the allowed range, and a standard deviation equal to 1/4 of the range. If 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.

Options parameters for dg_options (cont.)

keyword	*default*	*meaning*
`rembed`	0	If set to a non-zero value, use the "random embedding" scheme of de Groot *et al.*, Proteins **29,** 240-251 (1997), rather than metric matrix embedding.
`rbox`	20.0	Size, in Angstroms, of each side of the cubic into which the coordinates are randomly created in the random-embed procedure.
`riter`	1000	Maximum number of cycles for random-embed procedure. Each cycle selects 1000 pairs for adjustment.
`kchi`	1.0	Force constant for enforcement of chirality constraints.
`k4d`	1.0	Force constant for squeezing out the fourth dimensional coordinate. If this is non-zero, a penalty function will be added to the bounds-violation energy, which is equal to 0.5 * *k4d* * *w* * *w*, where *w* is the value of the fourth dimensional coordinate.
`sqviol`	0	If set to non-zero value, use parabolas for the violation energy when upper or lower bounds are violated; otherwise use functions based on those in the `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 %d0, 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.

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[29]. The templates are in PDB format and are located in `$NABHOME/dgdb/template/basepairs/` and `$NABHOME/dgdb/template/stacking/`.

A typical use of these templates would be to set the bounds between two residues to some percentage of the idealized distance described by the template. In this case, the template would be the reference molecule (the second molecule passed to the function). A typical call might be:

```
useboundsfrom(b, m, "1:2,3:??,H?[^'T]", getpdb( PATH + "gc.bdna.pdb" ), "::??,H?[^'T]", 0.1 );
```

where `PATH` is `$NABHOME/dgdb/template/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 and 6.

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*

29. W. Saenger, M. Turcotte, G. Lapalme, and F. Major, "Exploring the conformations of nucleic acids," J. Funct. Program. **5,** 443-460 (1995). Springer-Verlag,

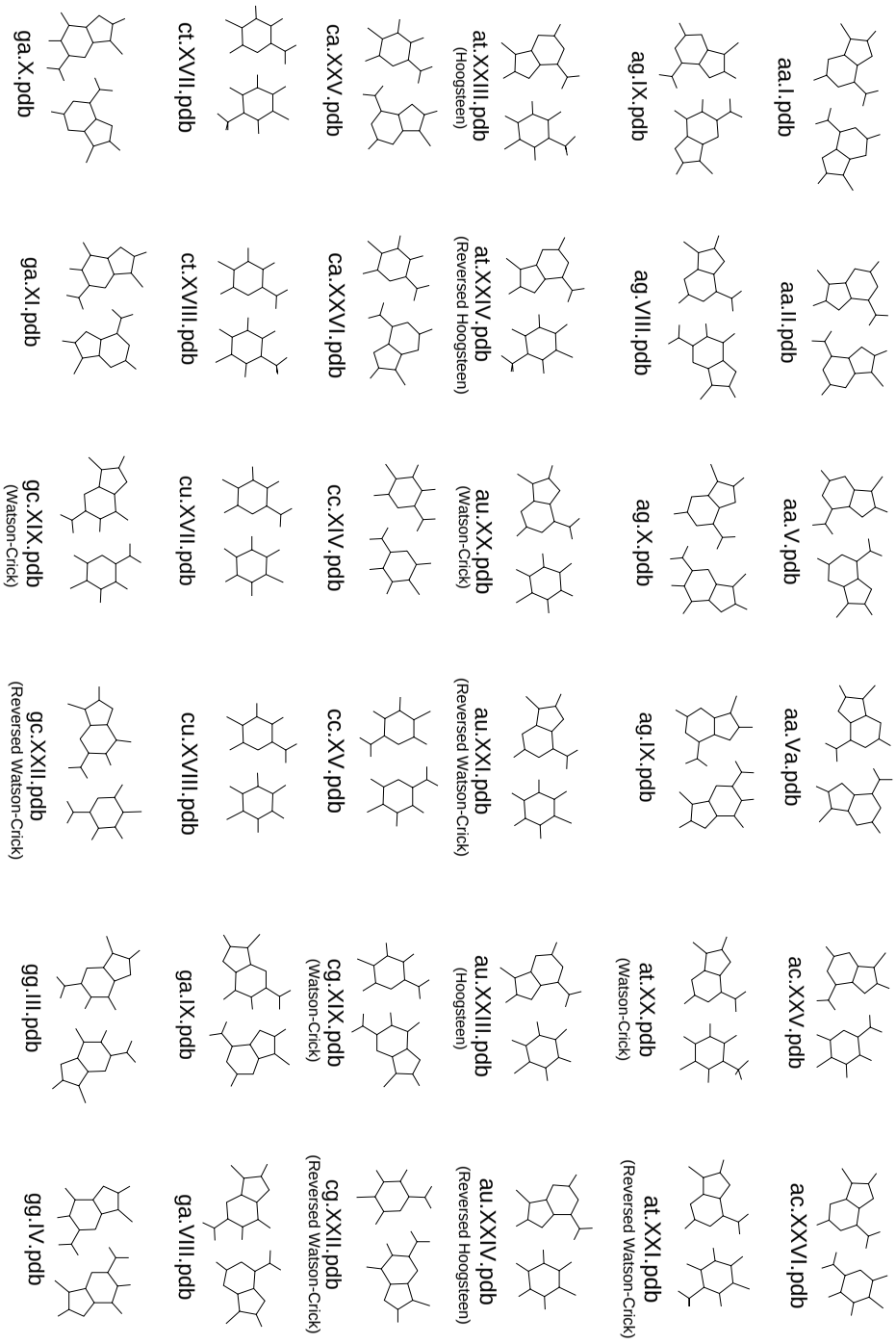


Figure 5. Basepair templates for use with `useboundsfrom()` (aa-gg).

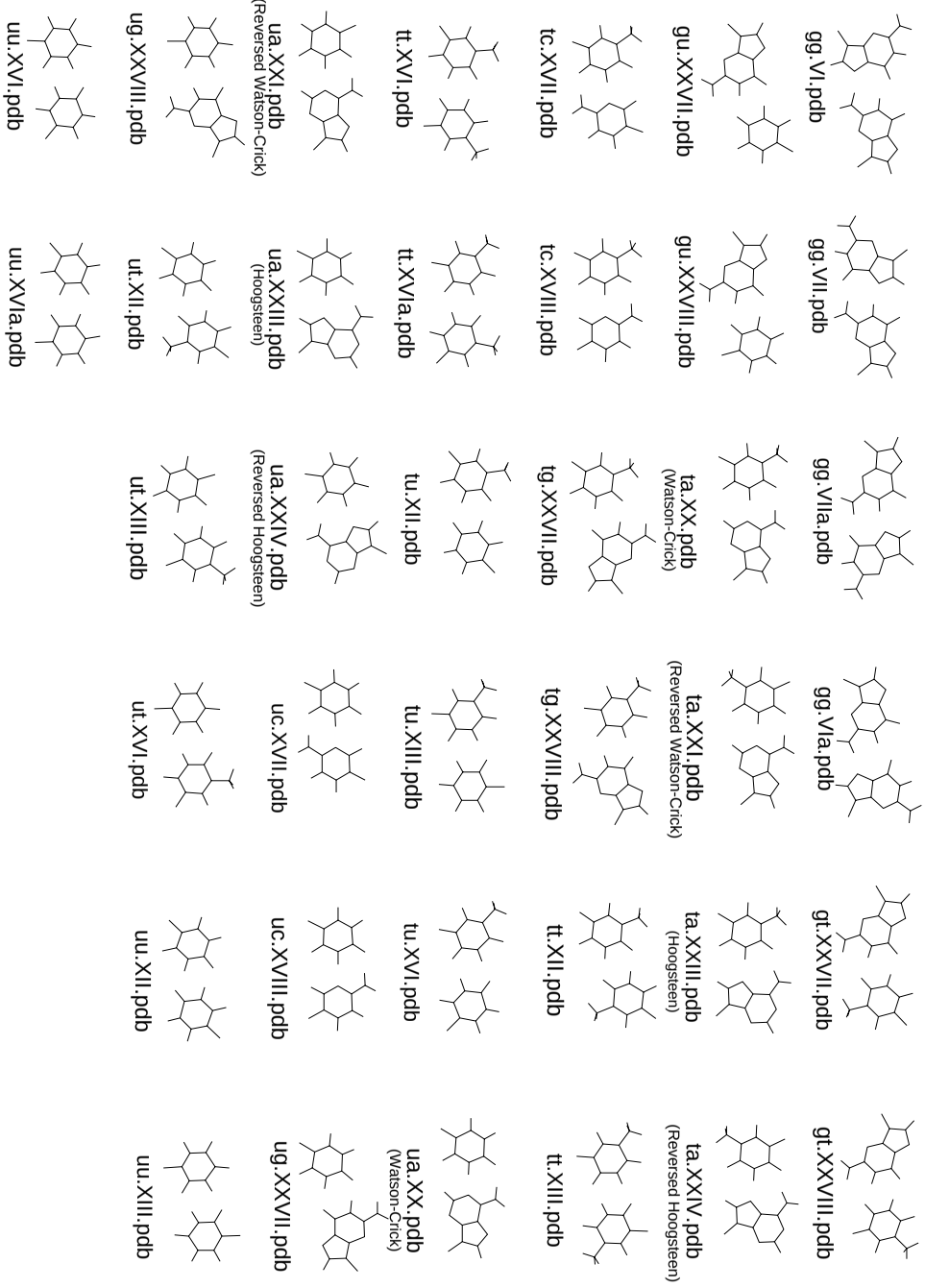


Figure 6. Basepair templates for use with `useboundsfrom()` (gg-uu).

represents the helical parameters of a'rna). The following stacking shemes are included in the `nab` distribution:

```
aa.a-rna.pdb    ca.adna.pdb     ga.adna.pdb     ta.bdna.pdb
aa.adna.pdb     ca.arna.pdb     ga.arna.pdb     ta.cdna.pdb
aa.arna.pdb     ca.bdna.pdb     ga.bdna.pdb     ta.ddna.pdb
aa.bdna.pdb     ca.cdna.pdb     ga.cdna.pdb     ta.tdna.pdb
aa.cdna.pdb     ca.ddna.pdb     ga.ddna.pdb     tc.adna.pdb
aa.ddna.pdb     ca.tdna.pdb     ga.tdna.pdb     tc.bdna.pdb
aa.tdna.pdb     cc.a-rna.pdb    gc.a-rna.pdb    tc.cdna.pdb
ac.a-rna.pdb    cc.adna.pdb     gc.adna.pdb     tc.ddna.pdb
ac.adna.pdb     cc.arna.pdb     gc.arna.pdb     tc.tdna.pdb
ac.arna.pdb     cc.bdna.pdb     gc.bdna.pdb     tg.adna.pdb
ac.bdna.pdb     cc.cdna.pdb     gc.cdna.pdb     tg.bdna.pdb
ac.cdna.pdb     cc.ddna.pdb     gc.ddna.pdb     tg.cdna.pdb
ac.ddna.pdb     cc.tdna.pdb     gc.tdna.pdb     tg.ddna.pdb
ac.tdna.pdb     cg.a-rna.pdb    gc.zdna.pdb     tg.tdna.pdb
ag.a-rna.pdb    cg.adna.pdb     gg.a-rna.pdb    tt.adna.pdb
ag.adna.pdb     cg.arna.pdb     gg.adna.pdb     tt.bdna.pdb
ag.arna.pdb     cg.bdna.pdb     gg.arna.pdb     tt.cdna.pdb
ag.bdna.pdb     cg.cdna.pdb     gg.bdna.pdb     tt.ddna.pdb
ag.cdna.pdb     cg.ddna.pdb     gg.cdna.pdb     tt.tdna.pdb
ag.ddna.pdb     cg.tdna.pdb     gg.ddna.pdb     ua.a-rna.pdb
ag.tdna.pdb     cg.zdna.pdb     gg.tdna.pdb     ua.arna.pdb
at.adna.pdb     ct.adna.pdb     gt.adna.pdb     uc.a-rna.pdb
at.bdna.pdb     ct.bdna.pdb     gt.bdna.pdb     uc.arna.pdb
at.cdna.pdb     ct.cdna.pdb     gt.cdna.pdb     ug.a-rna.pdb
at.ddna.pdb     ct.ddna.pdb     gt.ddna.pdb     ug.arna.pdb
at.tdna.pdb     ct.tdna.pdb     gt.tdna.pdb     uu.a-rna.pdb
au.a-rna.pdb    cu.a-rna.pdb    gu.a-rna.pdb    uu.arna.pdb
au.arna.pdb     cu.arna.pdb     gu.arna.pdb
ca.a-rna.pdb    ga.a-rna.pdb    ta.adna.pdb
```

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, `$NABHOME/dgdb`

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

```
A:C2-T:C1′  424    6.167    0.198    5.687    6.673
A:C2-T:C2   424    3.986    0.175    3.554    4.505
A:C2-T:C2′  424    7.255    0.304    5.967    7.944
A:C2-T:C3′  424    8.349    0.216    7.456    8.897
A:C2-T:C4   424    4.680    0.182    4.122    5.138
A:C2-T:C4′  424    8.222    0.248    7.493    8.800
A:C2-T:C5   424    5.924    0.168    5.414    6.413
A:C2-T:C5′  424    9.385    0.306    8.273   10.104
A:C2-T:C6   424    6.161    0.163    5.689    6.679
A:C2-T:C7   424    7.205    0.184    6.547    7.658
```

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

The databases were constructing using the coordinates from all the known nucleid acid structures from the Nucleic Acid Database (NDB - `http://www.ndbserver.ebi.ac.uk:5700/NDB/`. If one wishes to remake the databases, the coordinates of all the NDB structures should be downloaded and kept in the `$NABHOME/coords` directory. The databases are made by issuing the command `$NABHOME/dgdb/make_databases` *dblist* where *dblist* is a list of nucleic acid types (i.e., bdna, arna, *etc.*). If one wants to add new structures to the structure repository at `$NABHOME/coords`, it is necessary to make sure that the first two letters of the pdb file identify the nucleic acid type. *i.e.*, all bdna pdb files must begin with *bd*.

The `nab` functions used to create the databases are located in `$NABHOME/dgdb/functions`. The stacking databases were constructed as follows: If two residues stacked 5' to 3' in a helix have fewer than ten inter-residue atom distances closer than 2.0Å or larger than 9.0Å, and if the normals between the base planes are less than 20.0°, the residues were considered stacked. The base plane is calculated as the normal to the N1-C4 and midpoint of the C2-N3 and N1-C4 vectors. The first atom expression given to `setboundsfromdb()` specifies the 5' residue and the second atom expression specifies the 3' residue. The source for this function is `getstackdist.nab`.

Similarly, the basepair databases were constructed by measuring the heavy atom distances of corresponding residues in a helix to check for hydrogen bonding. Specifically, if an A-U basepair has an N1-N3 distance of between 2.3Å and 3.2Å and a N6-O4 distance of between 2.3Å and 3.3Å, 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 2.3Å and 3.3Å, the N4-O6 distance is between 2.3Å and 3.2Å, and the O2-N2 distance is between 2.3Å and 3.2Å.

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 three different database types provided with the `nab` distribution are named *nucleic_acid_type.database_type.*`db.` The following databases are included in the distribution:

```
adna.basepair.db
adna.stack.db
adna.nucleotide.db
arna.basepair.db
arna.stack.db
arna.nucleotide.db
bdna.basepair.db
bdna.stack.db
bdna.nucleotide.db
trna.nucleotide.db
trna.stack.db
zdna.basepair.db
zdna.stack.db
zdna.nucleotide.db
```

6. Molecular mechanics and molecular dynamics.

The initial models created by rigid-body transformations or distance geometry are often in need of further refinement, and molecular mechanics and dynamics can often be useful here. nab has facilities to allow molecular mechanics and molecular dynamics calculations to be carried out. At present, this uses the AMBER program LEaP to set up the parameters and topology; the force field calculations and manipulations like minimization and dynamics are done by routines in the nab suite. A version of LEaP is included in the NAB distribution, and is accessed by the leap() discussed below. A later chapter gives a more detailed description.

6.1. Basic molecular mechanics routines

```
int     leap( molecule mol, string commands_1, string commands_2 );
int     readparm( molecule m, string parmfile );

int     mme_init( molecule mol, string aexp, string aexp2,
                  point xyz_ref[], file f );
int     mm_options( string opts );
float   mme( point xyz[], point grad[], int iter );

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 minv[], 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[] );
```

leap() converts an nab molecule into into an AMBER prmtop file. This file is created in the nab's current working directory when leap() is called. The commands_1 string is passed to LEaP, and would typically point to a leaprc file that contained parameter and force field libraries to load. If commands_1 is empty, the all-atom AMBER 94 force field will be used. This string is interpreted by LEaP at the beginning of the run. The commands_2 string is interpreted after the molecule has been read in to a unit called "X". Typically, commands_2 would modify the molecule, say by adding or removing bonds, etc. leap() creates a "parameter-topology" file called prmtop, which typically is read by the readparm routine.

readparm reads an AMBER parameter-topology file, created by leap 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. This routine was written by Bill Ross at the University of California, San Franscisco, and is redistributed with permission.

setxyz_from_mol() copies the atomic coordinates of mol to the array xyz. setmol_from_xyz() replaces the atomic coodinates of mol with the contents of xyz. Both return the

number of atoms copied with a `0` indicating an error occurred.

The *getxv()* and *putxv()* routines read and write Amber-style restart files that have coordinates and velocities.

The `mme_init` function must be called 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 which atoms are to be allowed to move in minimization or dynamics: atoms that do not match `aexp` will have their positions in the gradient vector set to zero. A `NULL` atom expression will allow all atoms to move. The second string, `aexp2` identifies atoms whose positions are to be restrained to the positions in the array `xyz_ref`. The strength of this restraint will be given by the `wcons` variable set in mm_options. A `NULL` value for `aexp2` will cause all atoms to be constrained. The last parameter to `mme_init` is a file pointer for the output trajectory file. This should be NULL if no output file is desired.

`mm_options` is used to set parameters. 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.

Options parameters for mm_options

keyword	*default*	*meaning*
`ntpr`	10	frequency of printing of the energy and its components
`nsnb`	25	frequency at which the non-bonded list is updated
`cut`	8.0	non-bonded cutoff, in Angstroms
`scnb`	2.0	Scaling factor for 1-4 nonbonded interactions; default corresponds to the all-atom Amber force fields
`scee`	1.2	Scaling factor for 1-4 electrostatic interactions. default corresponds to the 1994 and later Amber force fields.
`wcons`	0.0	Restraint weight for keeping atoms close to their positions in `xyz_ref` (see *mme_init*).
`dim`	3	Number of spatial dimensions; supported values are 3 and 4.
`k4d`	1.0	Force constant for squeezing out the fourth dimensional coordinate, if dim=4. If this is non-zero, a penalty function will be added to the bounds-violation energy, which is equal to 0.5 * *k4d* * *w* * *w*, where *w* is the value of the fourth dimensional coordinate.

