Antechamber Users' Manual

Version 1.0

Junmei Wang[1] and David A. Case[2]

[1]*Encysive, Inc., Houston TX, 77030*

[2]*The Scripps Research Institute, La Jolla, CA 92037*

We thank Peter Kollman for encouragement and advice during the development of this project. *other acknowledgements here!*

1. Installation and Getting Started.

1.1. Installation.

The *antechamber* package is available from http://amber.scripps.edu/pub/antechamber/download.html. The first step in setting up the nab package is to unpack the tar file using the UNIX commands gunzip and tar:

```
gunzip antechamber-1.0.x.tar.gz
tar xvf antechamber-1.0.x.tar
```

The path to this new directory (*e.g.* `/usr/local/antechamber-1.0`, if you unpacked the distribution in `/usr/local`) should be defined as the environment variable $ACHOME. If you are using *sh, zsh* or *bash* as your shell:

```
export ACHOME=insertyourpathhere/antechamber-1.0
```

If you are using *csh* or *tcsh* as your shell:

```
setenv ACHOME insertyourpathhere/antechamber-1.0
```

Now, in the top-level ($ACHOME) directory, you should edit the *config.h* file, if necessary. If you have the GNU compiler tools installed, no changes should be necessary. Then

```
make
```

will compile things, putting the executable files in $ACHOME/bin. 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 = ( $ACHOME/bin $path )
rehash
```

1.2. Contacting the developers

Please send suggestions and questions to *jwang@encysive.com* or `case@scripps.edu`.

2. Antechamber

This is a set of tools to generate "prep" input files for organic molecules, which can then be read into LEaP. The Antechamber suite was written by Junmei Wang, and is designed to be used in conjunction with the "general AMBER force field (GAFF)" (*gaff.dat*) [1].

Molecular mechanics are the key component in the armamentarium used by computational chemists for rational drug design and many other tasks. Force fields are the cornerstone of molecular mechanics. A successful force field for drug design should work well both for biological macromolecules and the organic molecules. The AMBER force fields have built up a good reputation for its performance in studies of proteins and nucleic acids. However, the fact that AMBER has had only limited parameters for organic molecules has kept it from being widely used in ligand-binding or drug design applications. Antechamber is based on a new, general AMBER force field (GAFF) that covers most pharmaceutical molecules, and which is as compatible as possible with the traditional AMBER force fields.

Like the traditional AMBER force fields, GAFF uses a simple harmonic function form for bonds and angles. Unlike the traditional AMBER force fields, atom types in GAFF are more general and cover most of the organic chemical space. In total there are 33 basic atom types and 22 special atom types. The charge methods used in GAFF can be HF/6-31G* RESP or AM1-BCC [2,3]. All of the force field parameterizations were carried out with HF/6-31G* RESP charges. However, in most cases, AM1-BCC, which was parameterized to reproduce HF/6-31G* RESP charges, is recommended in the large-scale calculations because of its efficiency.

The van der Waals parameters are the same as those used by the traditional AMBER force fields. The equilibrium bond lengths and bond angles came from statistics derived from the Cambridge Structural Database, and *ab initio* calculations at the MP2/6-31G* level. The force constants for bonds and angles were estimated using empirical models, and the parameters in these models were trained using the force field parameters in the traditional AMBER force fields. General torsional angle parameters were extensively applied in order to reduce the huge number of torsional angle parameters to be derived. The force constants and phase angles in the torsional angle parameters were optimized using our PARMSCAN package [4], with an aim to reproduce the rotational profiles depicted by high-level *ab initio* calculations [geometry optimizations at the MP2/6-31G* level, followed by single point calculations at MP4/6-311G(d,p)].

By design, GAFF is a complete force field (so that missing parameters rarely occur), it covers almost all the organic chemical space that is made up of C, N, O, S, P, H, F, Cl, Br and I. Moreover, GAFF is totally compatible to the AMBER macromolecular force fields. We believe that the combination of GAFF with AMBER macromolecular force fields will provide an useful molecular mechanical tool for rational drug design, especially in binding free energy calculations and molecular docking studies.

As an auxiliary module in AMBER software packages, antechamber is devoted to build up the bridge between the force fields (GAFF and AMBER) and the MM programs, such as sander et al. With antechamber, one may solve the following problems: (1) identifying bond and atom types; (2) judging atomic equivalence; (3) generating residue topology files; and (4) finding missing force field parameters and supplying reasonable suggestions. The combination of GAFF and antechamber enables one to study most of the organic molecules with AMBER more efficiently. In the following, the main programs in the antechamber package are introduced.

2.1. Principal programs

The *antechamber* program itself is the main program of Antechamber: if your molecule falls in fairly broad categories, this should be all you need to convert an input pdb file into a "prep input" file ready for LEaP.

If there are missing parameters after *antechamber* is finished, you may want to run *parmchk* to generate a *frcmod* template that will assist you in generating the needed parameters.

2.1.1. antechamber

This is the most important program in the package. It can perform many file conversions, and can also assign atomic charges and atom types. As required by the input, *antechamber* executes the following programs: *divcon, atomtype, am1bcc, bondtype, espgen, respgen* and *prepgen*. It may also generate a lot of intermediate files (all in capital letters). If there is a problem with *antechamber*, you may want to run the individual programs that are described below. Antechamber options are given here:

```
-i   input file name
-fi  input file format
-o   output file name
-fo  output file format
-c   charge method
-cf  charge file name
-nc  net molecular charge (int)
-a   additional file name
-fa  additional file format
-ao  additional file operation
     crd : only read in coordinate
     crg: only read in charge
     name  : only read in atom name
     type  : only read in atom type
     bond  : only read in bond type
-m   mulitiplicity (2S+1), default is 1
-rn  residue name, if not available in the input file, default is MOL
-rf  residue toplogy file name in prep input file, default is molecule.res
-mk  divcon keyword, in a pair of quotation marks
-gk  gaussian keyword, in a pair of quotation marks
-at  atom type, can be gaff, amber, bcc and sybyl, default is gaff
-du  check atom name duplications, can be yes(y) or no(n), default is yes
-j   atom type and bond type prediction index, default is 4
     0    : no assignment
     1    : atom type
     2    : full  bond types
     3    : part  bond types
     4    : atom and full bond type
     5    : atom and part bond type
-s   status information, can be 0 (brief), 1 (the default) and 2 (verbose)
-pf  remove the intermediate files: can be yes (y) and no (n), default is no
-i -o -fi and -fo must appear in command lines and the others are optional
```

```
              List of the File Formats

file format type  abbre. index | file format type abbre. index
-------------------------------------------------------------
Antechamber        ac       1  | Sybyl Mol2         mol2    2
PDB                pdb      3  | Modifiled PDB      mpdb    4
amber PREP (int)   prepi    5  |
Gaussian Z-Matrix  gzmat    7  | Gaussian Cartesian gcrt    8
Mopac Internal     mopint   9  | Mopac Cartesian    mopcrt 10
Gaussian Output    gout    11  | Mopac Output       mopout 12
Alchemy            alc     13  | CSD                csd    14
MDL                mdl     15  | Hyper              hin    16
amber Restart      rst     17
-------------------------------------------------------------

amber restart file can only be read in as additional file

              List of the Charge Methods

charge method     abbre.  index | charge method      abbre. index
-----------------------------------------------------------------
RESP               resp     1  |  AM1-BCC            bcc     2
CM2                cm2      3  |  ESP (Kollman)      esp     4
Mulliken           mul      5  |  Gasteiger          gas     6
Read in Charge     rc       7  |  Write out charge   wc      8
-----------------------------------------------------------------
```

Examples:

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

The *-rn* line specifies the residue name to be used; thus, it must be one to three characters long. The *-at* flag is used to specify whether atom types are to be created for the general AMBER force field (gaff) or for atom types consistent with parm94.dat and parm99.dat (amber). Atom types for gaff are all in lower case, and the AMBER atom types are always in upper case. If you are using *antechamber* to create a modified residue for use with the standard AMBER parm94/parm99 force fields, you should set this flag to `amber`; if you are looking at a more arbitrary molecule, set this to `gaff`, even if you plan to use this as a ligand bound to a macromolecule described by the AMBER force fields.

2.1.2. parmchk

Parmchk reads in an ac file as well as a force field file (*gaff.dat* in $AMBERHOME/dat/leap/parm). It writes out a frcmod file for the missing parameters. For each atom type, an atom type corresponding file (ATCOR.DAT) lists its replaceable general atom type. Be careful to those problematic parameters indicated with "ATTN, need revision".