Options parameters for mm_options (continued)

keyword	*default*	*meaning*
`dt`	0.001	time step, ps.
`t`	0.0	initial time, ps.
`tautp`	0.2	temperature coupling parameter, in ps.
`temp0`	300.	target temperature, K
`vlimit`	20.	maximum absolute value of any component of the velocity vector
`ntpr_md`	10	printing freqeuncy for dynamics information
`zerov`	0	if non-zero, then the initial velocities will be set to zero; otherwise, the values passed into the `md` routine will be used.
`genmass`	10.	The general mass to use for MD if indivudal masses are not read from a prmtop file; value in amu.
`diel`	R	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.
`gb`	0	If set to 1, use the pairwise generalized Born model for solvation. For now, see the code in `sff.c` for details. Set diel to "C" if you use this option.
`gb_debug`	0	If set to 1, print out detailed information about the generalized Born calculations. Only useful for small molecules, since it generates voluminous output.
`epsext`	78.5	Exterior dielectric for generalized Born; interior dielectric is always 1.
`kappa`	0.0	Inverse of the Debye-Huckel length, if gb is turned on, in $Å^{-1}$.

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 and non-bonded updates.

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 funtion 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 dyanmics, using `func` as the force field (this would typcially be set to a function like `mme`.) The number of dynamical variables is given as input parameter `n`: this would be 3 times the number of atoms for ordinary cases, but might be different for other force fields or functions. The arrays `x[], f[]` and `v[]` hold the coordinates, gradient of the potential, and velocities, respectively, and are updated as the simulation progress. The input array `minv[]` must reserve space to hold the inverse of the masses of the particles.

6.2. Typical calling sequences.

The following segment shows some ways in which these routines can be put together to do some molecular mechanics and dynamics:

```
1   // carry out molecular mechanics minimization and some simple dynamics
2   molecule m;
3   int  ier;
4   float  m_xyz[ dynamic ], f_xyz[ dynamic ], v[ dynamic ], minv[ dynamic ] ;
5   float  dgrad, fret, dummy;
6
7   m = bdna( "gcgc" );
8   allocate m_xyz[ 3*m.natoms ];  allocate f_xyz[ 3*m.natoms ];
9   allocate v[ 3*m.natoms ];  allocate minv[ 3*m.natoms ];
10
11  leap( m, "", "" );
12  readparm( m, "prmtop" );
13  setxyz_from_mol( m, NULL, m_xyz );
14
15  mm_options( "cut=25.0, ntpr=10, nsnb=999" );
16  mme_init( m, NULL, "::ZZZ", dummy, NULL );
17  fret = mme( m_xyz, f_xyz, 1 );
18  printf( "Initial energy is %f0, 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*natom, 1000, m_xyz, minv, 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 the parameter topology file is created in line 11, and read back in at line 12. (The reason for separating these is that future runs of the program for the same molecule could omit line 9, and simply read in a pre-existing parameter-topology file.) 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 function value as a float 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.

7. Sample NAB applications.

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.

7.1. 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 Manaul**. 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 `T`s (or their lower case equivalents) into a uniform ideal Watson/Crick B-form DNA duplex. Each basepair has an X-offset of 2.25Å, an inclination of -4.96° and a helical step of 3.38Å rise and 36.0° 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. The nulcleic 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.

7.2. 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 expermental 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.

7.2.1. Refine DNA Backbone Geometry.

As mentioned previously, `wc_helix()` performs rigid body transformations on residues and does not correct for poor backbone geometry. Using distance geometry, several techniques are available to correct the backbone geometry. In program 7, an 8-basepair dna sequence is created using `wc_helix()`. A new bounds object is created on line 14, which automatically sets all the 1-2, 1-3, and 1-4 distance bounds information according the geometry of the model. Since this molecule was created using `wc_helix()`, the O3'-P distance between adjacent stacked residues is often not the optimal 1.595 Å, and hence, the 1-2, 1-3, and 1-4, distance bounds set by `newbounds()` are incorrect. We want to preserve the position of the nucleotide bases, however, since this is the helix whose backbone we wish to minimize. Hence the call to `useboundsfrom()` on line 17 which sets the bounds from every atom in each nucleotide base to the actual distance to every other atom in every other nucleotide base. *In general, the likelihood of a distance geometry refinement to satisfy a given bounds criteria is proportional to the number of (consistent) bounds set supporting that criteria*. In other words, the more bounds that are set supporting a given conformation, the greater the chance that conformation will resolve after the refinement. An example of this concept is the use of `useboundsfrom()` in line 17, which works to preserve our rigid helix conformation of all the nucleotide base atoms.

We can correct the backbone geometry by overwriting the erroneous bounds with more appropriate bounds. In lines 19-29, all the 1-2, 1-3, and 1-4 bounds involving the O3'-P connection between strand 1 residues are set to that which would be appropriate for an idealized phosphate linkage. Similarly, in lines 31-41, all the 1-2, 1-3, and 1-4 bounds involving the O3'-P connection among strand 2 residues are set to an idealized conformation. This technique is effective since all the 1-2, 1-3, and 1-4 distance bounds created by `newbounds()` include those of the idealized nucleotides in the nucleic acid libraries `dna.amber94.rlb`, `rna.amber94.rlb`, *etc.* contained in `reslib`. Hence, by setting these bounds and refining against the distance energy function, we are spreading the 'error' across the backbone, where the 'error' is the departure from the idealized sugar conformation and idealized phospate 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.amber94.rlb", "dna" );
10
11      m = wc_helix( seq, "dna.amber94.rlb", "dna", cseq, "dna.amber94.rlb",
12                    "dna", 2.25, -4.96, 36.0, 4.38, "" );
13
14      b = newbounds(m, "");
15      allocate xyz[ 4*m.natoms ];
16
17      useboundsfrom(b, m, "::??,H?[^T']", m, "::??,H?[^T']", 0.0 );
18      for ( i = 1; i < m.nresidues/2 ; i = i + 1 ){
19             setbounds(b,m, sprintf("1:%d:O3'",i),
20                            sprintf("1:%d:P",i+1),   1.595,1.595);
21             setbounds(b,m, sprintf("1:%d:O3'",i),
22                            sprintf("1:%d:O5'",i+1), 2.469,2.469);
23             setbounds(b,m, sprintf("1:%d:C3'",i),
24                            sprintf("1:%d:P",i+1),   2.609,2.609);
25             setbounds(b,m, sprintf("1:%d:O3'",i),
26                            sprintf("1:%d:O1P",i+1), 2.513,2.513);
27             setbounds(b,m, sprintf("1:%d:O3'",i),
28                            sprintf("1:%d:O2P",i+1), 2.515,2.515);
29             setbounds(b,m, sprintf("1:%d:C4'",i),
30                            sprintf("1:%d:P",i+1),   3.550,4.107);
31             setbounds(b,m, sprintf("1:%d:C2'",i),
32                            sprintf("1:%d:P",i+1),   3.550,4.071);
33             setbounds(b,m, sprintf("1:%d:C3'",i),
34                            sprintf("1:%d:O1P",i+1), 3.050,3.935);
35             setbounds(b,m, sprintf("1:%d:C3'",i),
36                            sprintf("1:%d:O2P",i+1), 3.050,4.004);
37             setbounds(b,m, sprintf("1:%d:C3'",i),
38                            sprintf("1:%d:O5'",i+1), 3.050,3.859);
39             setbounds(b,m, sprintf("1:%d:O3'",i),
40                            sprintf("1:%d:C5'",i+1), 3.050,3.943);
41
42             setbounds(b,m, sprintf("2:%d:P",i+1),
43                            sprintf("2:%d:O3'",i),   1.595,1.595);
44             setbounds(b,m, sprintf("2:%d:O5'",i+1),
45                            sprintf("2:%d:O3'",i),   2.469,2.469);
46             setbounds(b,m, sprintf("2:%d:P",i+1),
47                            sprintf("2:%d:C3'",i),   2.609,2.609);
48             setbounds(b,m, sprintf("2:%d:O1P",i+1),
```

```
49                          sprintf("2:%d:O3'",i),   2.513,2.513);
50          setbounds(b,m, sprintf("2:%d:O2P",i+1),
51                          sprintf("2:%d:O3'",i),   2.515,2.515);
52          setbounds(b,m, sprintf("2:%d:P",i+1),
53                          sprintf("2:%d:C4'",i),   3.550,4.107);
54          setbounds(b,m, sprintf("2:%d:P",i+1),
55                          sprintf("2:%d:C2'",i),   3.550,4.071);
56          setbounds(b,m, sprintf("2:%d:O1P",i+1),
57                          sprintf("2:%d:C3'",i),   3.050,3.935);
58          setbounds(b,m, sprintf("2:%d:O2P",i+1),
59                          sprintf("2:%d:C3'",i),   3.050,4.004);
60          setbounds(b,m, sprintf("2:%d:O5'",i+1),
61                          sprintf("2:%d:C3'",i),   3.050,3.859);
62          setbounds(b,m, sprintf("2:%d:C5'",i+1),
63                          sprintf("2:%d:O3'",i),   3.050,3.943);
64      }
65      tsmooth( b, 0.0005 );
66      dg_options(b, "seed=33333, gdist=0, ntpr=100, k4d=4.0" );
67      setxyzw_from_mol( m, NULL, xyz );
68      conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 500 );
69      setmol_from_xyzw( m, NULL, xyz );
70      putpdb( "acgtacgt.pdb", m );
```

The approach of Program 7 is effective but has a disadvantage in that it does not scale linearly with the number of atoms in the molecule. In particular, `tsmooth()` and `conjgrad()` require extensive CPU cycles for large numbers of residues. For this reason, the function `dg_helix()` was created. `dg_helix()` takes uses the same method of Program 7, but employs a 3-basepair helix template which traverses the new helix as it is being constructed. In this way, the helix is built in a piecewise manner and the maximum number of residues considered in each refinement is less than or equal to six. This is the preferred method of helix construction for large, idealized canonical duplexes.

7.2.2. 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 teriary 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 −1 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 4. 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 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 -

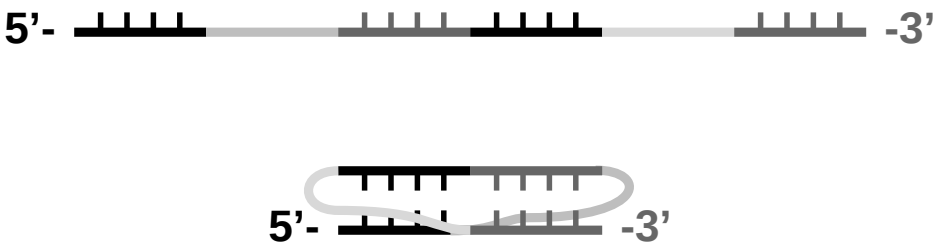


Figure 4. Single stranded RNA *(top)* folded into a pseudoknot *(bottom). The black and dark gray base pairs can be stacked.*

only connectivity and general base-base interactions. The stacking and base-pair interactions here are derived from NMR evidence, but in other cases might arise from other sorts of experiments, or as a model hypothesis to be tested.

The 20-base RNA sequence is defined on line 9. The molecule itself is created with the `link_na()` function call which creates an extended conformation of the RNA sequence and caps the 5’ and 3’ ends. Lines 15-18 define arrays that will be used in the simulated annealing of the structure. The bounds object is created in line 20 which automatically sets the 1-2, 1-3, and 1-4 distance bounds in the molecule. The loop in lines 22-25 sets the bounds of each atom in each residue base to the actual distance to every other atom in the same base. This has the effect of enforcing the planarity of the base by treating the base somewhat like a rigid body. In lines 27-45, bounds are set according to information stored in a database. The `setboundsfromdb()` call sets the bounds from all the atoms in the two specified residues to a *1.0* multiple of the standard deviation of the bounds distances in the specified database. Specifically, line 27 sets the bounds between the base atoms of the first and second residues of strand 1 to be within one standard deviation of a *typical* aRNA stacked pair. Similarly, line 39 sets the bounds between residues 1 and 13 to be that of *typical* Watson-Crick basepairs. For a description of the `setboundsfromdb()` function, see Chapter 1.

Line 47 smooths the bounds matrix, by attempting to adjust any sets of bounds that violate the triangle equality. Lines 49-50 initialize some distance geometry variables by setting the random number generator seed, declaring the type of distance distribution, how often to print the energy refinement process, declaring the penalty for using a 4th dimension in refinement, and which atoms to use to form the initial metric matrix. The coordinates are calculated and embedded into a 3D coordinate array, *xyz* by the `embed()` function call on line 51.

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

```
1     // Program 8 - create a pseudoknot using distance geometry
2     molecule m;
3     float   xyz[ dynamic ],minv[ dynamic ],f[ dynamic ],v[ dynamic ];
4     bounds  b;
```

```
5       int     i, seqlen;
6       float   fret;
7       string  seq, opt;
8
9       seq = "gcggaaacgccgcguaagcg";
10
11      seqlen = length(seq);
12
13      m = link_na("1", seq, "rna.amber94.rlb", "rna", "35");
14
15      allocate xyz[ 4*m.natoms ];
16      allocate minv[ 4*m.natoms ];
17      allocate f[ 4*m.natoms ];
18      allocate v[ 4*m.natoms ];
19
20      b = newbounds(m, "");
21
22      for ( i = 1; i <= seqlen; i = i + 1) {
23              useboundsfrom(b, m, sprintf("1:%d:??,H?[^'T]", i), m,
24              sprintf("1:%d:??,H?[^'T]", i), 0.0 );
25      }
26
27      setboundsfromdb(b, m, "1:1:",  "1:2:",  "arna.stack.db",    1.0);
28      setboundsfromdb(b, m, "1:2:",  "1:3:",  "arna.stack.db",    1.0);
29      setboundsfromdb(b, m, "1:3:",  "1:18:", "arna.stack.db",    1.0);
30      setboundsfromdb(b, m, "1:18:", "1:19:", "arna.stack.db",    1.0);
31      setboundsfromdb(b, m, "1:19:", "1:20:", "arna.stack.db",    1.0);
32
33      setboundsfromdb(b, m, "1:8:",  "1:9:",  "arna.stack.db",    1.0);
34      setboundsfromdb(b, m, "1:9:",  "1:10:", "arna.stack.db",    1.0);
35      setboundsfromdb(b, m, "1:10:", "1:11:", "arna.stack.db",    1.0);
36      setboundsfromdb(b, m, "1:11:", "1:12:", "arna.stack.db",    1.0);
37      setboundsfromdb(b, m, "1:12:", "1:13:", "arna.stack.db",    1.0);
38
39      setboundsfromdb(b, m, "1:1:",  "1:13:", "arna.basepair.db", 1.0);
40      setboundsfromdb(b, m, "1:2:",  "1:12:", "arna.basepair.db", 1.0);
41      setboundsfromdb(b, m, "1:3:",  "1:11:", "arna.basepair.db", 1.0);
42
43      setboundsfromdb(b, m, "1:8:",  "1:20:", "arna.basepair.db", 1.0);
44      setboundsfromdb(b, m, "1:9:",  "1:19:", "arna.basepair.db", 1.0);
45      setboundsfromdb(b, m, "1:10:", "1:18:", "arna.basepair.db", 1.0);
46
47      tsmooth(b, 0.0005);
48
49      opt = "seed=571, gdist=0, ntpr=50, k4d=2.0, randpair=5.";
50      dg_options( b, opt );
51      embed(b, xyz );
```

```
52
53      for ( i = 3000; i > 2800; i = i - 100 ){
54           conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.1, 10., 500 );
55
56           dg_options( b, "ntpr=1000, k4d=0.2" );
57           mm_options( "ntpr_md=50, zerov=1, temp0=" +sprintf("%d.",i));
58           md( 4*m.natoms, 1000, xyz, minv, f, v, db_viol );
59
60           dg_options( b, "ntpr=1000, k4d=4.0" );
61           mm_options( "zerov=0, temp0=0., tautp=0.3" );
62           md( 4*m.natoms, 8000, xyz, minv, f, v, db_viol );
63      }
64
65      setmol_from_xyzw( m, NULL, xyz );
66      putpdb( "pseudoknot.pdb", m );
```

The resulting structure of Program 8 is shown in Figure 5. This structure had an final total energy of 9.41 units. The helical region, shown as polytubes, shows stacking and wc-pairing interactions and a well-defined right-handed helical twist. Of course, good modeling of a "real" pseudoknot would require putting in more constraints, but this example should illustrate how to get started on problems like this.

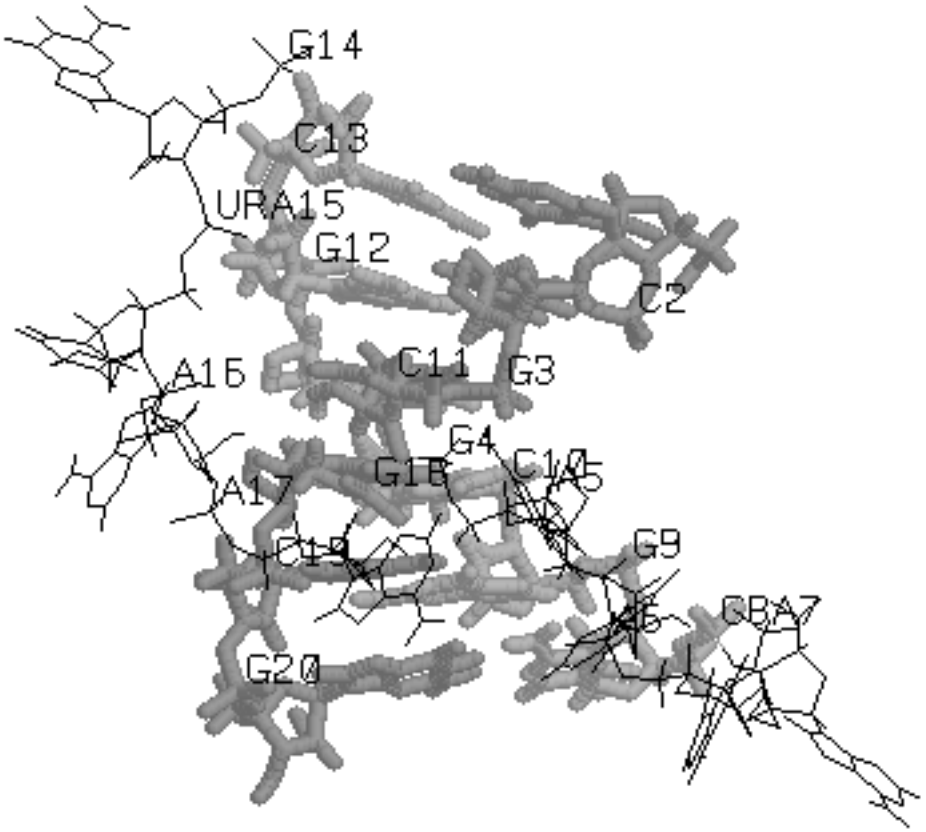


Figure 5. 20-base example RNA pseudoknot

7.2.3. 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 flexibile 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 lenghts 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
2   //   on proteins, with DYANA-like distance restraints.
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 ], minv[ 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";
75          aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
76          aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
77          aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
78          setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
79      }
```

```
80    for( i=36; i<=39; i++){
81        aex1 = ":" + sprintf( "%d", i ) + ":CA";
82        aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
83        aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
84        aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
85        setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
86    }
87    for( i=88; i<=89; i++){
88        aex1 = ":" + sprintf( "%d", i ) + ":CA";
89        aex2 = ":" + sprintf( "%d", i+1 ) + ":CA";
90        aex3 = ":" + sprintf( "%d", i+2 ) + ":CA";
91        aex4 = ":" + sprintf( "%d", i+3 ) + ":CA";
92        setchivol( b, m, aex1, aex2, aex3, aex4, 7.0 );
93    }
94
95    // set up some options for the distance geometry calculation
96    //  here use the random embed method:
97    dgopts = "ntpr=10000,rembed=1,rbox=300.,riter=250000,seed=8511135";
98    dg_options( b, dgopts );
99
100   // do triangle-smoothing on the bounds matrix, then embed:
101   geodesics( b ); embed( b, xyz );
102
103   // now do conjugate-gradient minimization on the resulting structures:
104
105   // first, weight the chirality constraints heavily:
106   dg_options( b, "ntpr=20, k4d=5.0, sqviol=0, kchi=50." );
107   conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.02, 1000., 300 );
108
109   // next, squeeze out the fourth dimension, and increase penalties for
110   //    distance violations:
111   dg_options( b, "k4d=10.0, sqviol=1, kchi=50." );
112   conjgrad( xyz, 4*m.natoms, fret, db_viol, 0.02, 100., 400 );
113
114   // transfer the coordinates from the "xyz" array to the molecule
115   // itself, and print out the violations:
116   setmol_from_xyzw( m, NULL, xyz );
117   dumpboundsviolations( stdout, b, 0.5 );
118
119   // do a final short molecular-mechanics "clean-up":
120   setxyz_from_mol( m, NULL, xyz );
121   leap( m, "", "" );
122   readparm( m, "prmtop" );
123
124   mm_options( "cut=10.0" );
125   mme_init( m, NULL, "::ZZZ", xyz, NULL );
126   conjgrad( xyz, 3*m.natoms, fret, mme, 0.02, 100., 200 );
```

```
127     setmol_from_xyz( m, NULL, xyz );
128     putpdb( argv[3] + ".mm.pdb", m );
```

Once the covalent bounds are created, the the bounds matrix is modified by constraints constructed from an NMR analysis program. This particular example uses the format of the DYANA program, but NAB could be easily modified to read in other formats as well. Here are a few lines from the *mrf2.7col* file:

```
  1 ARG+  QB      2 ALA   QB       7.0
  4 GLU-  HA     93 LYS+  QB       7.0
  5 GLN   QB      8 LEU   QQD      9.9
  5 GLN   HA      9 VAL   QQG      6.4
 85 ILE   HA     92 ILE   QD1      6.0
  5 GLN   HN      1 ARG+  O        2.0
  5 GLN   N       1 ARG+  O        3.0
  6 ALA   HN      2 ALA   O        2.0
  6 ALA   N       2 ALA   O        3.0
```

The format should be self-explanatory, with the final number giving the upper bound. Code in lines 31-69 reads these in, and translates pseudo-atom codes like "QQD" into atom names. Lines 71-93 add in chirality constraints to ensure right-handed alpha-helices: distance constraints alone do not distinguish chirality, so additions like this are often necessary. The "actual" distance geometry steps take place in line 101, first by triangle-smoothing the bounds, then by embedding them into a three-dimensional object. The structures at this point are actually generally quite bad, so "real-space" refinenement is carried out in lines 103-112, and a final short molecular mechanics minimzation 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 futher refinement in Amber yields good structures.

7.3. 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.

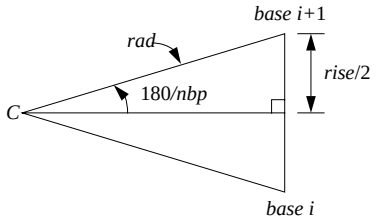
However, this method is inadequate for the more complicated curves that can be handled by example 3. Here each base is placed on the curve so that its helical axis is aligned correctly, but its helical orientation with respect to the previous base is arbitrary. It is then rotated about its helical axis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

7.4. Closed Circular DNA.

This section describes how to use nab to make closed circular duplex DNA with a uniform rise of 3.38Å. 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 Δ*linking number* which is the number of extra turns to add or substract. If the original circle had *nbp*/10 turns, the supercoiled circle will have $npb/10 + \Delta lk$ turns. As each turn represents 360° of twist and there are *nbp* base pairs, the twist between base pairs is:

$$(nbp/10 + \Delta lk) \times 360/nbp$$

At this point, we are ready to create models of circular DNA. Bases are added to model in three stages. Each base pair is created using the `nab` builtin `wc_helix()`. It is originally in the XY plane with its center at the origin. This makes it convenient to create the DNA circle in the XZ plane. After the base pair has been created, it is rotated around its own helical axis to give it the proper twist, translated along the global X axis to the point where its center intersects the circle and finally rotated about the Y axis to move it to its final location. Since the first base pair would be both twisted about Z and rotated about Y 0°, those steps are skipped for base one. A detailed description follows the code.

```
1     // Program 9 - Create closed circular DNA.
2     #define RISE    3.38
3
4     int     b, nbp, dlk;
5     float       rad, twist, ttw;
6     molecule    m, 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 ] );
13        exit( 1 );
14    }
15
16    nbp = atoi( argv[ 2 ] );
17    if( !nbp || nbp % 10 ){
18        fprintf( stderr,
19        "%s: Num. of base pairs must be multiple of 10\n",
20            argv[ 1 ] );
```

```
21          exit( 1 );
22      }
23
24      dlk = atoi( argv[ 3 ] );
25
26      twist = ( nbp / 10 + dlk ) * 360.0 / nbp;
27      rad = 0.5 * RISE / sin( 180.0 / nbp );
28
29      matdx = newtransform( rad, 0.0, 0.0, 0.0, 0.0, 0.0 );
30
31      m = newmolecule();
32      addstrand( m, "A" );
33      addstrand( m, "B" );
34      ttw = 0.0;
35      for( b = 1; b <= nbp; b = b + 1 ){
36
37          getbase( b, sbase, abase );
38
39          m1 = wc_helix(
40              sbase, "dna.amber94.rlb", "dna", abase, "dna.amber94.rlb",
41              "dna", 2.25, -4.96, 0.0, 0.0 );
42
43          if( b > 1 ){
44              mattw = newtransform( 0.,0.,0.,0.,0.,ttw );
45              transformmol( mattw, m1, NULL );
46          }
47
48          transformmol( matdx, m1, NULL );
49
50          if( b > 1 ){
51              matry = newtransform(
52                  0.,0.,0.,0.,-360.*(b-1)/nbp,0. );
53              transformmol( matry, m1, NULL );
54          }
55
56          mergestr( m, "A", "last", m1, "sense", "first" );
57          mergestr( m, "B", "first", m1, "anti", "last" );
58          if( b > 1 ){
59              connectres( m, "A", b - 1, "O3'", b, "P" );
60              connectres( m, "B", 1, "O3'", 2, "P" );
61          }
62
63          ttw = ttw + twist;
64          if( ttw >= 360.0 )
65              ttw = ttw - 360.0;
66      }
67
```

```
68      connectres( m, "A", nbp, "O3'", 1, "P" );
69      connectres( m, "B", nbp, "O3'", 1, "P" );
70
71      putpdb( "circ.pdb", m );
72      putbnd( "circ.bnd", m );
```

The code requires two integer arguments which specify the number of base pairs and the Δ*linking number* or the amount of supercoiling. Lines 11-24 process the arguments making sure that they conform to the model's assumptions. In lines 11-14, the code checks that there are exactly three arguments (the `nab` program's name is argument one), and exits with a error message if the number of arguments is different. Next lines 16-22 set the number of base pairs (`nbp`) and test to make certain it is a nonzero multiple of 10, again exiting with an error message if it is not. Finally the Δ*linking number* (`dlk`) is set in line 24. The helical twist and circle radius are computed in lines 26 and 27 in accordance with the formulas developed above. Line 29 creates a transformation matrix, `matdx`, that is used to move each base from the global origin along the X-axis to the point where its center intersects the circle.

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

One or three transformations is required to position this base in its correct place in the circle. It must be rotated about the Z-axis (its helical axis) so that it is one additional unit of twist beyond the previous base. This twist is done in lines 43-46. Since the first base needs 0° twist, this step is skipped for it. In line 48, the base pair is moved in the positive direction along the X-axis to place the base pair's origin on the circle. Finally, the base pair is rotated about the Y-axis in lines 50-54 to bring it to its proper position on the circle. Again, since this rotation is 0° for base 1, this step is also skipped for the first base.

In lines 56-57, the newly positioned base pair in `m1` is added to the growing molecule in `m`. Note that since the two strands of DNA are antiparallel, the `"sense"` strand of `m1` is added after the last base of the `"A"` strand of `m` and the `"anti"` strand of `m1` is added before the first base of the `"B"` strand of `m`. For all but the first base, the newly added residues are bonded to the residues they follow (or precede). This is done by the two calls to `connectres()` in lines 59-60. Again, due to the antiparallel nature of DNA, the new residue in the `"A"` strand is residue `b`, but is residue `1` in the `"B"` strand. In line 63-65, the total twist (`ttw`) is updated and adjusted to keep in in the range [0,360). After all base pairs have been added the loop exits.

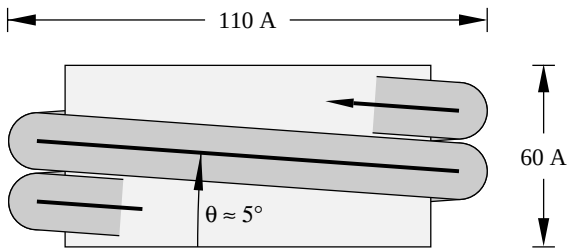
After the loop exit, since this is a *closed* circular molecule the first and last bases of each strand must be bonded and this is done with the two calls to `connectres()` in lines 67-68. The last step is to save the molecule's coordinates and connectivity in lines 71-72. The `nab` builtin `putpdb()` writes the coordinate information in PDB format to the file `"circ.pdb"` and the `nab` builtin `putbnd()` saves the bonding as pairs of integers, one pair/line in the file `"circ.bnd"`, where each integer in a pair refers to an `ATOM` record in the previously written PDB file.

7.5. 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 ÅÅ.

The nucleosome core fragment [30] is composed of duplex DNA wound in a left handed helix around a cental 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:

$$d = 3.38\sin(\theta); \quad \phi = 360t/(n-1) \qquad \text{(\_\&dy)}$$

$$r = \frac{3.38(n-1)\cos(\theta)}{2\pi t} \qquad \text{(\_\&rad)}$$

where *d* and ϕ are the displacement along and rotation about the protein core axis for each base pair.

These relationships are easily derived. Let the nucleosome core particle be oriented so that its helical axis is along the global Y-axis and the lower cap of the protein core is in the XZ plane.

30. B. Lewin, in *Genes IV*, (Cell Press, Cambridge, Mass., 1990). pp. 409-425.

Consider the circle that is the projection of the helical axis of the DNA duplex onto the XZ plane. As the duplex spirals along the core particle it will go around the circle t times, for a total rotation of $360t°$. The duplex contains $n-1$ steps, resulting $360t/(n-1)°$ of rotation between successive base pairs.

```
1   // Program 10.  Create simple nucleosome model.
2   #define PI  3.141593
3   #define RISE    3.38
4   #define TWIST   36.0
5   int         b, nbp; int getbase();
6   float       nt, theta, phi, rad, dy, ttw, len, plen, side;
7   molecule    m, m1;
8   matrix      matdx, matrx, maty, matry, mattw;
9   string      sbase, abase;
10
11  nt = atof( argv[ 2 ] );     // number of turns
12  nbp = atoi( argv[ 3 ] );    // number of base pairs
13  theta = atof( argv[ 4 ] ); // winding angle
14
15  dy = RISE * sin( theta );
16  phi = 360.0 * nt / ( nbp-1 );
17  rad = (( nbp-1 )*RISE*cos( theta ))/( 2*PI*nt );
18
19  matdx = newtransform( rad, 0.0, 0.0, 0.0, 0.0, 0.0 );
20  matrx = newtransform( 0.0, 0.0, 0.0, -theta, 0.0, 0.0 );
21
22  m = newmolecule();
23  addstrand( m, "A" ); addstrand( m, "B" );
24  ttw = 0.0;
25  for( b = 1; b <= nbp; b = b + 1 ){
26      getbase( b, sbase, abase );
27      m1 = wc_helix( sbase, "", "dna", abase, "", "dna",
28          2.25, -4.96, 0.0, 0.0 );
29      mattw = newtransform( 0., 0., 0., 0., 0., ttw );
30      transformmol( mattw, m1, NULL );
31      transformmol( matrx, m1, NULL );
32      transformmol( matdx, m1, NULL );
33      maty = newtransform( 0.,dy*(b-1),0., 0.,-phi*(b-1),0.);
34      transformmol( maty, m1, NULL );
35
36      mergestr( m, "A", "last", m1, "sense", "first" );
37      mergestr( m, "B", "first", m1, "anti", "last" );
38      if( b > 1 ){
39          connectres( m, "A", b - 1, "O3'", b, "P" );
40          connectres( m, "B", 1, "O3'", 2, "P" );
41      }
42      ttw += TWIST; if( ttw >= 360.0 ) ttw -= 360.0;
43  }
44  putpdb( "nuc.pdb", m );
```

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

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` buitin `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()`.

7.6. "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 include that 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 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 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.

7.6.1. 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

$$p_1, p_2, \cdots, p_n$$

and interpolates it to produce a new set of points

$$np_1, np_2, \cdots, np_m$$

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()` [31] 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 requirment 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

---

31. W. H. Press, S. A. Teukolsky, W. T. Vettering, and B. P. Flannery, in *Numerical Recipes in C*, (Cambridge, New York, 1992). pp. 113-117.

interpolated values are computed using one or more calls to `splint()`.

What is unusual about this interpolation is that the points at which the interpolation is to be performed are unknown. Instead, these points are chosen so that the distance between each point and its successor is the constant value `RISE`, set here to 3.38 which is the rise of an ideal B-DNA duplex. Thus, we have to search for the points and most of the code is devoted to doing this search. The details follow the listing.

```
1     // Program 11 - Build DNA along a curve
2     #define RISE    3.38
3
4     #define EPS 1e-3
5     #define APPROX(a,b) (fabs((a)-(b))<=EPS)
6     #define MAXI    20
7
8     #define MAXPTS  150
9     int npts;
10    float   a[ MAXPTS ];
11    float   x[ MAXPTS ], y[ MAXPTS ], z[ MAXPTS ];
12    float   x2[ MAXPTS ], y2[ MAXPTS ], z2[ MAXPTS ];
13    float   tmp[ MAXPTS ];
14
15    string  line;
16
17    int i, li, ni;
18    float   dx, dy, dz;
19    float   la, lx, ly, lz, na, nx, ny, nz;
20    float   d, tfrac, frac;
21
22    int spline();
23    int splint();
24
25    for( npts = 0; line = getline( stdin ); ){
26        npts = npts + 1;
27        a[ npts ] = npts;
28        sscanf( line, "%lf %lf %lf",
29            x[ npts ], y[ npts ], z[ npts ] );
30    }
31
32    spline( a, x, npts, 1e30, 1e30, x2, tmp );
33    spline( a, y, npts, 1e30, 1e30, y2, tmp );
34    spline( a, z, npts, 1e30, 1e30, z2, tmp );
35
36    li = 1; la = 1.0; lx = x[1]; ly = y[1]; lz = z[1];
37    printf( "%8.3f %8.3f %8.3f\n", lx, ly, lz );
38
```

```
39      while( li < npts ){
40          ni = li + 1;
41          na = a[ ni ];
42          nx = x[ ni ]; ny = y[ ni ]; nz = z[ ni ];
43          dx = nx - lx; dy = ny - ly; dz = nz - lz;
44          d = sqrt( dx*dx + dy*dy + dz*dz );
45          if( d > RISE ){
46              tfrac = frac = .5;
47              for( i = 1; i <= MAXI; i = i + 1 ){
48                  na = la + tfrac * ( a[ni] - la );
49                  splint( a, x, x2, npts, na, nx );
50                  splint( a, y, y2, npts, na, ny );
51                  splint( a, z, z2, npts, na, nz );
52                  dx = nx - lx; dy = ny - ly; dz = nz - lz;
53                  d = sqrt( dx*dx + dy*dy + dz*dz );
54                  frac = 0.5 * frac;
55                  if( APPROX( d, RISE ) )
56                      break;
57                  else if( d > RISE )
58                      tfrac = tfrac - frac;
59                  else if( d < RISE )
60                      tfrac = tfrac + frac;
61              }
62              printf( "%8.3f %8.3f %8.3f\n", nx, ny, nz );
63          }else if( d < RISE ){
64              li = ni;
65              continue;
66          }else if( d == RISE ){
67              printf( "%8.3f %8.3f %8.3f\n", nx, ny, nz );
68              li = ni;
69          }
70          la = na;
71          lx = nx; ly = ny; lz = nz;
72      }
```

Execution begins in line 25 where the points are read from `stdin` one point or three numbers/line and stored in the three arrays `x`, `y` and `z`. The independent variable for each spline, stored in the array `a` is created at this time holding the numbers `1` to `npts`. The second derivatives for the three splines, one each for interpolation along the X, Y and Z directions are computed in lines 32-34. Each call to `spline()` has two arguments set to `1e30` which indicates that the second derivative values should be 0 at the first and last points of the table. The first point of the interpolated set is set to the first point of the original set and written to `stdout` in lines 36-37.

The search that finds the new points is lines 39-72. To see how it works consider the figure below. The dots marked p_1, p_2, ... , p_n correspond to the orginal 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.

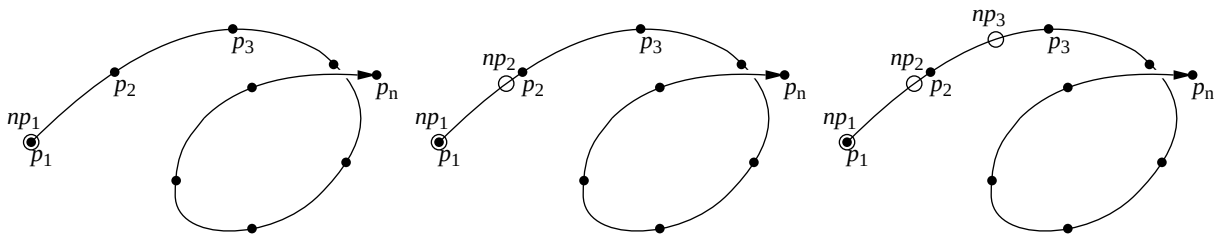
The search works by first finding a point of the original table that is at least RISE distance from the last point found. If the last point of the original table is not far enough from the last point found, the search loop exits and the program ends. However, if the search does find a point in the original 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.



7.6.2. Driver Code.

This section describes the main routine or driver of the second program which is the actual DNA bender. This routine reads in the points, then calls `putdna()` (described in the next section) to place base pairs at each point. The points are either read from `stdin` or from the file whose name is the second command line argument. The source of the points is determined in lines 8-18, being `stdin` if the command line contained a single arguments or in the second argument if it was present. If the argument count was greater than two, the program prints an error message and exits. The points are read in the loop in lines 20-26. Any line with a # in column 1 is a comment and is ignored. All other lines are assumed to contain three numbers which are extracted from the string, `line` and stored in the point array `pts` by the `nab` builtin `sscanf()` (lines 23-24). The number of points is kept in `npts`. Once all points have been read, the loop exits and the point file is closed if it is not `stdin`. Finally, the points are passed to the function `putdna()` which will place a base pair at each point and save the coordinates and connectivity of the resulting molecule in the pair of files `dna.path.pdb` and `dna.path.bnd`.