```
Usage: parmchk -i input file name
               -o frcmod file name
               -f input file format (prepi, ac ,mol2)
               -p ff parmfile
               -c atom type correspondening file, default is ATCOR.DAT
```

Example:

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

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

2.2. A simple example for antechamber

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

As with any automated procedure, caution should be taken to examine the output. Furthermore, the procedure, although carefully tested, has not been widely used by lots of people, so users should certainly be on the lookout for unusual or incorrect behavior.

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

```
ATOM      1  CG  TP      1      -1.959   0.102   0.795
ATOM      2  CD1 TP      1      -1.249   0.602  -0.303
ATOM      3  CD2 TP      1      -2.071   0.865   1.963
ATOM      4  CE1 TP      1      -0.646   1.863  -0.234
ATOM      5  C6  TP      1      -1.472   2.129   2.031
ATOM      6  CZ  TP      1      -0.759   2.627   0.934
ATOM      7  HE2 TP      1      -1.558   2.719   2.931
ATOM      8  S15 TP      1      -2.782   0.365   3.060
ATOM      9  H19 TP      1      -3.541   0.979   3.274
ATOM     10  H29 TP      1      -0.787  -0.043  -0.938
ATOM     11  H30 TP      1       0.373   2.045  -0.784
ATOM     12  H31 TP      1      -0.092   3.578   0.781
ATOM     13  H32 TP      1      -2.379  -0.916   0.901
```

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

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

This command says that the input format is pdb, output format is Sybyl mol2, and the BCC charge model is to be used. The output file is shown in the box titled *tp.mol2*. The format of this file is a common one understood by many programs.

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

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

tp.mol2

```
@<TRIPOS>MOLECULE
TP
   13    13     1     0     0
SMALL
bcc


@<TRIPOS>ATOM
      1 CG          -1.9590    0.1020    0.7950 ca        1 TP       -0.1186
      2 CD1         -1.2490    0.6020   -0.3030 ca        1 TP       -0.1138
      3 CD2         -2.0710    0.8650    1.9630 ca        1 TP        0.0162
      4 CE1         -0.6460    1.8630   -0.2340 ca        1 TP       -0.1370
      5 C6          -1.4720    2.1290    2.0310 ca        1 TP       -0.1452
      6 CZ          -0.7590    2.6270    0.9340 ca        1 TP       -0.1122
      7 HE2         -1.5580    2.7190    2.9310 ha        1 TP        0.1295
      8 S15         -2.7820    0.3650    3.0600 sh        1 TP       -0.2540
      9 H19         -3.5410    0.9790    3.2740 hs        1 TP        0.1908
     10 H29         -0.7870   -0.0430   -0.9380 ha        1 TP        0.1345
     11 H30          0.3730    2.0450   -0.7840 ha        1 TP        0.1336
     12 H31         -0.0920    3.5780    0.7810 ha        1 TP        0.1332
     13 H32         -2.3790   -0.9160    0.9010 ha        1 TP        0.1432
@<TRIPOS>BOND
     1    1    2 ar
     2    1    3 ar
     3    1   13 1
     4    2    4 ar
     5    2   10 1
     6    3    5 ar
     7    3    8 1
     8    4    6 ar
     9    4   11 1
    10    5    6 ar
    11    5    7 1
    12    6   12 1
    13    8    9 1
@<TRIPOS>SUBSTRUCTURE
     1 TP          1 TEMP              0 ****  ****    0 ROOT
```

This yields the *frcmod* file:

```
remark goes here
MASS

BOND

ANGLE
ca-ca-ha   50.000     120.000   same as ca-ca-hc

DIHE

IMPROPER
ca-ca-ca-ha         1.1          180.0         2.0     Using default value
ca-ca-ca-sh         1.1          180.0         2.0     Using default value