```
1    // Program 12 - DNA bender main program
2    string      line;
3    file        pf;
4    int     npts;
5    point       pts[ 5000 ];
6    int     putdna();
7
8    if( argc == 1 )
9        pf = stdin;
10   else if( argc > 2 ){
11       fprintf( stderr, "usage: %s [ path-file ]\n",
12           argv[ 1 ], argv[ 2 ] );
13       exit( 1 );
14   }else if( !( pf = fopen( argv[ 2 ], "r" ) ) ){
15       fprintf( stderr, "%s: can't open %s\n",
16           argv[ 1 ], argv[ 2 ] );
17       exit( 1 );
18   }
19
20   for( npts = 0; line = getline( pf ); ){
```

```
21          if( substr( line, 1, 1 ) != "#" ){
22              npts = npts + 1;
23              sscanf( line, "%lf %lf %lf",
24                  pts[ npts ].x, pts[ npts ].y, pts[ npts ].z );
25          }
26      }
27
28      if( pf != stdin )
29          fclose( pf );
30
31      putdna( "dna.path", pts, npts );
```

7.6.3. Wrap DNA.

Every `nab` molecule contains a frame, a moveable handle that can be used to position the molecule. A frame consists of three othogonal 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 orthorgonal unit vectors remain untouched. While this may at first glance seem odd, it makes possible the following three stage process of setting the molecule's frame, aligning that frame on the *global* frame, then transforming the molecule with respect to the global axes and origin which provides a convenient way to position and orient a molecule's frame at arbitrary points in space. With all this in mind, here is the source to `putdna()` which bends a B-DNA duplex about an open space curve.

```
1   // Program 13 - place base pairs on a curve.
2   point       s_ax[ 4 ];
3   int     getbase();
4
5   int putdna( string mname, point pts[ 1 ], int npts )
6   {
7       int p;
8       float   tw;
9       residue r;
10      molecule    m, m_path, m_ax, m_bp;
11      point   p1, p2, p3, p4;
12      string  sbase, abase;
13      string  aex;
14      matrix  mat;
15
16      m_ax = newmolecule();
17      addstrand( m_ax, "A" );
18      r = getresidue( "AXS", "axes.rlb" );
19      addresidue( m_ax, "A", r );
20      setxyz_from_mol( m_ax, NULL, s_ax );
21
22      m_path = newmolecule();
23      addstrand( m_path, "A" );
24
25      m = newmolecule();
26      addstrand( m, "A" );
27      addstrand( m, "B" );
28
29      for( p = 1; p < npts; p = p + 1 ){
30          setmol_from_xyz( m_ax, NULL, s_ax );
31          setframe( 1, m_ax,
32              "::ORG", "::ORG", "::SXT", "::ORG", "::CYT" );
33          axis2frame( m_path, pts[ p ], pts[ p + 1 ] );
34          alignframe( m_ax, m_path );
35          mergestr( m_path, "A", "last", m_ax, "A", "first" );
36          if( p > 1 ){
37              setpoint( m_path, sprintf( "A:%d:CYT",p-1 ), p1 );
38              setpoint( m_path, sprintf( "A:%d:ORG",p-1 ), p2 );
39              setpoint( m_path, sprintf( "A:%d:ORG",p ), p3 );
40              setpoint( m_path, sprintf( "A:%d:CYT",p ), p4 );
41              tw = 36.0 - torsionp( p1, p2, p3, p4 );
42              mat = rot4p( p2, p3, tw );
43              aex = sprintf( ":%d:", p );
44              transformmol( mat, m_path, aex );
45              setpoint( m_path, sprintf( "A:%d:ORG",p ), p1 );
46              setpoint( m_path, sprintf( "A:%d:SXT",p ), p2 );
47              setpoint( m_path, sprintf( "A:%d:CYT",p ), p3 );
```

```
48                  setframep( 1, m_path, p1, p1, p2, p1, p3 );
49              }
50
51              getbase( p, sbase, abase );
52              m_bp = wc_helix( sbase, "dna.amber94.rlb", "dna",
53                  abase, "dna.amber94.rlb", "dna",
54                  2.25, -5.0, 0.0, 0.0 );
55              alignframe( m_bp, m_path );
56              mergestr( m, "A", "last", m_bp, "sense", "first" );
57              mergestr( m, "B", "first", m_bp, "anti", "last" );
58              if( p >  1 ){
59                  connectres( m, "A", p - 1, "O3'", p, "P" );
60                  connectres( m, "B", 2, "P", 1, "O3'" );
61              }
62          }
63
64          putpdb( mname + ".pdb", m );
65          putbnd( mname + ".bnd", m );
66      };
```

`putdna()` takes three arguments—`name`, a string that will be used to name the PDB and bond files that hold the bent duplex, `pts` an array of points containing the origin of each base pair and `npts` the number of points in the array. `putdna()` uses four molecules. `m_ax` holds a small artificial molecule containing four atoms that is a proxy for the some of the frame's used placing the base pairs. The molecule `m_path` will eventually hold one copy of `m_ax` for each point in the input array. The molecule `m_bp` holds each base pair after it is created by `wc_helix()` and `m` will eventually hold the bent dna. Once again the function `getbase()` (to be defined by the user) provides the mapping between the current point (`p`) and the nucleotides required in the base pair at that point.

Execution of `putdna()` begins in line 16 with the creation of `m_ax`. This molecule is given one strand `"A"`, into which is added one copy of the special residue `AXS` from the standard `nab` residue library `"axes.rlb"` (lines 17-19). This residue contains four atoms named `ORG`, `SXT`, `CYT` and `NZT`. These atoms are placed so that `ORG` is at (0,0,0) and `SXT`, `CYT` and `NZT` are 1 Å 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 iniital 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 1Å 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 orienation of its X and Y vectors is arbitary. While this is 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 36.0°. The amount of rotation required to give it the correct twist is computed in line 41. A transformation matrix that will rotate the new AXS residue about the ORG—ORG axis by this amount is created in line 42, the atom expression that names the AXS residue is created in line 43 and the residue rotated in line 44. Once the new residue is given the correct twist the frame m_path is moved to the new residue in lines 45-48.

The base pair is added in lines 51-60. The user defined function getbase() converts the point nubmer (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 orientaton using by aligning its frame on the frame of m_path that we hvae just created (line 55). The new pair is merged into m and bonded with the previous base pair if it exits. After the loop exits, the bend DNA duplex coordinates are save as PDB and it connectivity as a bnd file in the calls to putdpb() and putbnd() in lines 64-65, whereupon putdna() returns to the caller.

7.7. Building peptides

The next 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.

```
// Program  14 -- build a peptide sequence

// "peptide" is an nab program that will generate a pdb file given a structure
// type and a sequence.  It was created by Paul Beroza.

// The command line syntax for peptide is:

// % peptide  structure sequence pdbout [ -lib libfile ]

// where "structure" defines the type of structure to be created and "sequence"
// is a string o of 1 letter amino acid codes.  For example:

// % peptide ALPHA  AAAAA aaaa.pdb
```

```
// will create and alanine pentapeptide in an alpha helical structure.

// The structure definitions are stored in a library file that can be specified
// on the command line  (the "-lib libfile" option), or by default is in
// $NABHOME/reslib/conf.lib.

// I've included a sample library  "conf.lib" This file looks like:

// --------------------
// ALPHA  1     alpha helix
// phi    -57.0 psi    -47.0 omega  180.0

// ABETA 1      anti-parallel beta sheet
// phi    -139.0 psi    135.0 omega  -178.0
//      .
//      .
//      .etc.
// --------------------

// The file contains sets of definitions, one for each structure type.  The
// definitions above are separated by a blank line, but that is not necessary.
// Each time peptide finds a line that begins with an alphanumeric character,
// it initializes a new structure type with the first string in the line as its
// identifying string.  The <structure> on the command line must match one of
// the structure types in the "conf.lib" file.

// The next field on the structure type line is the number of residues in the
// structure.  The following lines must contain the phi psi and omega values
// for each of the residues in the structure type.  The angles may be in any
// order, but the string defining the angle must precede its floating point
// value.

// If the number of residues = 1, it is a special structure for which the phi
// psi and omega values are the same for all residues in the structure.  For
// these structure types, the <sequence> may be of any length.  For other
// structure types, the number of residues in <sequence> must agree with the
// number of residues in the corresponding structure type in the "conf.lib"
// file.  The resulting pdb file is written to standard out.

// Please let me know of any bugs or suggestions.

// Enjoy,

// Paul Beroza  <pberoza@info.combichem.com>


#define MAXRES 500
#define USAGE "Usage: %s structure_type sequence pdbout <-lib XXX>0, argv[1]
```

```
int fix_angles( molecule m1, int i, int nr, float omega, float psi, float phi)
{

        //atom expressions to rotate about angles:
        string omega_string, psi_string, phi_string;

        //atom expressions for backbone atoms:
        string npos, cpos, capos, nm1pos, cm1pos, cam1pos;

        point n_xyz, ca_xyz, c_xyz; //coords for res i bb
        point cm1_xyz; //coords for res i - 1 bb
        point u, v, zax, p_head, p_tail;
        point           va, vb, vc;
        float           a0, rot_angle, phi0, psi0, omega0;
        atom            a;
        int             ii;
        matrix          mat;

        if (i > nr) nr = i;
        omega_string = sprintf(":%d-%d:", i, nr);
        psi_string = sprintf(":%d:O|:%d-%d:", i - 1, i, nr);
        phi_string = sprintf(":%d:C*,O*,?[A-Z]*|:%d-%d:*", i, i + 1, nr);
        npos = sprintf(":%d:N", i);
        cpos = sprintf(":%d:C", i);
        capos = sprintf(":%d:CA", i);
        cm1pos = sprintf(":%d:C", i - 1);
        cam1pos = sprintf(":%d:CA", i - 1);
        nm1pos = sprintf(":%d:N", i - 1);

        //create z - axis for rotation to get
        // C(i - 1) - N(i) - CA(i) bond angle = 121.9;

        setpoint(m1, npos, n_xyz);
        setpoint(m1, capos, ca_xyz);
        setpoint(m1, cpos, c_xyz);
        setpoint(m1, cm1pos, cm1_xyz);

        u = ca_xyz - n_xyz;
        v = cm1_xyz - n_xyz;
        zax = u ^ v;

        a0 = angle(m1, cm1pos, npos, capos);
        rot_angle = 121.9 - a0;

        p_tail = n_xyz;
        p_head = n_xyz + zax;

        mat = rot4p(p_head, p_tail, rot_angle);
```

```
        transformmol(mat, m1, omega_string);

        psi0 = torsion(m1, nm1pos, cam1pos, cm1pos, npos);
        rot_angle = psi - psi0;
        mat = rot4(m1, cam1pos, cm1pos, rot_angle);
        transformmol(mat, m1, psi_string);

        omega0 = torsion(m1, cam1pos, cm1pos, npos, capos);
        rot_angle = omega - omega0;
        mat = rot4(m1, cm1pos, npos, rot_angle);
        transformmol(mat, m1, omega_string);

        phi0 = torsion(m1, cm1pos, npos, capos, cpos);
        rot_angle = phi - phi0;
        mat = rot4(m1, npos, capos, rot_angle);
        transformmol(mat, m1, phi_string);

        return 0;
};

#define MAXTEMPLATES 50

int match_template(file f, float phi[1], float psi[1], float omega[1],
        string struct_type, int nres)
{
        string          line;
        int             ir, template_nres, ntemp, found;
        string          ttype, template_name[MAXTEMPLATES];
        string          s1, s2, s3;
        float           f1, f2, f3;
        string          ftmp;

        found = 0;
        ntemp = 0;
        while (line = getline(f)) {
                sscanf(line, "%s %d", ttype, template_nres);
                if (ttype == "")
                        continue;
                if (template_nres < 1) {
                        fprintf(stderr, "template has no residues0);
                        exit(0);
                }
                ++ntemp;
                template_name[ntemp] = ttype;
                if (ttype != struct_type) {
                for (ir = 1; ir <= template_nres; ir++)
                        line = getline(f);
                        continue;
```

```
                }
                found = 1;
                if (template_nres != 1 && template_nres != nres) {
                fprintf(stderr, "template has %d atoms and sequence has %d0,
                        template_nres, nres);
                        exit(0);
                }
                for (ir = 1; ir <= template_nres; ir++) {
                        line = getline(f);
                        sscanf(line, "%s %lf %s %lf %s %lf", s1, f1, s2, f2, s3, f3);
                        if (s1 == "phi") phi[ir] = f1;
                        else if (s1 == "psi") psi[ir] = f1;
                        else if (s1 == "omega") omega[ir] = f1;

                        if (s2 == "phi") phi[ir] = f2;
                        else if (s2 == "psi") psi[ir] = f2;
                        else if (s2 == "omega") omega[ir] = f2;

                        if (s3 == "phi") phi[ir] = f3;
                        else if (s3 == "psi") psi[ir] = f3;
                        else if (s3 == "omega") omega[ir] = f3;
                }

        //template_nres == 1 is a special case for which all
        // residues in the sequence adopt the 1 triplet of phi / psi / omega values

               if (template_nres == 1) {
                        for (ir = 2; ir <= nres; ir++) {
                                phi[ir] = phi[1];
                                psi[ir] = psi[1];
                                omega[ir] = omega[1];
                        }
                }
                break;
        }
        if (!found) {
                fprintf(stderr, "template not found0);
                fprintf(stderr, "must be one of:");
                for (ir = 1; ir <= ntemp; ++ir)
                        fprintf(stderr, " %s", template_name[ir]);
                fprintf(stderr, "0);
                exit(0);
        }
        return 0;
};

//main routine: process the input, then call the above routines
```

```
int             ir, nr;
string          seq, struct_type;
molecule        m1;
float           omega[MAXRES], psi[MAXRES], phi[MAXRES];
point           ax, center;
atom            a;
file            conformation_file;
string          outfile;
int             ac;

if (argc != 4 && argc != 6) {
        fprintf(stderr, USAGE);
        exit(1);
}
if (argc > 4) {
        if (argv[5] != "-lib") {
                fprintf(stderr, USAGE);
                exit(1);
        }
        conformation_file = fopen(argv[6], "r");
        if (conformation_file == NULL) {
        fprintf(stderr, "conformation file not found %s0, argv[6]);
                exit(1);
        }
} else {
        conformation_file = fopen(getenv("NABHOME") + "/reslib/conf.lib", "r");
        if (conformation_file == NULL) {
                fprintf(stderr, "conformation file not found %s0,
                        getenv("NABHOME") + "/reslib/conf.lib" );
                exit(1);
        }
}

struct_type = sprintf("%s", argv[2]);
seq = sprintf("%s", argv[3]);
nr = length(seq);
outfile = argv[4];

if (nr > MAXRES) {
        fprintf(stderr, "MAXRES exceeded0);
        exit(0);
}

//get the needed phi, psi and omega values from a template:
match_template(conformation_file, phi, psi, omega, struct_type, nr);

//generate a structure in the extended conformation:
m1 = linkprot("new", seq, "");
```

```
//adjust the phi, psi, and omega angles:
for (ir = 2; ir <= nr; ++ir){
        fix_angles(m1, ir, nr, omega[ir], psi[ir - 1], phi[ir]);
}

putpdb(outfile, m1);
```

8. NAB and AVS.

8.1. Introduction.

`nab` encourages users to build models of structure families defined by one or more parameters. Unfortunately, while a family's parameters may be obvious, their interaction and limits are often difficult to fully grasp. The usual approach to this situation is to create sufficient instances of the family to sample the parameter space and to view the results with a molecular graphics system. `nab` offers an alternative. In conjunction with the AVS graphics environment, `nab` can convert a standalone `nab` program into an AVS module. Parameters to the former program will automatically be connected to AVS widgets and/or ports allowing real time interactive viewing of their effects and interactions.

This capability was originally intended as a quick way to visualize the interactions of a model's parameters; however, `nab`'s AVS capabilities have been extended to permit it to perform a variety of data flow molecular calculations. Subject to some limitations, `nab` can generate AVS modules that can read and/or write `int`, `float`, `string` and `molecule` values to and/or from a network. Parameters may be mapped onto widgets or directly onto ports.

8.2. AVS.

AVS is a program that allows users to create "visualization networks". It does this by providing an environment called the "Network Editor" which is used to connect elements of a library of standard visualization and data manipulation tools called modules. A module is incorporated into a network by dragging its icon from the appropriate module library menu into the Network Editor's work space. Each module has one or more "ports" represented as small colored bars on either the top and/or bottom edge(s) of its rectangular icon. Colored bars on the top of a module's icon are input ports which can accept data from other modules in the network. Colored bars on the bottom of the icon are output ports which are used to send data created or modified by this module to other modules in the network. A port's colors represent the type of its data. A port is connected by moving the mouse onto it and pressing the middle button. The Network Editor will draw thin lines between the port and all other ports to which it can be connected. To select a connection, the user continues to hold down the middle button and moves the mouse onto the desired connection. Once the connection is established, the Network Editor replaces the thin line by a thick line and the mouse button is released. The general rule involving connections is that any input port may be connected to any output port as long as their colors match.

AVS divides modules into four classes depending on their role in a network. Modules in different classes are placed into separate Module Library Menus in the Network Editor's "Resource Area". Modules that introduce data into a network are called "Data Input" modules. Modules that accept data, operate on it and send it on are called "Filters" if the output data has the same general type as the input data or "Mappers" if they are different. Modules that terminate a data path are called "Data Output" modules. This classification is both artificial and somewhat arbitrary. It is artificial in that the Flow Executive which runs the modules does not distinguish between module classes. And it can be arbitrary because sometimes a module can be used in more than one role in a network. However, since most modules do fit this classification, having a separate Module Library Menu for each class simplifies finding the right module for the task at hand.

AVS offers a very high level 3-D graphics application programming interface (API) through its geometry type. It provides a standard module called the "Geometry Viewer" which accepts connections from other modules in the network that have geometry outputs. It renders the structures that are sent over these connections. The Geometry Viewer is a complete 3-D viewing system. Its most important capabilities include the ability to select and position objects, to control their surface

properties and rendering levels (sticks, flat shading, etc), and to set the number, color and location of lights.

AVS supports several types of connections of which `nab` uses only a few. These are the simple types that contain a single `int`, `float` or `string` value and the field type which is used to represent mathematical fields—functions that have a value at every point in some space. These values can be scalars or vectors. The vector length is arbitrary but all vectors in a particular field must have the same length. Also all the values of a field including components of the vectors must have the same simple type, for example `float`. The dimensionality of the space can be 1-, 2- or 3-D and the mapping of the field values to points in space can be implicit or explicit. Thus the AVS field type serves as a generalized array. `nab` also uses the geometry type as most `nab` generated modules create or modify molecules which are eventually converted into AVS geometry and displayed. However, this use is indirect as `nab` represents molecules as fields and uses other modules, notably `mv102` which is discussed in the Appendices to convert these fields into geometry that can be viewed.

AVS provides a large set of standard modules, but they can not provide for every possible application. They do offer a chemistry type, but it was designed for quantum chemistry and is not suitable for macromolecules. However, it is relatively easy for users to design and implement their own "custom" modules. AVS provides a "Module Generator" for producing a new module's skeleton. This skeleton includes code that creates both ports for the module's data and widgets for its parameters. The stylized nature of the code required to create modules made it easy to give `nab` this capability.

8.3. nab Extensions for Defining Modules.

The conversion of an `nab` function into an AVS module is straightforward, but it does require that additional information about the function be made available to the the `nab` compiler. The compiler needs to know which function is to be converted into a module and which of its parameters are to be mapped onto ports and which onto widgets. There are two possible ways to present this information to the compiler. One would be to extend the grammar with additional productions that would only be used for module creation. This approach was rejected as the number of new productions would be considerable. In addition, AVS is licensed separately from `nab` and a site without an AVS distribution would be unable to properly create modules.

The method used by `nab` to convert a function into a module uses three things. The function's name must have the form `AVS_`*ident*, where *ident* is an identifier—a letter followed by zero or more letters, digits and underscores. A special comment line describing each of that function's parameters is required. And in order to activate any AVS module creation, the `nab` source must be compiled with the `-avs` option. The comments that contain the compiler directives that are used to convert a function into a module have the following forms. Items in italics stand for general instances of things that depend on the function being converted.

`//AVSinfo`	`port`	*pname*	*direction*	Map parameter onto a port.
`//AVSinfo`	`parm`	*pname*	*options*	Map parameter onto a widget.
`//AVSinfo`	`send`	*pname*	*properties*	Molecule properties to send.
`//AVSinfo`	`free`	*pname*		Free space allocated to *pname*.

direction is one or both of the words `in` or `out`. A value of `in` maps the parameter onto an input port and reads its value from it. A value of `out` maps the parameter onto an output port and writes its value to it. A port may be both `in` and `out` in which case, the value is read from the input port and written back to the output port. The two strings `in out` and `out in` are equivalent. `nab` variables that correspond to widgets or input only ports are read only and can not appear on the left hand side of

an assignment statement.

The value of *options* depends on the type of the parameter. For `int` and `float` parameters, *options*, if present, is a triple of numbers specifying the parameter's default, minimum and maximum values. If *options* is not given, the default, minimum and maximum values are set to `0`, `-10000` and `10000`. For a `string` parameter, *options*, if present consists of the string's default value. If the default value includes white space, it must be enclosed in double quotes (`"`). If *options* is absent, the default value is `NULL`.

The `send` directive applies only to `molecule` parameters mapped onto output ports or the return value of a `molecule` function. It tells the `nab` compiler which of the molecule's atomic properties are to be sent along with its coordinates. *properties* is one or more of the following words in any order: `charge`, `radius`, `float1` or `float2`. The word `all` can be used to send all of a `molecule`'s properties. If *pname* has the special value `return-value`, this `send` applies to the function's return value.

The `free` directive tells the `nab` compiler to free the `nab` variable *pname* after the module executes. This directive applies only to `string` and `molecule` variables. Storage allocated to module variables without a `free` directive is lost after each module execution. Again if *pname* has the value `return-value` the function's return value will be freed after module execution. Users should always include `free` directives for those `string` and `molecule` variables that can be freed after execution since the amount of storage lost over a large number of module executions can be enough to cause the program to run out of memory and abort. `nab` can not automatically generate free directives because it can not always tell if two variables point to the same data. The most common example of this would be in a module that reads a molecule from the network and modifies it, then returns the modified molecule. In this case the input molecule parameter and the function return value point to the same data. If the module wrapper automatically freed all data after use, it would successfully free the returned molecule the first time, and then fail when it tried to free it again.

8.4. How nab Creates Modules.

`nab` converts a function into a module by generating a "wrapper"—additional code invisible at the `nab` level—that sets up and calls the original function. The wrapper does the following things. It registers the function and its description with the AVS Network Editor. This allows the Network Editor to create the function's module icon and to determine the types of data it can send and receive. The wrapper sets up the module's ports and widgets. It copies data from the input ports and widgets into the function's parameters. If the function executes successfully, it copies any output parameters and the function's return value to the output ports. The wrapper also performs any required conversions between `nab` data types and their AVS equivalents.

8.4.1. Molecule Fields.

The choice of an AVS representation for `nab` molecules was limited to those AVS types that support objects with internal structure. These are the field, UCD (for "Unstructured Cell Data"), geometry and chemistry types. The chemistry type was designed for quantum calculations and is not suitable for macromolecules. Both the geometry and UCD types are powerful enough to represent molecules, but the implementation would be somewhat opaque. This left the field type. The AVS limitation that all the data in a field have the same fundamental type meant that an `nab` molecule required at least three fields as it contains a mixture of `int`, `float` and `string` data organized into a three level hierarchy of strands, residues and atoms.

The current implementation which is not entirely satisfactory uses three fields. A byte field containing one entry of 16 bytes/atom carries names and some hierarchy information. Each entry is organized as follows. The atom's name starts at byte 0, takes 1 to 4 bytes and is terminated with a zero byte. The name of the residue that contains this atom begins at byte 5, takes 1 to 4 bytes and is terminated with a zero byte. Finally the number of the residue that contains this atom represented as a character string (`%d`) starts at byte 10, takes 1 to 5 places and is terminated by a zero byte. Residues are numbered consecutively from `0` beginning with the first residue of the first strand in the `nab` molecule. The residues of the second and subsequent strands follow in the order that the strands were created with `addstrand()`. A float field with one entry/atom carries the atom's coordinates and a user selected subset of its properties. The third field is an integer field that carries the molecule's bonding information, one entry/bond, where the two elements of an entry refer to atoms in the byte and float fields. The implementation which was motivated by the available macromolecular viewer `mv102` reduces `nab`'s three level molecular hierarchy of strands, residues and atoms to a two level hierarchy of residues and atoms. The table below shows how the various fields are declared.

Component	AVS Field Type
Names	`1-D 1-space 1-vector uniform byte`
Bonds	`1-D 1-space 2-vector uniform integer`
Atoms	`1-D 1-space uniform integer`

8.4.2. Implementation Details.

To understand how `nab` converts a function into a module requires a short description of the basic AVS module and how it works. A module is a standalone program that is executed under control of the AVS Network Editor's "Flow Executive". Every module must contain two subroutines, called the "description procedure" and the "compute procedure". The description procedure is a function that makes calls to the AVS runtime library to describe the module's ports and parameters and identifies its compute procedure. When a module is loaded into a "Module Library", the Flow Executive runs its description procedure. This registers the module with the Flow Executive which builds the new module's icon, and enters its port and parameter requirements into the Flow Executive's internal data base.

A module's compute procedure is what actually does the module's work. When the module is inserted into a network and it becomes active because new data has been presented to its input ports or its parameter widgets have been changed, the Flow Executive runs the compute procedure. This procedure in turns calls whatever user code is required to perform the module's task. In the case of an `nab` generated module, it calls the `nab` function. AVS expects a module's compute procedure to return a `0` if it fails and a non-zero value if it succeeds. When the compute procedure fails, AVS aborts execution of the network without sending the module's data downstream.

8.4.3. Limitations of nab created AVS modules.

The `nab` module generator is still in the early phase of its development, and it contains several implementation restrictions, some of which will be removed as the development continues. These limitations are: 1) the loss of information when an `nab` molecule is converted into its AVS representation; 2) the limitation of parameters to scalars; 3) the inability to send and receive upstream geometry data; 4) the blocking of the entire AVS system if the module attempts to read from `stdin`, and 5) the requirement that a function return value of `0` indicates the module failed aborting network execution.

Restrictions 1-3 will be removed in Version 1.2 of `nab`. The information that is lost in the translation from an `nab` molecule the current AVS representation will be placed in a fixed length block that

precedes the molecule's names in its byte field. The mapping of array parameters onto fields is straightforward except in the case `string` and `molecule` data, where AVS's field requirement of uniform vector length results in wasted space. Accepting upstream geometry data requires defining a mapping onto standard `nab` constructs and extending the wrapper to perform this translation.

Restrictions 4 and 5 are harder to remove. Since an AVS module operates as a child process of the Flow Executive, the module inherits its three standard file descriptors. These are generally attached to the tty that AVS was started in and successfully reading from `stdin` is very difficult. Writing to either `stdout` or `stderr` is possible as long as neither has been redirected. The last restriction, that of using a `0` to indicate module failure can not be removed without modifying the semantics of an `nab` function to allow it to return *two* values, the actual function value and and an indicator that this value is valid, which is impossible since functions by definition return a single value.

8.5. Examples of nab Created Modules.

`nab` classifies a module by what it does with `molecule` data. Modules that only create `molecule` data are Data Input, modules that read and write `molecule` data are Filters, and modules that only use `molecule` data are Data Output. `nab` does not create `Mapper` modules. Any `nab` module that does not involve `molecule` data is a Data Input module.

8.5.1. Data Input Modules.

`nab` has been used to create numerous Data Input modules involving both nucleic acids and proteins. Two of them are discussed in some detail below. The first is a DNA Duplex Generator that was the very first `nab` generated module. The second is a DNA Bender that shows the power of this approach.

8.5.1.1. DNA Duplex Generator.

This module creates models of uniform DNA duplexes of Watson/Crick base pairs. The inputs are the sequence of one strand and four numbers that define the duplex's X-offset, inclination, twist and rise. Duplex creation requires two steps. The `nab` builtin function `wc_complement()` creates a string that represents the complement of the input sequence. Then the input string, the newly created complement string and the four helical parameters are sent to the `nab` builtin `wc_helix()` which converts them into the desired duplex which in turn is returned as the value of the function `AVS_dna()` and displayed by the AVS Geometry Viewer.

The function includes seven `//AVSinfo` directives and is called `AVS_dna()`. The name of the module it generates is `dna`. Each of the four float parameters will be mapped onto an AVS dial widget and the directives limit the ranges of the dials from `-10,000`- `10,000` to something more appropriate. The parameter `seq` has no default. The two `free` directives (lines 8-9) cause the `nab` to free the space that holds the input sequence and returned molecule after module execution.

```
1    // AVS_dna() - AVS Watson/Crick duplex generator
2
3    //AVSinfo   parm    seq
4    //AVSinfo   parm    xoff    2.25 -5 10
5    //AVSinfo   parm    incl    -4.96 -20 30
6    //AVSinfo   parm    twist   36.0 20 45
```

```
7     //AVSinfo   parm    rise    3.38 2 4.5
8     //AVSinfo   free    seq
9     //AVSinfo   free    return-value
10    molecule    AVS_dna( string seq,
11        float xoff, float incl, float twist, float rise )
12    {
13        string cseq;
14        molecule m;
15
16        cseq = wc_complement( seq, "dna.amber94.rlb", "dna" );
17
18        m = wc_helix( seq, "dna.amber94.rlb", "dna",
19            cseq, "dna.amber94.rlb", "dna",
20            xoff, incl, twist, rise );
21        return( m );
22    };
```

8.5.1.2. DNA Bender.

There are many times when it is necessary to deform a piece of duplex DNA. It might need to be "unwound" in order to insert an intercalator between two base pairs or it might need to be "bent" to see to align the grooves on one side of the duplex with "ridges" on another molecule. The traditional way of doing this is to select and interactively change the relevant torsion angles. Unfortunately, due the complexity of the DNA backbone, several torsion angles may need to be changed in a concerted fashion to achieve the desired base positions. And to make things even worse, these torsion angle movements will not move the complementary strand which is only hydrogen bonded to the strand being bent. It would be much simpler if the user could insert a "hinge" between adjacent base pairs of the original duplex and then move one side of the duplex (both strands) without moving the other side. The `nab` module discussed in this section does just that.

The DNA bender module is a fairly long `nab` program. However it not very complex. The first half of the code—the function `AVS_dnabender()`—creates a strandard B-form duplex with the desired sequence, uses its `base` parameter to move the "hinge" or bending site, and then uses the other six parameters to change the position and/or orientation of the selected half of the duplex. The second half of the DNA bender—the function `putaxes()`—is used to create a coordinate frame that is placed at the bending site so the user can predict the effect of translation or rotation. This coordinate frame is the one defined by the Watson/Crick pair at the bending site and remains associated with that base pair as it is transformed.

The first time the DNA bender executes, it creates the molecule with the specified sequence. Subsequent executions either change the hinge point or translate or rotate a portion of the duplex about a one of the axes at the bending site. Each time the module executes a transformation, it *changes* the coordinates of the DNA. Thus the effects of the sequence of transformations accumulate in the molecule's coordinates. This requires that the molecule continue to exist when the module is inactive and that the module's first execution be distinguished from the others. Both requirements are met by using a global variable to hold the molecule (line 3). `nab` global variables exist throughout program execution, or in the case of an `nab` generated AVS module, for the entire time the module is connected in a network. Since all `nab` global variables are initialized to `0`, a `0` or `NULL` value of `m` can be used to indicate the module's first execution.

The `if` statement in lines 30-54 detects the module's first execution. It creates the molecule by using `wc_complement()` to create the string representing the the complementary strand followed by a call to `wc_helix()` to create a standard B-form DNA duplex. The number of residues in the duplex is saved to `nres`. Lines 36-38 translate the new duplex so its center of mass is at the origin.

The module uses a second molecule of four atoms to represent a coordinate frame. One atom is at the origin and the other three are 10Å along the individual axes. These atoms are read from a PDB file into `m_axes`. Since they are both distant and in separate residues, the three calls to `connectres()` in lines 44-46 are used to bond the atom at the origin to the other three. When this molecule is drawn as lines, it will look like a coordinate frame. Since the coordinates of this molecule will be transformed each time the DNA is bent, their original coordinates are saved in the `point` array `s_axes` (line 47). Finally, a third strand "'axes" is added to the duplex and the contents of `m_axes` is added to it (lines 49-51) and the new molecule is sent out to network.

The second time and subsequent times the module is called, this `if` statement is skipped and lines 56-93 are executed instead. The `if`-tree in lines 58-67 computes the atom expression that will be use to select which residues of the duplex are to be transformed. A `base` value of `0` transforms the entire duplex with respect to the global coordinate system. Other `base` values transform the subduplex that extends from the selected base to the 3' end of the strand that contains that base. Thus `base` values from 1 to $N/2$ bend one half of the duplex and `base` values from $N/2+1$ to N bend the other half. As example, consider a molecule of 20 bases (or 10 base pairs). If `base` is set to `6`, then sub duplex consisting of bases 6-10:11-15 will be transformed. Now if `base` is changed to `15` the complement of `6`, the sub duplex 1-6:15-20 will be transformed.

The actual transformations are applied in lines 68-90. Lines 68-82 apply the rotations and 83-90 apply the translations. Rotations are done in the order of Z then Y then X. Each rotation is done around the the two atoms of the the third strand `"axes"` that both define the selected axis and represent it in the display. After each transformation the input parameter specifying that transformation's value is reset to `0.0` using the AVS library call `AVSmodify_float_parameter()`. `nab` has the builtin function `rot4()` to provide rotations about an arbitrary axis, but does have an analogous function for translating along an arbitrary axis. The user written C function `d2rd()` converts the desired translations along the axes at the bending site into equivalent displacements along standard axes. These displacements are converted into a transformation matrix in line 85 and then applied to the selected sub duplex. `putaxes()` is called to update the position and orientation of the `"axes"` strand and the newly bent molecule is sent to the network by the `return` statement in line 93.

The `nab` function `putaxes()` (lines 96-154) transforms the coordinates of the `"axes"` strand of `m` so they represent either the global coordinate frame or the coordinate frame defined by the Watson/Crick base pair from `base` to its mate. The function must deal with three cases defined by the value of `base`. If `base` is `0`, then the global frame has been selected. This is handled in line 104 by the `nab` builtin `setmol_from_xyz()` which replaces the current coordinates of `m`'s `"axes"` strand with their original values which were saved in `s_axes`.

The other two cases are for `base` values between 1 and $N/2$ and between $N/2+1$ to N. These are handled by the code in lines 105-128 and lines 129-153 respectively. These two sections do the same thing except that the first group of lines creates a frame that goes from the `"sense"` base to the `"anti"` base and the the second group creates a frame that goes from the `"anti"` base to the `"sense"` base.

A Watson/Crick base pair is normally associated with the following coordinate frame. The Y axis is along the direction from the `C1'` atom of a base to the `C1'` atom of its mate, the X axis is the perpendicular bisector of this vector that is in the mean base pair plane and the Z axes is X×Y. The

origin is located at the intersection of the base pair plane and the helical axis formed by a uniform duplex created from this base pair geometry.

The Y axis is directly accessible to `nab`, but it needs to create a stand in for the X axis from two atoms of the selected base. However, the the names of the required atoms depend on the type of selected base, which is not directly available at the `nab` level. The solution is another small user written C function, `getname_res_r()` which returns the residue name of the selected base. If the name begins with an `A` or a `G`, it is a purine and the X axis can be approximated by the vector from the `C5` to the `N3` atoms, otherwise it is a pyrimidine and the X axis goes from the `C5` to the `N1` atoms. Once the four atoms have been selected the `nab` builtin `setframe()` to build the desired coordinate frame.

Next the `"axes"` strand of `m` is transformed so that it agrees with this frame. This is done in lines 121-128 (or 145-152). First, the coordinates of `m_axes` are restored to their original values. Then the frame of `m_axes` is reset to be the global coordinate frame. The transformation in line 124 (148) translates these coordinates so that when the frame of `m_axes` is aligned with the frame created from the selected base pair in line 119 (143), the two atoms of `m_axis` representing the Z axis will be aligned along the helical axis made by an ideal B-form duplex formed by the selected base pair. `m_axes` is transformed by aligning its frame on the frame of the selected base pair and its new coordinates are replace the coordinates of the `"axes"` strand of `m` thus completing the operation.