NONBON
```

In this case, there was one missing angle parameter from the *gaff.dat* file, and it was determined by analogy to a similar, known, parameter. The missing improper dihedral term was assigned a default value. (As *gaff.dat* continues to be developed, there should be fewer and fewer missing parameters to be estimated by *parmchk*. The above example is actually drawn from Amber 7; in Amber 8, all of the needed parameters are in *gaff.dat*, as can be seen in $AMBERHOME/test/antechamber/tp.) In some cases, *parmchk* may be unable to make a good estimate; it will then insert a placeholder (with zeros everywhere) into the *frcmod* file, with the comment "ATTN: needs revision". After manually editing this to take care of the elements that "need revision", you are ready to read this residue into LEaP, either as a residue on its own, or as part of a larger system. The following LEaP input file (*leap.in*) will just create a system with thiophenol in it:

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

You can read this into LEaP as follows:

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

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

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

descriptions that are generally compatible with the AMBER macromolecular force fields.

The procedure above only works as it stands for neutral molecules. If your molecule is charged, you need to set the *-nc* flag in the initial *antechamber* run. Also note that this procedure depends heavily upon the initial 3D structure: it must have all hydrogens present, and the charges computed are those for the conformation you provide, after minimization in the AM1 Hamiltonian. In fact, this means that you must have an reasonable all-atom initial model of your molecule (so that it can be minimized with the AM1 Hamiltonian), and you must specify what its net charge is. The system should really be a closed-shell molecule, since all of the atom-typing rules assume this implicitly.

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

(1) For the input molecules, make sure there are no open valences and the sturctures are reasonable.

(2) Failures are most likely produced when *antechamber* infers an incorrect connectivity. In such cases, you can revise by hand the connectivity information in "ac" or "mol2" files. Systematic errors could be corrected by revising the parameters in CONNECT.TPL in $AMBERHOME/dat/antechamber.

(3) It is a good idea to check the intermediate files in case of a program failure, and you can run separate programs one by one. Use the "-s 2" flag to *antechamber* to see details of what it is doing.

(4) Please visit *www.amber.ucsf.edu/antechamber.html* to obtain the latest information about *antechamber* development and to download the latest GAFF parameters. Please report program failures to Junmei Wang at <jwang@encysive.com>.

2.3. Programs called by antechamber

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

2.3.1. atomtype

Atomtype reads in an ac file and assigns the atom types. You may find the default definition files in $AMBERHOME/dat/antechamber: ATOMTYPE_AMBER.DEF (AMBER), ATOMTYPE_GFF.DEF (general AMBER force field). ATOMTYPE_GFF.DEF is the default definition file.

```
Usage: atomtype -i input file name
                -o output file name (ac)
                -f input file format(ac (the default) or mol2)
                -p amber or gaff or bcc or gas, it is supressed by "-d" option
                -d atom type defination file, optional
```

Example:

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

This command assigns atom types for *sustiva_resp.ac* with amber atom type definitions. The

output file name is *sustiva_resp_at.ac*

2.3.2. am1bcc

Am1bcc first reads in an ac or mol2 file with or without assigned AM1-BCC atom types and bond types. Then the bcc parameter file (the default, BCCPARM.DAT is in $AMBERHOME/dat/antechamber) is read in. An ac file with AM1-BCC charges [2,3] is written out. Be sure the charges in the input ac file are AM1-Mulliken charges.

```
Usage: am1bcc -i input file name in ac format
              -o output file name
              -f output file format(pdb or ac, optional, default is ac)
              -p bcc parm file name (optional))
              -j atom and bond type judge option, default is 0)
                 0: No judgement
                 1: Atom type
                 2: Full bond type
                 3: Partial bond type
                 4: Atom and full bond type
                 5: Atom and partial bond type
```

Example:

```
am1bcc -i comp1.ac -o comp1_bcc.ac -f ac -j 4
```

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

2.3.3. bondtype

bondtype is a program to assign the atom types and bond types according to the AM1-BCC definitions (BCCTYPE.DEF in $AMBERHOME/dat/antechamber). This program can read an ac file or mol2 file; the output file is an ac file with predicted atom types and bond types. You can choose to determine to assign atom types or bond types or both. If there is some problem with the assignment of bond types, you will get some warnings and for each problematic bond, a "!!!" is appended at the end of the line. In initial tests, the current version works for most organic molecules (>95% overall and >90% for charged molecules).

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

Example:

```
#! /bin/csh -fv
set mols = /bin/ls*.ac
foreach mol ($mols)
   set mol_dir = $mol:r
   antechamber -i $mol_dir.ac -fi ac -fo ac -o $mol_dir.ac -c mul
   bondtype -i $mol_dir.ac -f ac -o $mol_dir.dat -j full
   am1bcc -i $mol_dir.dat -o $mol_dir_bcc.ac -f ac -j 0
end
exit(0)
```

The above script finds all the files with the extension of "ac", calculates the Mulliken charges using *antechamber*, and predicts the atom and bond types with bondtype. Finally, AM1-BCC charges are generated by running am1bcc to do the bond charge correction.

2.3.4. prepgen

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

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

Examples:

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

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

2.3.5. espgen

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

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

Example:

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

The above command reads in sustiva_g98.out and writes out sustiva.esp, which can be used by the resp program. Note that this program replaces shell scripts formerly found on the AMBER web site that perform equivalent tasks.

2.3.6. respgen

Respgen generates the input files for two-stage resp fitting. The current version only supports single molecule fitting. Atom equivalence is recognized automatically.

```
Usage: respgen -i input file name(ac)
               -o output file name
               -f output file format (resp1 or resp2)
                  resp1 - first stage resp fitting
                  resp2 - second stage resp fitting
```

Example:

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

The above commands first generate the input files (sustiva.respin1 and sustiva.respin2) for resp fitting, then do two-stage resp fitting and finally use *antechamber* to read in the resp charges and write out an ac file − sustiva_resp.ac.

2.4. Miscellaneous programs

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

2.4.1. crdgrow

Crdgrow reads an incomplete pdb file (at least three atoms in this file) and a prep input file, and then generates a complete pdb file. It can be used to do residue mutation. For example, if you want to change one protein residue to another one, you can just keep the mainchain atoms in a pdb file and read in the prep input file of the residue to be changed, and crdgrow will generate the coordinates of the missing atoms.

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

```
                    -p prepin file name
                    -f prepin file format: prepi (the default)
```

Example:

```
crdgrow -i ref.pdb -o new.pdb -p sustiva_int.prep
```

This command reads in ref.pdb (only four atoms) and prep input file sustiva_int.prep, then generates the coordinates of the missing atoms and writes out a pdb file (new.pdb).

2.4.2. parmcal

Parmcal is an interactive program to calculate the bond length and bond angle parameters, according to the rules outlined in [1].

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

2.4.3. database

Database reads in a multiple sdf or mol2 file and a description file to run a set of commands for each record sequentially. The commands are defined in the description file.

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

Example:

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

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

2.4.4. translate

Translate performs translation or rotation or least-squared fitting on a file in either pdb, ac or mol2 format. There are five "command" modes, which are

center	Move an atom (specified by -a1) or the geometric center of the molecule to the cartesian coordinate orgin.
translate	Translate the molecule; the X-vector, Y-vector and Z-vector are specified by -vx, -vy, -vz, respectively.
rotate1	Rotate the molecule by an amount (in degrees, specified by -d) along the axis defined by two atoms (specified by -a1 and -a2).
rotate2	Rotate the molecule by an amount (in degrees, specified by -d) along the axis defined by two points (specified by ((-x1, -y1, -z1) and (-x2, -y2, -z2)).

match Do a least-squares fit, wth the reference molecule being read in with "-r" flag.

```
Usage:
 translate -i  input file name (pdb, ac or mol2)
           -o  output file name
           -r  reference file name
           -f  file format
           -c  command (center, translate, rotate1, rotate2, match)
               center:     need -a1;
               translate:  need -vx, -vy and -vz;
               rotate1:    need -a1, -a2 and -d;
               rotate2:    need -x1, -y1, -z1, -x2, -y2, -z2 and -d;
               match:      need -r;
           -d  degree to be rotated
           -vx x vector
           -vy y vector
           -vz z vector
           -a1 id of atom 1 (0 = coordinate center)
           -a2 id of atom 2
           -x1 coord x for point 1
           -y1 coord y for point 1
           -z1 coord z for point 1
           -x2 coord x for point 2
           -y2 coord y for point 2
           -z2 coord z for point 2
```

Examples:

```
translate -i nad.mol2 -f mol2 -o nad_trans.mol2 -c center -a1 0
translate -i nad.mol2 -f mol2 -o nad_match.mol2 -c match -r nad_ref.mol2
translate -i nad.mol2 -f mol2 -o nad_rotate.mol2 -c rotate2 \
            -x1 0.0 -y1 0.0 -z1 0.0 -x2 1.0 -y2 0.0 -z2 0.0 -d 90.0
```

The first command translates the coordinate center of the molecule to the origin; the second command performs least-squared fitting using nad_ref.mol2 as the refereneral molecule; the last command rotates the molecule 90 degrees about the X-axis.

3. LEaP

3.1. Introduction

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

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

This is a simplified version of the LEaP documentation. It does not describe elements that are not supported by NAB; these include the graphical user interface, commands related to periodic boundary simulations, and items related to perturbation calculations. A more complete account can be had in the the *Amber Users' Manual*, which is available at `http://amber.scripps.edu`.

3.2. Concepts

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

3.2.1. Commands

A researcher uses LEaP by entering commands that manipulate objects. An object is just a basic building block; some examples of objects are ATOMs, RESIDUEs, UNITs, and 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.

3.2.2. Variables

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

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

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

3.2.3. Objects

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

3.2.3.1. NUMBERs

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

3.2.3.2. 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"
```

3.2.3.3. 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.

3.2.3.4. 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.

3.2.3.5. 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.

3.2.3.6. 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.

3.2.3.7. 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.

3.2.3.8. 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.

3.3. 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

3.3.1. Building a Molecule For Molecular Mechanics

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

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

(2) The molecule must be constructed within LEaP;

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

The most typical command sequence is the following:

```
source leaprc.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.)

3.3.2. 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 }
```

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

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

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

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

3.3.3. Nucleic Acid Residues

The 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."

3.3.4. Miscellaneous Residues

Miscellaneous Residue	unit/residue name
TIP3P water molecule	TP3
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.

3.4. Commands

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

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

3.4.1. add

`add a b`

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

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

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

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

3.4.2. 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.

3.4.3. 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.

3.4.4. 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.

3.4.5. 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.

3.4.6. 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.*

3.4.7. 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 }
            :  :
> }
```

3.4.8. 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.

3.4.9. 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
```

3.4.10. 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
```

3.4.11. check

```
check unit [ parms ]
```

```
UNIT    unit
PARMSET parms
```

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

- 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.
```

3.4.12. 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>
```

3.4.13. 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.

3.4.14. 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.)

3.4.15. createParmset

```
variable = createParmset  name
```

```
PARMSET variable
STRING  name
```

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

```
> newparms = createParmset pertParms
```

3.4.16. createResidue

```
variable = createResidue  name
```

```
RESIDUE variable
STRING  name
```

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

3.4.17. 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.)

3.4.18. 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.

3.4.19. 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
```

3.4.20. 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.

3.4.21. 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.)

3.4.22. 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.

3.4.23. 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.

3.4.24. list

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

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

3.4.25. loadAmberParams

```
variable = loadAmberParams  filename
```

```
PARMSET variable
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.
```

3.4.26. 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
```

3.4.27. 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
```

3.4.28. loadMol2

`variable = loadMol2 filename`

```
STRING  filename
object  variable
```

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

3.4.29. 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.

3.4.30. 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".

3.4.31. 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
```

3.4.32. 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
```

3.4.33. quit

Quit the LEaP program.

3.4.34. 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>
```

3.4.35. saveAmberParm

```
saveAmberParm unit topologyfilename coordinatefilename
```

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

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

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

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

3.4.36. 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.)

3.4.37. 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
```

3.4.38. 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>
```

3.4.39. 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.

3.4.40. 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.
```

3.4.41. 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.
```

3.4.42. 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.

3.4.43. 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  }
 }
```

3.4.44. 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 }
```

3.4.45. 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
```

3.4.46. 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.)

4. References

1. J. Wang, R.M. Wolf, J.W. Caldwell, P.A. Kollamn & D.A. Case. Development and testing of a general Amber force field. *J. Comput. Chem.* **25,** 1157-1174 (2004).

2. A. Jakalian, B.L. Bush, D.B. Jack & C.I. Bayly. Fast, Efficient Generation of High-Quality Atomic Charges. AM1-BCC Model: I. Method. *J. Comput. Chem.* **21,** 132-146 (2000).

3. A. Jakalian, D.B. Jack & C.I. Bayly. Fast, Efficient Generation of High-Quality Atomic Charges. AM1-BCC Model:. *J. Comput. Chem.* **23,** 1623-1641 (2002). I. Parameterization and Validation,

4. J. Wang & P.A. Kollman. Automatic Parameterization of Force Field by Systematic Search and Genetic Algorithms. *J. Comput. Chem.* **22,** 1219-1228 (2001).

5. Index

V

Z