```
1    // AVS_dnabender() - use helical parameters to deform DNA.
2
3    molecule    m, m_axes;
4    int     nres;
5    point       s_axes[ 4 ];
6    matrix      a_mat;
7
8    string      getname_res_r() c;
9    int     putaxes();
10   int     AVSmodify_float_parameter() c;
11   int     d2rd();
12
13   //AVSinfo   parm    base    0 0 1000
14   //AVSinfo   parm    dx  0 -3 3
15   //AVSinfo   parm    dy  0 -3 3
16   //AVSinfo   parm    dz  0 -3 3
17   //AVSinfo   parm    rx  0 -20 20
18   //AVSinfo   parm    ry  0 -20 20
19   //AVSinfo   parm    rz  0 -20 20
20   molecule    AVS_dnabender( string seq, int base,
21       float dx, float dy, float dz,
22       float rx, float ry, float rz )
23   {
24       string  cseq, arg;
25       string  aex;
26       matrix  mat;
27       point   com;
```

```
28          float   rdx, rdy, rdz;
29
30          if( !m ){
31              cseq = wc_complement( seq, "dna.amber94.rlb", "dna" );
32              m = wc_helix( seq, "dna.amber94.rlb", "dna",
33                  cseq, "dna.amber94.rlb", "dna",
34                  2.25, -4.96, 36.0, 3.38 );
35              nres = 2 * length( seq );
36              setpoint( m, NULL, com );
37              mat = newtransform( -com.x,-com.y,-com.z,0.,0.,0. );
38              transformmol( mat, m, NULL );
39
40              a_mat = newtransform( -2.25,0.,0.,0.,0.,0. );
41
42              m_axes = getpdb(
43                  "/home/macke/nab5/nreslib.0/XYZ.big.axes" );
44              connectres( m_axes, "1", 1, "ORG", 2, "SXT" );
45              connectres( m_axes, "1", 1, "ORG", 3, "CYT" );
46              connectres( m_axes, "1", 1, "ORG", 4, "NZT" );
47              setxyz_from_mol( m_axes, NULL, s_axes );
48
49              addstrand( m, "axes" );
50              mergestr( m, "axes", "last",
51                  m_axes, "1", "first" );
52
53              return( m );
54          }
55
56          if( base > nres )
57              base = nres;
58          if( base == 0 ){
59              aex = NULL;
60          }else if( base <= nres/2 ){
61              aex = sprintf( "sense:%d-%d:|anti:%d-%d:",
62                  base, nres / 2, 1, nres / 2 - base + 1 );
63          }else{
64              aex = sprintf( "sense:%d-%d:|anti:%d-%d:",
65                  1, nres - base + 1,
66                  base - nres / 2, nres / 2 );
67          }
68          if( rz != 0.0 ){
69              mat = rot4( m, "axes::O*", "axes::*Z*", rz );
70              transformmol( mat, m, aex );
71              AVSmodify_float_parameter( "rz",1,0.,0.,0. );
72          }
73          if( ry != 0.0 ){
74              mat = rot4( m, "axes::O*", "axes::*Y*", ry );
```

```
75              transformmol( mat, m, aex );
76              AVSmodify_float_parameter( "ry",1,0.,0.,0. );
77          }
78          if( rx != 0.0 ){
79              mat = rot4( m, "axes::O*", "axes::*X*", rx );
80              transformmol( mat, m, aex );
81              AVSmodify_float_parameter( "rx",1,0.,0.,0. );
82          }
83          if( dx != 0.0 || dy != 0.0 || dz != 0.0 ){
84              d2rd( m,dx,dy,dz,rdx,rdy,rdz );
85              mat = newtransform( rdx,rdy,rdz,0.,0.,0. );
86              transformmol( mat, m, aex );
87              AVSmodify_float_parameter( "dx",1,0.,0.,0. );
88              AVSmodify_float_parameter( "dy",1,0.,0.,0. );
89              AVSmodify_float_parameter( "dz",1,0.,0.,0. );
90          }
91          putaxes( m, base );
92
93          return( m );
94      };
95
96      int putaxes( molecule m, int base )
97      {
98          int sb, ab;
99          string  rname;
100         string  xt, xh, yt, yh;
101         point   apts[ 4 ];
102
103         if( base == 0 )
104             setmol_from_xyz( m, "axes::", s_axes );
105         else if( base <= nres/2 ){
106             rname = getname_res_r( m, base );
107             sb = base;
108             ab = nres / 2 - base + 1;
109             if( rname =~ "^[AG]" ){
110                 xt = sprintf( "sense:%d:C5", sb );
111                 xh = sprintf( "sense:%d:N3", sb );
112             }else{
113                 xt = sprintf( "sense:%d:C5", sb );
114                 xh = sprintf( "sense:%d:N1", sb );
115             }
116             yt = sprintf( "sense:%d:C1'", sb );
117             yh = sprintf( "anti:%d:C1'", ab );
118
119             setframe( 2, m, yt + "|" + yh, xt, xh, yt, yh );
120
121             setmol_from_xyz( m_axes, NULL, s_axes );
```

```
122             setframe( 2, m_axes,
123                 "::ORG", "::ORG", "::SXT", "::ORG", "::CYT" );
124             transformmol( a_mat, m_axes, NULL );
125
126             alignframe( m_axes, m );
127             setxyz_from_mol( m_axes, NULL, apts );
128             setmol_from_xyz( m, "axes::", apts );
129         }else{
130             rname = getname_res_r( m, base );
131             sb = nres - base + 1;
132             ab = base - nres / 2;
133             if( rname =~ "^[AG]" ){
134                 xt = sprintf( "anti:%d:C5", ab );
135                 xh = sprintf( "anti:%d:N3", ab );
136             }else{
137                 xt = sprintf( "anti:%d:C5", ab );
138                 xh = sprintf( "anti:%d:N1", ab );
139             }
140             yt = sprintf( "anti:%d:C1'", ab );
141             yt = sprintf( "sense:%d:C1'", sb );
142
143             setframe( 2, m, yt + "|" + yh, xt, xh, yt, yh );
144
145             setmol_from_xyz( m_axes, NULL, s_axes );
146             setframe( 2, m_axes,
147                 "::ORG", "::ORG", "::SXT", "::ORG", "::CYT" );
148             transformmol( a_mat, m_axes, NULL );
149
150             alignframe( m_axes, m );
151             setxyz_from_mol( m_axes, NULL, apts );
152             setmol_from_xyz( m, "axes::", apts );
153         }
154     };
```

8.5.2. Filter Modules.

This section covers `nab` generated filters, modules that receive a molecule from the network, process it and then send it back to the network. The computations that `nab` generated filters can perform are limited by what can be received and sent in the three fields that `nab` uses to create a "molecule port". A network molecule can have up to four extra `float` values per atom, accessed in `nab` as the `atom` attributes `charge`, `radius`, `float1` and `float2`, that can contain the data for and the results of a filter's computation. Any other data required for or generated by the filter must either must be packed into these `atom` attributes or not passed along the network. Nevertheless, there are many molecular computations that either operate at a per atom level or can be cast into that form. The two examples discussed next use the atom attribute `float1` to hold the result of their "per atom" calculations. This value will be used by the molecular display module `mv102` to color the atoms, providing a visual display of the computations.

8.5.2.1. Helical Interaction.

This example computes the hydrogen bonding patterns of a short helical peptide in a molecular dynamics trajectory. The peptide, $(AAQAA)_3$, is created in an all α-helical conformation. As the simulation proceeds parts of the helix shift between 3,10 and α. At the very end of the trajectory, the C-terminal end shifts between α and π. This code considers that a carbonyl oxygen is hydrogen bonded to an amino hydrogen if the O•••H distance is between 1.5Å and 3.0 Å and the CO•••H angle between 120° and 180°. The code assigns a value to each atom depending on the type of hydrogen bond it forms. The carbonyl atoms and the amino hydrogen atoms involved in 3,10, α and π helices are given the values `3`, `4` and `5`, which is the distance in residues between the donor and acceptor groups. An amino hydrogen atom that is shared between two carbonyl oxygens is given the average value of the two carbonyls. Atoms not involved in hydrogen bonding are assigned the "undefined" value `-1`.

The code uses two "user" `atom` attributes—`int1` and `float1`. `int1` will contain the number of hydrogen bonds an atom forms and `float1` will contain the "sum" of the donor:acceptor residue distances for each of these hydrogen bonds. After all hydrogen bonds have been detected, this sum will be divided by the number of hydrogen bonds to give values of `3`, `4` and `5` for 3,10, α and π helices and `3.5` and `4.5` for donors that are shared between 3,10 and α helices and α and π helices. The attributes are initialized in the `for-in` loop in lines 16-18 and the average computed in the `for-in` loop in lines 48-53. Atoms not participating in hydrogen bonding (ie `int1` still `0`) are given the value `-1.0` which is the default value that the `mv102` viewing module uses for an "undefined" property value.

The peptide has 17 residues—the 15 amino acids $(AAQAA)_3$ plus two end caps. The heart of the code is the two nested loops in lines 21-46. The outer loop ranges over the potential acceptor residues 2-13 and the inner loop over each acceptor's potential donors which are the residues 3 to 5 downstream from it. Since acceptor residues after residue 11 can not all three donors, the inner loop has exit-`if` (lines 23-24) that is taken if the donor residue number would be greater than 16.

If both donor and acceptor residues exist then they are checked for hydrogen bonding. The O•••H distance is computed in lines 26-29 and if it is outside of the range [1.5Å,3.0Å], the inner loop in lines 22-45 is advanced to the next donor candidate using the `continue` statement in line 29. If the O•••H distance is acceptable, the CO•••H angle is computed in lines 33-36 and if it is between 120° and 180° the `atom` attributes are updated. These two tests make use of `nab`'s `point` or vector operations. The O•••H and CO vectors are created using `point` differences in lines 28 and 34. Their lengths are determined using the infix dot product operator (`@`) in lines 29 and 35. Finally the angle between the CO and O•••H vectors is computed using another infix dot product in line 36.

The function `findatom()` is a user written C function that is used in place of the `nab` builtin `setpoint()`. Normally `nab` users select *sets* of atoms including those containing a single atom via atom expressions—strings that contains the desired atoms' strand, residue and atom names. To do so here would require creating atom expressions for the carbonyl carbon and oxygen and the amino hydrogen through `nab`'s string operations. For example, the atom expression that selects the carbonyl oxygen of residue `6` is `":6:O"`. Unfortunately while atom expressions are convenient they are not fast as they must be tested against *all* the atoms of molecule to see which ones they match. Since speed is important in this application, the C function `findatom()` was written to quickly select a *single* atom from a specified residue resulting in an execution speed up of about 30! For this reason it is likely that some mechanism like `findatom()` but using some form of infix notation will be added to `nab` to quickly select *single* atoms from a molecule.

```
1   // AVS_helix() - compute hbonds in a peptide helix
2
3   atom    findatom() c;
4   //AVSinfo   port    m   in
5   //AVSinfo   free    m
6   //AVSinfo   free    return-value
7   //AVSinfo   send    return-value    float1
8   molecule AVS_helix( molecule m )
9   {
10      atom    a;
11      int  ra, rd, r;
12      atom    c, o, h;
13      point   oc, oh;
14      float   ds, dy, dz, d_oh, d_oc, a_coh;
15
16      for( a in m ){
17          a.float1 = 0;
18          a.int1 = 0;
19      }
20
21      for( ra = 2; ra <= 13; ra = ra + 1 ){
22          for( r = 3; r <= 5; r = r + 1 ){
23              if( ( rd = ra + r ) > 16 )
24                  break;
25
26              o = findatom( m, "1", ra, "O" );
27              h = findatom( m, "1", rd, "H" );
28              oh = h.pos - o.pos;
29              d_oh = sqrt( oh @ oh );
30              if( d_oh < 1.50 || d_oh > 3.00 )
31                  continue;
32
33              c = findatom( m, "1", ra, "C" );
34              oc = c.pos - o.pos;
35              d_oc = sqrt( oc @ oc );
36              a_coh = acos( ( oh @ oc ) / ( d_oh * d_oc ) );
37              if( a_coh >= 120 && a_coh < 180.0 ){
38                  c.float1 = c.float1 + r;
39                  c.int1 = c.int1 + 1;
40                  o.float1 = o.float1 + r;
41                  o.int1 = o.int1 + 1;
42                  h.float1 = h.float1 + r;
43                  h.int1 = h.int1 + 1;
44              }
45          }
```

```
46          }
47
48          for( a in m ){
49              if( a.int1 )
50                  a.float1 = a.float1 / a.int1;
51              else
52                  a.float1 = -1.0;
53          }
54
55          return( m );
56      };
```

8.5.2.2. Protein Folding on a Lattice.

This filter analyses the conformations adopted by a protein during a folding simulation. It computes the distance for each atom between its position in the current conformation and its position in the final or folded conformation. This distance is saved in the `atom` attribute `float1`. The `send` directive in line 10 insures that these values are sent down the network along with the returned molecule where it will be used by `mv102` to color the atom depending on how far away it is from its final position.

The final conformation is read from the PDB file whose name is provided by `fsname` parameter. The positions of the atoms in this file are saved in the local private and persistent (declared as `static` in the C code) array `p_fs` (line 4). The `if`-tree in lines 17-26 insures that the final conformation file is read only once. Once the final structure file has been read, the `molecule` variable `m_fs` will no longer have the value `NULL` and the `if` will skip lines 18-25. The distances are are computed in the `for-in` loop in lines 29-33. The distances are always computed, even if the final structure has not yet been read in. In such cases, the distance will be the distance of the current atom's position from the origin, as `nab` initializes all global variables to 0.

```
1   // AVS_dist() - Compute atomic dist. between cur. & final conf.
2
3   molecule m_fs;
4   point  p_fs[ 100 ];
5   int natoms;
6   //AVSinfo   port    m   in
7   //AVSinfo   parm    fsname  ""
8   //AVSinfo   free    m
9   //AVSinfo   free    return-value
10  //AVSinfo   send    return-value    float1
11  molecule AVS_dist( molecule m, string fsname )
12  {
13      atom   a;
14      int anum;
15      point  vec;
16
17      if( !m_fs ){
```

```
18              if( fsname && fsname != "" ){
19                  m_fs = getpdb( fsname );
20                  natoms = 0;
21                  for( a in m_fs ){
22                      natoms = natoms + 1;
23                      p_fs[ natoms ] = a.pos;
24                  }
25              }
26          }
27
28          anum = 0;
29          for( a in m ){
30              anum = anum + 1;
31              vec = a.pos - p_fs[ anum ];
32              a.float1 = sqrt( vec @ vec );
33          }
34
35          return( m );
36      };
```

8.6. Data Output Example.

Data Output modules are the least used category of both `nab` created modules and user written modules in general. AVS is a *visualization* system, and nearly all networks end in either the Geometry Viewer or some other graphic output module. Nevertheless, there is at least one use for an `nab` generated Data Output module which is to save a “network molecule” as a PDB file. The module that does this is discussed next.

8.6.1. Write molecule.

`AVS_writepdb()` reads a molecule from its network `in` ports and saves it as a PDB file. The name of the PDB file is `fname`. The file will only be written when the integer parameter `write` is `1`. This is necessary because each module in a network is activated each time new data is presented to it. Without the `write` switch, `AVS_writepdb()` would be constantly writing and rewriting the file `fname`.

```
1       // AVS_writepdb() - save a network molecule as a PDB file
2
3       //AVSinfo   parm    fname
4       //AVSinfo   parm    write   0 0 1
5       //AVSinfo   port    mol in
6       //AVSinfo   free    mol
7       int AVS_writepdb( string fname, int write, molecule mol )
8       {
9
10          if( write ){
11              if( fname && fname != "" )
```

```
12                  return( putpdb( fname, mol ) );
13          }else
14              return( 0 );
15      };
```

9. LEaP

9.1. Introduction

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

```
Read AMBER PREP input files
Read AMBER PARM format parameter sets
Read and write Object File Format files (OFF)
Read and write PDB files
Construct new residues and molecules using simple commands
Link together residues and create nonbonded complexes of molecules
Place counterions around a molecule
Solvate molecules in arbitrary solvents
Modify internal coordinates within a molecule
Generate files that contain topology and parameters for AMBER and SPASMS
```

9.2. Concepts

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

9.2.1. Commands

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

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

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

For example:

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

Each command is followed by zero or more arguments that are separated by whitespace. Some commands return objects which are then associated with a variable using an assignment (=) statement. Each command acts upon its arguments, and some of the commands modify their arguments' contents. The commands themselves are case- insensitive. That is, in the above example, `edit` could have been entered as `Edit`, `eDiT`, or any combination of upper and lower case characters. Similarly, `loadPdb`

could have been entered a number of different ways, including `loadpdb`. In this manual, we frequently use a mixed case for commands. We do this to enhance the differences between commands and as a mnemonic device. Thus, while we write `createAtom`, `createResidue`, and `createUnit` in the manual, the user can use any case when entering these commands into the program.

The arguments in the command text may be *objects* such as NUMBERs, STRINGs, or LISTs or they may be *variables*. These two subjects are discussed next.

9.2.2. Variables

A *variable* is a handle for accessing an object. A variable name can be any alphanumeric string whose first character is an alphabetic character. (Alphanumeric means that the characters of the name may be letters, numbers, or special symbols such as "*". The following special symbols should not be used in variable names: dollar sign, comma, period, pound sign, equal sign, space, semicolon, double quote, or list open or close characters { and }. LEaP commands should not be used as variable names. Variable names are case-sensitive: "ARG" and "arg" are different variables. Variables are associated with objects using an assignment statement not unlike regular computer languages such as FORTRAN or C.

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

In the above examples, both `mole` and `MOLE` are variable names, whose contents are the same (6.02E23). Despite the fact that both `mole` and `MOLE` have the same contents, they are *not* the same variable. This is due to the fact that variable names are case-sensitive. LEaP maintains a list of variables that are currently defined and this list can be displayed using the `list` command. The contents of a variable can be printed using the `desc` command.

9.2.3. Objects

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

9.2.3.1. NUMBERs

NUMBERs are simple objects and they are identical to double precision variables in FORTRAN and double in C.

9.2.3.2. STRINGs

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

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

9.2.3.3. LISTs

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

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

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

9.2.3.4. PARMSETs (Parameter Sets)

PARMSETs are objects that contain bond, angle, torsion, and nonbond parameters for AMBER force field calculations. They are normally loaded from *e.g.* `parm94.dat` and `frcmod` files.

9.2.3.5. ATOMs

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

`name`

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

`type`

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

`charge`

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

`element`

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

`position`

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

AMBER also supports a type of calculation known as Free Energy Perturbation. During Free Energy Perturbation, one chemical species is slowly transformed into another and the energy change associated with the transformation is measured. In order to perform a Free Energy Perturbation, the properties of the perturbed ATOMs must also be set. These properties correspond to the ATOM properties described above, but the values represent the final state of the perturbed species, as described below. If a Free Energy Perturbation calculation is not to be performed, the following properties can be left as `null`. They are only used when the "PERTURB" property's value is "true" for that atom, when doing a `saveAmberParmPert` to save a perturbation topology file. (Note that mass is never perturbed.)

`pertName`

This property can either be `null` or a case sensitive STRING. The property is a unique identifier for an ATOM in its final state during a Free Energy Perturbation calculation. If it is `null` then the perturbed ATOM will inherit the unperturbed name. The `pertName` has no effect on calculations and is mainly useful as a reminder of what was intended.

`pertType`

This property can either be `null` or a STRING. If the value is `null` then the ATOM `type` will not be perturbed in a perturbation calculation. If the `pertType` is a STRING, the STRING is the AMBER force field atom type of the perturbed ATOM. This property is case-sensitive.

`pertCharge`

The `pertCharge` property is a NUMBER. It represents the final electrostatic point charge on an ATOM during a Free Energy Perturbation.

9.2.3.6. RESIDUEs

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

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

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

`connect0`

This defines an ATOM that is used in making links to other RESIDUEs. In UNITs containing single RESIDUEs, the RESIDUEs' `connect0` ATOM is usually defined as the UNITs' `head` ATOM. (This is how the standard library UNITs are defined.) For amino acids, the convention is to make the N-terminal nitrogen the `connect0` ATOM.

`connect1` This defines an ATOM that is used in making links to other RESIDUEs. In UNITs containing single RESIDUEs, the RESIDUEs' `connect1` ATOM is usually defined as the UNITs' `tail` ATOM. (This is done in the standard library UNITs.) For amino acids, the convention is to make the C-terminal oxygen the `connect1` ATOM.

`connect2` This is an ATOM property which defines an ATOM that can be used in making links to other RESIDUEs. In amino acids, the convention is that this is the ATOM to which disulphide bridges are made.

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

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

9.2.3.7. UNITs

UNITs are the most complex objects within LEaP, and the most important. UNITs, when paired with one or more PARMSETs, contain all of the information required to perform a calculation using AMBER. UNITs have the following properties which can be changed using the `set` command:

`head`

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

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

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

Examples of setting the above properties are:

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

The first example makes the amide nitrogen in the first RESIDUE within "dipeptide" the `head` ATOM. The second example places a rectangular bounding box around the origin with the (X, Y, Z)

dimensions of (5.0, 10.0, 15.0) in angstroms. The third example defines a solvent cap centered at (15.0, 10.0, 5.0) angstroms with a radius of 8.0 Å. **Note:** the "set cap" command does not actually solvate, it just sets an attribute. See the `solvateCap` command for a more practical case.

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

9.2.3.8. Complex objects and accessing subobjects

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

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

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

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

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

The first translates to "some RESIDUE named `ALA` within the UNIT named `dipeptide`". The second form translates as "the RESIDUE with sequence number `1` within the UNIT named `dipeptide`". The second form is more useful because every subobject within an object is guaranteed to have a unique sequence number. If the first form is used and there is more than one RESIDUE with the name `ALA`, then an arbitrary residue with the name `ALA` is returned. To access ATOMs within RESIDUEs, the notation to use is as follows:

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

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

In order to further illustrate the concepts of UNITs, RESIDUEs, and ATOMs, we can examine the log file from a LEaP session. Part of this log file is printed below.

```
> loadOff all_amino94.lib
> desc GLY
UNIT name: GLY
Head atom: .R<GLY 1>.A<N 1>
Tail atom: .R<GLY 1>.A<C 6>
Contents:
R<GLY 1>
> desc GLY.1
RESIDUE name: GLY
RESIDUE sequence number: 1
RESIDUE PDB sequence number: 0
Type: protein
Connection atoms:
 Connect atom 0: A<N 1>
 Connect atom 1: A<C 6>
Contents:
A<N 1>
A<HN 2>
A<CA 3>
A<HA2 4>
A<HA3 5>
A<C 6>
A<O 7>
> desc GLY.1.3
ATOM
Normal                  Perturbed
Name:    CA             CA
Type:    CT             CT
Charge:  -0.025         0.000
Element: C              (not affected by pert)
Atom position: 3.970048, 2.845795, 0.000000
Atom velocity: 0.000000, 0.000000, 0.000000
  Bonded to .R<GLY 1>.A<N 1> by a single bond.
  Bonded to .R<GLY 1>.A<HA2 4> by a single bond.
  Bonded to .R<GLY 1>.A<HA3 5> by a single bond.
  Bonded to .R<GLY 1>.A<C 6> by a single bond.
```

In this example, command lines are prefaced by ">" and the LEaP program output has no such character preface. The first command,

```
> loadOff all_amino94.lib
```

loads an OFF library containing amino acids. The second command,

```
> desc GLY
```

allows us to examine the contents of the amino acid UNIT, GLY. The UNIT contains one RESIDUE which is named GLY and this RESIDUE is the first residue in the UNIT (`R<GLY 1>`). In fact, it is also the only RESIDUE in the UNIT. The `head` and `tail` ATOMs of the UNIT are defined as the N- and C-termini, respectively. The `box` and `cap` UNIT properties are defined as "null". If these latter two properties had values other than "null", the information would have been included in the output of the `desc` command.

The next command line in the session,

```
> desc GLY.1
```

enables us to examine the first residue in the GLY UNIT. This RESIDUE is named GLY and its residue type is that of a `protein`. The `connect0` ATOM (N) is the same as the UNITs' `head` ATOM and the `connect1` ATOM (C) is the same as the UNITs' `tail` ATOM. There are seven ATOM objects contained within the RESIDUE GLY in the UNIT GLY.

Finally, let us look at one of the ATOMs in the GLY RESIDUE.

```
> desc GLY.1.3
```

The ATOM has a name (CA) that is unique among the atoms of the residue. The AMBER force field atom type for CA is CT. The type of element, atomic point charge, and Cartesian coordinates for this ATOM have been defined along with its bonding attributes. Other force field parameters, such as the van der Waals well depth, are obtained from PARMSETs.

9.3. Basic instructions for using LEaP with NAB

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

9.3.1. Building a Molecule For Molecular Mechanics

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

(1) Any needed UNIT or PARMSET objects must be loaded;

(2) The molecule must be constructed within LEaP;

(3) The user must output topology and coordinate files from LEaP to use in AMBER.

The most typical command sequence is the following:

```
source leaprc.ff94              load a force field
x = loadPdb trypsin.pdb         load in a structure
   ....                         add in cross-links, solvate, etc.
set default OldPrmtopFormat on  NAB uses an older version format
saveAmberParm x prmtop prmcrd   save files for sander or gibbs
```

There are a number of variants of this:

(1) Although *loadPdb* is by far the most common way to enter a structure, one might use *loadOff*, or *loadAmberPrep*, or use the *zmat* command to build a molecule from a z-matrix. See the

Commands section below for desciptions of these options. For case where you do not have a starting structure (in the form of a pdb file) LEaP can be used to build the molecule; you will find, however, that this is not always as easy as it might be. Many experienced Amber users turn to other (commerical and non-commerical) programs to create their initial structures.

(2) Be very attentive to any errors produced in the *loadPdb* step; these generally mean that LEaP has mis-read the file. A general rule of thumb is to keep editing your input pdb file until LEaP stops complaining. It is often convenient to use the *addPdbAtomMap* or *addPdbResMap* commands to make systematic changes from the names in your pdb files to those in the Amber topology files; see the *leaprc* files for examples of this.

(3) The *saveAmberParm* command cited above is appropriate for calculations that do not compute free energies; for the latter you will need to use *saveAmberParmPert*. For polarizable force fields, you will need to add *Pol* to the above commands (see the Commands section, below.)

9.3.2. Amino Acid Residues

The accompanying table shows the amino acid UNITs and their aliases are defined in the LEaP libraries.

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

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

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

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

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

Group or residue	Residue Name, Alias
Acetyl beginning group	ACE
Amine ending group	NHE
N-methylamine ending group	NME
Alanine	ALA
Arginine	ARG
Asparagine	ASN
Aspartic acid	ASP
Aspartic acid--protonated	ASH
Cysteine	CYS
Cystine, S--S crosslink	CYX
Glutamic acid	GLU
Glutamic acid--protonated	GLH
Glutamine	GLN
Glycine	GLY
Histidine, delta H	HID
Histidine, epsilon H	HIE
Histidine, protonated	HIP
Isoleucine	ILE
Leucine	LEU
Lysine	LYS
Methionine	MET
Phenylalanine	PHE
Proline	PRO
Serine	SER
Threonine	THR
Tryptophan	TRP
Tyrosine	TYR
Valine	VAL

9.3.3. Nucleic Acid Residues

The following are defined for the 1994 force field.

Group or residue	Residue Name, Alias
Adenine	DA,RA
Thymine	DT
Uracil	RU
Cytosine	DC,RC
Guanine	DG,RG

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

9.3.4. Miscellaneous Residues

Miscellaneous Residue	unit/residue name
TIP3P water molecule	TP3
Periodic box of TIP3P water	WATBOX216
TIP4P water model	TP4
TIP5P water model	TP5
SPC/E water model	SPC
Cesium cation	Cs+
Potassium cation	K+
Rubidium cation	Rb+
Lithium cation	Li+
Sodium cation	Na+ or IP
Chlorine	Cl- or IM
Large cation	IB

"IB" represents a solvated monovalent cation (say, sodium) for use in vacuum simulations. The cation UNITs are found in the files "ions91.lib" and "ions94.lib", while the water UNITs are in the file "solvents.lib". The `leaprc` files assign the variables WAT and HOH to the TP3 UNIT found in the OFF library file. Thus, if a PDB file is read and that file contains either the residue name HOH or WAT, the TP3 UNIT will be substituted. See Chapter 3 for a discussion of how to use other water models.

A periodic box of 216 TIP3P waters (WATBOX216) is provided in the file "solvents.lib". The box measures 18.774 angstroms on a side. This box of waters has been equilibrated by a Monte Carlo simulation. It is the UNIT that should be used to solvate systems with TIP3P water molecules within LEaP. It has been provided by W. L. Jorgensen. Boxes are also available for chloroform, methanol, and N-methylacetamide; these are described in Chapter 2.

9.4. Commands

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

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

9.4.1. add

`add a b`

`UNIT/RESIDUE/ATOM` *a,b*

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

The following example illustrates both the `add` command and the way the tip3p water molecule is created for the LEaP distribution tape.

```
> h1 = createAtom H1 HW  0.417
> h2 = createAtom H2 HW  0.417
> o  = createAtom O  OW -0.834
>
> set h1 element H
> set h2 element H
> set o  element O
>
> r = createResidue TIP3
> add r h1
> add r h2
> add r o
>
> bond h1 o
> bond h2 o
> bond h1 h2
>
> TIP3 = createUnit TIP3
>
> add TIP3 r
> set TIP3.1 restype solvent
> set TIP3.1 imagingAtom TIP3.1.O
>
> zMatrix TIP3 {
>       { H1 O 0.9572 }
>       { H2 O H1 0.9572 104.52 }
> }
>
> saveOff TIP3 water.lib
Saving TIP3.
Building topology.
Building atom parameters.
```

9.4.2. addAtomTypes

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

```
STRING type
STRING element
STRING hybrid
```

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

9.4.3. addIons

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

```
UNIT   unit
UNIT   ion1
NUMBER numIon1
UNIT   ion2
NUMBER numIon2
```

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

9.4.4. addIons2

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

```
UNIT   unit
UNIT   ion1
NUMBER numIon1
UNIT   ion2
NUMBER numIon2
```

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

9.4.5. addPath

```
addPath path
```

```
STRING path
```

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

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

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

9.4.6. addPdbAtomMap

```
addPdbAtomMap list
```

```
LIST   list
```

The atom Name Map is used to try to map atom names read from PDB files to atoms within residue UNITs when the atom name in the PDB file does not match an atom in the residue. This enables PDB files to be read in without extensive editing of atom names. Typically, this command is placed in the LEaP start-up file, "leaprc", so that assignments are made at the beginning of the session. The LIST is a LIST of LISTs. Each sublist contains two entries to add to the Name Map. Each entry has the form:

```
{ string string }
```

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

9.4.7. addPdbResMap

```
addPdbResMap list
```

```
LIST   list
```

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

```
{ double string string }
```

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

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

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

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

9.4.8. alias

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

```
STRING string1
STRING string2
```

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

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

9.4.9. bond

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

```
ATOM   atom1
ATOM   atom2
STRING order
```

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

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

9.4.10. bondByDistance

```
bondByDistance container [ maxBond ]
```

```
CONT   container
```

```
NUMBER maxBond
```

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

```
bondByDistance alkylChain
```

9.4.11. center

`center container`

```
UNIT/RESIDUE/ATOM container
```

Display the coordinates of the geometric center of the ATOMs within container. In the following example, the alanine UNIT found in the amino acid library has been examined by the center command:

```
> center ALA
The center is at: 4.04, 2.80, 0.49
```

9.4.12. charge

`charge container`

```
UNIT/RESIDUE/ATOM container
```

This command calculates the total charge of the ATOMs within container. The total charges for both standard and, where applicable, perturbed systems are displayed. In the following example, the alanine UNIT found in the amino acid library has been examined by the charge command:

```
> charge ALA
Total unperturbed charge: 0.00
Total perturbed charge:   0.00
```

9.4.13. check

`check unit [ parms ]`

```
UNIT   unit
PARMSETparms
```

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

- long bonds
- short bonds
- non-integral total charge of the UNIT.
- missing force field atom types
- close contacts (< 1.5 Å) between nonbonded ATOMs.

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

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

9.4.14. combine

```
variable = combine  list
```

```
object variable
LIST   list
```

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

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

9.4.15. copy

```
newvariable = copy  variable
```

```
object newvariable
object variable
```

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

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

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

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

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

9.4.16. createAtom

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

```
ATOM   variable
STRING name
STRING type
NUMBER charge
```

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

9.4.17. createParmset

```
variable = createParmset  name
```

```
PARMSETvariable
STRING name
```

Return a new and empty PARMSET with the name "name".

```
> newparms = createParmset pertParms
```

9.4.18. createResidue

```
variable = createResidue  name
```

```
RESIDUEvariable
STRING name
```

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

9.4.19. createUnit

```
variable = createUnit  name
```

```
UNIT   variable
STRING name
```

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

9.4.20. deleteBond

```
deleteBond atom1 atom2
```

```
ATOM   atom1
ATOM   atom2
```

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

9.4.21. desc

```
desc variable
```

```
object variable
```

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

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

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

```
> desc ALA.1
RESIDUE name: ALA
RESIDUE sequence number: 1
Type: protein
Connection atoms:
Connect atom 0: A<N 1>
Connect atom 1: A<C 9>
Contents:
A<N 1>
```

```
A<HN 2>
A<CA 3>
A<HA 4>
A<CB 5>
A<HB1 6>
A<HB2 7>
A<HB3 8>
A<C 9>
A<O 10>
```

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

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

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

```
> desc ALA.1.1
```

9.4.22. edit

```
edit unit
```

```
UNIT   unit
```

In xleap this command creates a Unit Editor that contains the UNIT unit. The user can view and edit the contents of the UNIT using the mouse. The command causes a copy of the object to be edited. If the object that the user wants to edit is "null", then the edit command assumes that the user wants to edit a new UNIT with a single RESIDUE within it. PARMSETs can also be edited. In tleap this command prints an error message.

9.4.23. groupSelectedAtoms

```
groupSelectedAtoms unit name
```

```
UNIT   unit
STRING name
```

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

```
groupSelectedAtoms TRP sideChain
```

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

```
select TRP@sideChain


center TRP@sideChain
```

The latter example will calculate the center of the atoms in the "sideChain" group. (see the *select* command for a more detailed example.)

9.4.24. help

```
help [string]

    STRING string
```

This command prints a description of the command in string. If the STRING is not given then a list of help topics is provided.

9.4.25. impose

```
impose unit seqlist internals

    UNIT   unit
    LIST   seqlist
    LIST   internals
```

The impose command allows the user to impose internal coordinates on the UNIT. The list of RESIDUEs to impose the internal coordinates upon is in seqlist. The internal coordinates to impose are in the LIST internals.

The command works by looking into each RESIDUE within the UNIT that is listed in the seqlist argument and attempts to apply each of the internal coordinates within internals. The seqlist argument is a LIST of NUMBERS that represent sequence numbers or ranges of sequence numbers. Ranges of sequence numbers are represented by two element LISTs that contain the first and last sequence number in the range. The user can specify sequence number ranges that are larger than what is found in the UNIT. For example, the range { 1 999 } represents all RESIDUEs in a 200 RESIDUE UNIT.

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

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

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

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

will impose on the residues with sequence numbers 1, 2, 5, 6, 7, 8, 9, 10, and 12 within the UNIT peptide a bond length of 5.0 angstroms between the alpha and beta carbons. RESIDUEs without an ATOM named CB (like glycine) will be unaffected.

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

9.4.26. list

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

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

9.4.27. loadAmberParams

```
variable = loadAmberParams  filename
```

```
PARMSETvariable
STRING filename
```

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

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

```
Saving parm91.
```

9.4.28. loadAmberPrep

```
loadAmberPrep filename [ prefix ]
```

```
STRING filename
STRING prefix
```

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

```
   0  0  2
     Crown Fragment A
cra.res
CRA  INT 0
CORRECT  NOMIT DU  BEG
0.0
1  DUMM  DU M  0  0  0     0.      0.      0.
2  DUMM  DU M  0  0  0     1.000   0.      0.
3  DUMM  DU M  0  0  0     1.000  90.      0.
4  C1    CT M  0  0  0     1.540 112.    169.
5  H1A   HC E  0  0  0     1.098 109.47  -110.0
6  H1B   HC E  0  0  0     1.098 109.47   110.0
7  O2    OS M  0  0  0     1.430 112.    -72.
8  C3    CT M  0  0  0     1.430 112.    169.
9  H3A   HC E  0  0  0     1.098 109.47  -49.0
10 H3B   HC E  0  0  0     1.098 109.47   49.0

CHARGE
 0.2442  -0.0207  -0.0207  -0.4057  0.2442
 -0.0207  -0.0207

DONE
STOP
```

This fragment can be loaded into LEaP using the following command:

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

9.4.29. loadOff

```
loadOff filename

      STRING filename
```

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

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

9.4.30. loadPdb

```
variable = loadPdb filename

      STRING filename
      object variable
```

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

```
> crambin = loadPdb 1crn
Loading PDB file
Matching PDB residue names to LEaP variables.
Mapped residue THR, term: 0, seq. number: 0 to: NTHR.
Residue THR, term: M, seq. number: 1 was not
found in name map.
Residue CYS, term: M, seq. number: 2 was not
found in name map.
```

```
Residue CYS, term: M, seq. number: 3 was not
found in name map.
Residue PRO, term: M, seq. number: 4 was not
found in name map.
:               :               :
Residue TYR, term: M, seq. number: 43 was not
found in name map.
Residue ALA, term: M, seq. number: 44 was not
found in name map.
Mapped residue ASN, term: 1, seq. number: 45 to: CASN.
Joining NTHR - THR
Joining THR - CYS
Joining CYS - CYS
Joining CYS - PRO
:               :               :
Joining ASP - TYR
Joining TYR - ALA
Joining ALA - CASN
```

The above edited listing shows the use of this command to load a PDB file for the protein crambin. Several disulphide bonds are present in the protein and these bonds are indicated in the PDB file. The loadPdb command, however, cannot read this information from the PDB file. It is necessary for the user to explicitly define disulphide bonds using the *bond* command.

9.4.31. loadPdbUsingSeq

`loadPdbUsingSeq filename unitlist`

```
STRING filename
LIST   unitlist
```

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

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

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

9.4.32. logFile

`logFile filename`

```
STRING filename
```

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

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

9.4.33. measureGeom

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

```
ATOM   atom1
ATOM   atom2
ATOM   atom3
ATOM   atom4
```

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

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

```
> desc ALA.ALA
RESIDUE name: ALA
RESIDUE sequence number: 1
Type: protein
Connection atoms:
Connect atom 0: A<N 1>
Connect atom 1: A<C 9>
Contents:
A<N 1>
A<HN 2>
A<CA 3>
A<HA 4>
A<CB 5>
A<HB1 6>
A<HB2 7>
A<HB3 8>
A<C 9>
A<O 10>
> measureGeom ALA.ALA.1 ALA.ALA.3
Distance: 1.45 angstroms
> measureGeom ALA.ALA.1 ALA.ALA.3 ALA.ALA.5
Angle: 111.10 degrees
> measureGeom ALA.ALA.N ALA.ALA.CA ALA.ALA.C ALA.ALA.O
Torsion angle: 0.00 degrees
```

9.4.34. quit

Quit the LEaP program.

9.4.35. remove

```
remove a b
```

```
CONT   a
CONT   b
```

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

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

9.4.36. saveAmberParm

```
saveAmberParm unit topologyfilename coordinatefilename
```

```
UNIT   unit
STRING topologyfilename
STRING coordinatefilename
```

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

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

```
> saveamberparm ALA ala.top ala.crd
Building topology.
Building atom parameters.
Building bond parameters.
Building angle parameters.
Building proper torsion parameters.
Building improper torsion parameters.
Building H-Bond parameters.
```

9.4.37. saveAmberParmPol

```
saveAmberParmPol unit topologyfilename coordinatefilename
```

```
UNIT   unit
STRING topologyfilename
STRING coordinatefilename
```

Like saveAmberParm, but includes atomic polarizabilities in the topology file for use with IPOL=1 in Sander. The polarizabilities are according to atom type, and are defined in the 'mass' section of the *parm.dat* or *frcmod* file. Note: charges are normally scaled when polarizabilities are used - see scaleCharges for an easy way of doing this.

9.4.38. saveAmberParmPert

```
saveAmberParmPert unit topologyfilename coordinatefilename
```

```
UNIT   unit
STRING topologyfilename
STRING coordinatefilename
```

This command is the same as *saveAmberParm*, except a perturbation topology file is written instead of a plain minimization/dynamics one.

Save the AMBER topology and coordinate files for the UNIT into the files named topologyfilename and coordinatefilename respectively. This command will cause LEaP to search its list of PARMSETs for parameters defining all of the interactions between the ATOMs within the UNIT. This command produces topology files and coordinate files that are identical in format to those produced by AMBER PARM and can be read into gibbs for perturbation calculations.

```
> saveAmberParmPert pert pert.leap.top pert.leap.crd
Building topology.
Building atom parameters.
Building bond parameters.
Building angle parameters.
Building proper torsion parameters.
Building improper torsion parameters.
Building H-Bond parameters.
```

9.4.39. saveAmberParmPolPert

```
saveAmberParmPolPert unit topologyfilename coordinatefilename
```

```
UNIT   unit
STRING topologyfilename
STRING coordinatefilename
```

Like saveAmberParmPert, but includes atomic polarizabilities in the topology file for use with IPOL=1 in Gibbs. The polarizabilities are according to atom type, and are defined in the 'mass' section of the parm.dat or frcmod file. Note: charges are normally scaled when polarizabilities are used - see scaleCharges for an easy way of doing this.

9.4.40. saveOff

```
saveOff object filename
```

```
object object
STRING filename
```

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

9.4.41. savePdb

```
savePdb unit filename
```

```
UNIT   unit
STRING filename
```

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

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

9.4.42. scaleCharges

```
scaleCharges container scale_factor
```

```
UNIT/RESIDUE/ATOM   container
NUMBER              scale_factor
```

This command scales the charges in the object by _scale_factor_, which must be > 0. It is useful for building systems for use with polarizable atoms, e.g.

```
> x = copy solute
> scaleCharges x 0.8
> y = copy WATBOX216
> scalecharges y 0.875
> solvatebox x y 10
> saveamberparmpol x x.top x.crd
```

9.4.43. sequence

```
variable = sequence  list
```

```
UNIT   variable
LIST   list
```

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

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

9.4.44. set

```
set default variable value
```

```
      STRING variable
      STRING value
```

or

```
set container parameter object

      CONT   container
      STRING parameter
      object object
```

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

For "default" parameters

OldPrmtopFormat
: If set to "on", the saveAmberParm command will write a prmtop file in the format used in Amber6 and before; if set to "off" (the default), it will use the new format.

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

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

For ATOMs:

name
: A unique STRING descriptor used to identify ATOMs.

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

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

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

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

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

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

For RESIDUEs:

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

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

connect2 This is an ATOM property which defines an ATOM that can be used in making links to other RESIDUEs. In amino acids, the convention is that this is the ATOM to which disulphide bridges are made.

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

name This STRING property is the RESIDUE name.

For UNITs:

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

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

box The property defines the bounding box of the UNIT. If it is defined as null then no bounding box is defined. If the value is a single NUMBER then the bounding box will be defined to be a cube with each side being NUMBER of angstroms across. If the value is a LIST then it must be a LIST containing three numbers, the lengths of the three sides of the bounding box.

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

9.4.45. setBox

```
setBox unit   vdw OR centers  [ buffer OR buffer_xyz_list ]

      UNIT   unit
```

The setBox command adds a periodic box to the UNIT, turning it into a periodic system for the simulation programs. It does not add any solvent to the system. The choice of "vdw" or "centers" determines whether the box encloses the entire atoms or just the atom centers - use "centers" if the system has been previously equilibrated as a periodic box. See the solvateBox command for a description of the buffer variable, which extends either type of box by an arbitrary amount.

9.4.46. solvateBox

```
solvateBox solute solvent buffer [ iso ] [ closeness ]

      UNIT   solute
      UNIT   solvent
      object buffer
      NUMBER closeness
```

The *solvateBox* command creates a rectangular parallelopiped solvent box around the solute UNIT. The solute UNIT is modified by the addition of solvent RESIDUEs. (For most liquid state simulations, the *solvateOct* command discussed below is probably a better choice.)

The normal choice for a TIP3 _solvent_ UNIT is WATBOX216, which is a snapshot from a room-temperature equilibration for this model. If you want to solvate with other water models (say TIP4P), try the following: *(a)* solvate the system with WATBOX216, using the default TIP3 model; *(b)* use *ambpdb* to convert your *prmtop* file to Brookhaven format; *(c)* restart LEaP, choose the TIP4P water model (instructions are in the Database chapter), then use loadPdb to bring back in the system you have created.

Note that equilibration will always be required to bring the artificial box to reasonable density, since Van der Waals voids remain due to the impossibility of natural packing of solvent around the solute and at the edges of the box. First, equilibrate the system at constant volume to the temperature you want, then turn on constant pressure to adjust the system density to the desired value.

The solvent UNIT is copied and repeated in all three spatial directions to create a box containing the entire solute and a buffer zone defined by the buffer argument. The buffer argument defines the distance, in angstroms, between the wall of the box and the closest ATOM in the solute. If the buffer argument is a single NUMBER, then the buffer distance is the same for the x, y, and z directions, unless the 'iso' option is used to make the box cubic, with the shortest box clearance = buffer. If the buffer argument is a LIST of three NUMBERS, then the NUMBERs are applied to the x, y, and z axes respectively. As the larger box is created and superimposed on the solute, solvent molecules overlapping the solute are removed.

The optional closeness parameter can be used to control how close, in angstroms, solvent ATOMs can come to solute ATOMs. The default value of the closeness argument is 1.0. Smaller values allow solvent ATOMs to come closer to solute ATOMs. The criterion for rejection of overlapping solvent RESIDUEs is if the distance between any solvent ATOM to the closest solute ATOM is less than the sum of the ATOMs VANDERWAAL distances multiplied by the closeness argument.

This command modifies the _solute_ UNIT in several ways. First, the coordinates of the ATOMs are modified to move the center of a box enclosing the Van der Waals radii of the atoms to the origin. Secondly, the UNIT is modified by the addition of _solvent_ RESIDUEs copied from the _solvent_ UNIT. Finally, the box parameter of the new system (still named for the _solute_) is modified to reflect the fact that a periodic, rectilinear solvent box has been created around it.

In this example, it is assumed that the file solvents.lib, containing WATBOX216, has been loaded already (as is done by the default leaprc):

```
      >> mol = loadpdb my.pdb
```

```
>> solvateBox sol WATBOX216 10
  Solute vdw bounding box:              7.512 12.339 12.066
  Total bounding box for atom centers:  27.512 32.339 32.066
  Solvent unit box:                     18.774 18.774 18.774
  Total vdw box size:                   30.995 35.538 35.416 angstroms.
  Total mass 14470.768 amu,  Density 0.616 g/cc
  Added 785 residues.
```

Again, note that the density of 0.601 g/cc points to the need for constant pressure equilibration. (See the discussion of equilibration in the Q&A section of the amber web.)

9.4.47. solvateCap

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

```
UNIT   solute
UNIT   solvent
object position
NUMBER radius
NUMBER closeness
```

The solvateCap command creates a solvent cap around the solute UNIT. The solute UNIT is modified by the addition of solvent RESIDUEs. The solvent box will be repeated in all three spatial directions to create a large solvent sphere with a radius of radius angstroms.

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

The optional closeness parameter can be used to control how close, in angstroms, solvent ATOMs can come to solute ATOMs. The default value of the closeness argument is 1.0. Smaller values allow solvent ATOMs to come closer to solute ATOMs. The criterion for rejection of overlapping solvent RESIDUEs is if the distance between any solvent ATOM to the closest solute ATOM is less than the sum of the ATOMs VANDERWAAL's distances multiplied by the closeness argument.

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

```
>> mol = loadpdb my.pdb
>> solvateCap mol WATBOX216 mol.2.CA 8.0 2.0
Added 3 residues.
```

9.4.48. solvateDontClip

```
solvateDontClip solute solvent buffer [ closeness ]
```

```
UNIT   solute
UNIT   solvent
object buffer
NUMBER closeness
```

This command is identical to the *solvateBox* command except that the solvent box that is created is not clipped to the boundary of the buffer region. This command forms larger solvent boxes than does *solvateBox* because it does not cause solvent that is outside the buffer region to be discarded. This helps to preserve the periodic structure of properly constructed solvent boxes, preventing hot-spots from forming.

```
>> mol = loadpdb my.pdb
>> solvateDontClip mol WATBOX216 10
  Solute vdw bounding box:              7.512 12.339 12.066
  Total bounding box for atom centers:  27.512 32.339 32.066
  Solvent unit box:                     18.774 18.774 18.774
  Total vdw box size:                   41.120 40.899 41.075 angstroms.
  Total mass 30595.088 amu,  Density 0.735 g/cc
  Added 1680 residues.
```

Note the larger number of waters added, compared to solvateBox; in the case of this solute and choice of buffer, the overall box size is increased by about 10 angstroms in each direction.

9.4.49. solvateOct

```
solvateOct solute solvent buffer [aniso] [ closeness ]
```

```
UNIT                        _solute_
UNIT                        _solvent_
object                      _buffer_
NUMBER                      _closeness_
```

The solvateOct command is the same as solvateBox, except the corners of the box are sliced off, resulting in a truncated octahedron, which typically gives a more uniform distribution of solvent around the solute. In solvateOct, when a LIST is given for the buffer argument, four numbers are given instead of three, where the fourth is the diagonal clearance. If 0.0 is given as the fourth number, the diagonal clearance resulting from the application of the x,y,z clearances is reported. If a non-0 value is given, this may require scaling up the other clearances, which is also reported.

Unless the 'aniso' option is used, an isometric truncated octahedron is produced and rotated to an orientation used by the *sander* PME code. (Note: don't use the 'aniso' option unless you are sure you know what you are doing; it is only there for expert backward compatibility, and probably has no real use anymore.)

9.4.50. solvateShell

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

```
UNIT   solute
```

```
UNIT   solvent
NUMBER thickness
NUMBER closeness
```

The *solvateShell* command adds a solvent shell to the solute UNIT. The resulting solute/solvent UNIT will be irregular in shape since it will reflect the contours of the solute. The solute UNIT is modified by the addition of solvent RESIDUEs. The solvent box will be repeated in three directions to create a large solvent box that can contain the entire solute and a shell thickness angstroms thick. The solvent RESIDUEs are then added to the solute UNIT if they lie within the shell defined by thickness and do not overlap with the solute ATOMs. The optional closeness parameter can be used to control how close solvent ATOMs can come to solute ATOMs. The default value of the closeness argument is 1.0. Please see the *solvateBox* command for more details on the closeness parameter.

```
>> mol = loadpdb my.pdb
>> solvateShell mol WATBOX216 8.0
  Solute vdw bounding box:              7.512 12.339 12.066
  Total bounding box for atom centers:  23.512 28.339 28.066
  Solvent unit box:                     18.774 18.774 18.774
  Added 147 residues.
```

9.4.51. source

```
source filename
```

```
STRING filename
```

This command executes commands within a text file. To display the commands as they are read, see the *verbosity* command.

9.4.52. transform

```
transform atoms, matrix
```

```
CONT   atoms
LIST   matrix
```

Transform all of the ATOMs within atoms by the (3×3) or (4×4) matrix represented by the nine or sixteen NUMBERS in the LIST of LISTs *matrix*. The general matrix looks like:

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

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

```
1   0    0
```

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

This reflection could be combined with a six angstrom translation along the x-axis by using the following matrix.

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

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

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

9.4.53. translate

`translate atoms direction`

```
CONT   atoms
LIST   direction
```

Translate all of the ATOMs within atoms by the vector defined by the three NUMBERS in the LIST *direction*.

Example:

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

9.4.54. verbosity

`verbosity level`

```
NUMBER level
```

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

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

9.4.55. zMatrix

```
zMatrix object zmatrix

      CONT   object
      LIST   matrix
```

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

```
{ a1 a2 bond12 }
```

This entry defines the coordinate of a1 by placing it bond12 angstroms along the x-axis from ATOM a2. If ATOM a2 does not have coordinates defined then ATOM a2 is placed at the origin.

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

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

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

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

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

This entry defines the coordinate of a1 by placing it bond12 angstroms away from ATOM a2, making angles angle123 between ATOMs a1, a2, and a3, and angle124 between ATOMs a1, a2, and a4. The argument orientation defines whether the ATOM a1 is above or below a plane defined by the ATOMs a2, a3, and a4. If orientation is positive then a1 will be placed in such a way so that the inner product of (a3-a2) cross (a4-a2) with (a1-a2) is positive. Otherwise a1 will be placed on the other side of the plane. This allows the coordinates of a molecule like fluoro-chloro-bromo-methane to be defined without having to resort to dummy atoms.

The first arguments within the *zMatrix* entries (a1, a2, a3, a4) are either ATOMs or STRINGS containing names of ATOMs within object. The subsequent arguments are all NUMBERS. Any ATOM can be placed at the a1 position, even those that have coordinates defined. This feature can be used to provide an endless supply of dummy atoms, if they are required. A predefined dummy atom with the name "*" (a single asterisk, no quotes) can also be used.

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

10. Index