MODELLER
A Program for Protein Structure Modeling
Release 10.7, r13093

Andrej Šali

with help from

Ben Webb, M.S. Madhusudhan, Min-Yi Shen, Guangqiang Dong,
Marc A. Martı-Renom, Narayanan Eswar, Frank Alber, Maya Topf,
Baldomero Oliva, András Fiser, Roberto Sánchez, Bozidar Yerkovich,
Azat Badretdinov, Francisco Melo, John P. Overington, and Eric Feyfant

email: `modeller-care AT salilab.org`
URL `https://salilab.org/modeller/`

2025/05/28

Contents

List of Figures

List of Tables

.

Copyright notice

The source code for glib can be downloaded from http://www.gtk.org/download/. libiconv and libintl source code can be downloaded from the GNU website.

CMP is included in Modeller. The source code can be obtained from https://github.com/camgunz/cmp. It is copyrighted under the MIT License with the conditions below.

HDF5 is included in Modeller, with the conditions below.

Acknowledgments

I am grateful to my PhD supervisor Professor Tom L. Blundell in whose laboratory at Birkbeck College the program was initiated.

I would also like to thank Professor Martin Karplus who allowed some of the data in the CHARMM topology and library files to be used with MODELLER.

I am in debt to the MODELLER users for their constructive criticisms and suggestions.

MODELLER was written when at

1989–1990:	Department of Crystallography, Birkbeck College University of London, Malet St, London WC1E 7HX, UK.
1990–1991:	ICRF Unit of Structural Molecular Biology, Birkbeck College Malet St, London WC1E 7HX, UK.
1991–1994:	Department of Chemistry, Harvard University 12 Oxford St, Cambridge, MA 02138, USA.
1995–2003:	The Rockefeller University, 1230 York Ave, New York, NY 10021, USA.
2003–:	University of California, San Francisco, 1700 4th Street, San Francisco, CA 94143, USA.

Chapter 1

Introduction

1.1 What is Modeller?

Modeller is a computer program that models three-dimensional structures of proteins and their assemblies by satisfaction of spatial restraints.

Modeller is most frequently used for homology or comparative protein structure modeling: The user provides an alignment of a sequence to be modeled with known related structures and Modeller will automatically calculate a model with all non-hydrogen atoms (these structures are often homologs, but certainly don't have to be, hence the term "comparative" modeling).

More generally, the input to the program are restraints on the spatial structure of the amino acid sequence(s) and ligands to be modeled. The output is a 3D structure that satisfies these restraints as well as possible. Restraints can in principle be derived from a number of different sources. These include related protein structures (comparative modeling), NMR experiments (NMR refinement), rules of secondary structure packing (combinatorial modeling), cross-linking experiments, fluorescence spectroscopy, image reconstruction in electron microscopy, site-directed mutagenesis, intuition, residue–residue and atom–atom potentials of mean force, *etc.* The restraints can operate on distances, angles, dihedral angles, pairs of dihedral angles and some other spatial features defined by atoms or pseudo atoms. Presently, Modeller automatically derives the restraints only from the known related structures and their alignment with the target sequence.

A 3D model is obtained by optimization of a molecular probability density function (pdf). The molecular pdf for comparative modeling is optimized with the variable target function procedure in Cartesian space that employs methods of conjugate gradients and molecular dynamics with simulated annealing.

Modeller can also perform multiple comparison of protein sequences and/or structures, clustering of proteins, and searching of sequence databases. The program is used with a scripting language and does not include any graphics. It is written in standard Fortran 90 and will run on Unix, Windows, or Mac computers.

1.2 Modeller bibliography

In your publications using Modeller, please quote

A. Šali and T. L. Blundell. Comparative protein modelling by satisfaction of spatial restraints. *J. Mol. Biol.* **234**, 779–815, 1993.

More information about the methods implemented in Modeller, their use, applications, and limitations can be found in the papers listed on our web site at `https://salilab.org/publications/`. Here is a subset of these publications:

1. A. Šali and T. L. Blundell. Comparative protein modelling by satisfaction of spatial restraints. *J. Mol. Biol.* **234**, 779–815, 1993.
2. A. Šali and J. P. Overington. Derivation of rules for comparative protein modeling from a database of protein structure alignments. *Protein Science* **3**, 1582–1596, 1994.
3. R. Sánchez and A. Šali. Comparative protein structure modeling: Introduction and practical examples with Modeller. In *Protein Structure Prediction: Methods and Protocols,* D.M. Webster, editor, 97–129. Humana Press, 2000.
4. M. A. Martí-Renom, A. Stuart, A. Fiser, R. Sánchez, F. Melo and A. Šali. Comparative protein structure modeling of genes and genomes. *Ann. Rev. Biophys. Biomolec. Struct.* **29**, 291–325, 2000.
5. A. Fiser, R. K. G. Do and A. Šali. Modeling of loops in protein structures. *Protein Science* **9**, 1753–1773, 2000.
6. F. Melo, R. Sánchez, A. Šali. Statistical potentials for fold assessment. *Protein Science* **11**, 430–448, 2002.
7. M. A. Martí-Renom, B. Yerkovich, and A. Šali. Comparative protein structure prediction. John Wiley & Sons, Inc. Current Protocols in Protein Science 1, 2.9.1 – 2.9.22, 2002.
8. U. Pieper, N. Eswar, A. C. Stuart, V. A. Ilyin and A. Šali. MODBASE, a database of annotated comparative protein structure models. *Nucleic Acids Research* **30**, 255–259, 2002.
9. A. Fiser and A. Šali. MODELLER: generation and refinement of homology-based protein structure models. In *Methods in Enzymology*, C.W. Carter and R.M. Sweet, eds. Academic Press, San Diego, **374**, 463–493, 2003.
10. N. Eswar, B. John, N. Mirkovic, A. Fiser, V. A. Ilyin, U. Pieper, A. C. Stuart, M. A. Martí-Renom, M. S. Madhusudhan, B. Yerkovich and A. Šali. Tools for comparative protein structure modeling and analysis. *Nucleic Acids Research* **31**, 3375–3380, 2003.

1.3 Obtaining and installing the program

This manual assumes that you already have Modeller installed on your computer. Please refer to the release notes on the Modeller web site for information on obtaining and installing the program.

1.4 Bug reports

Please report Modeller bugs by e-mail to the Modeller developers (for contact information, see `https://salilab.org/modeller/contact.html`), or, if you suspect your inputs may be faulty (and the files are not confidential) to the users' mailing list (see the same web page).

In order to be able to reproduce the bug, we will need all of your original input files (*e.g.*, script file, alignment, PDBs). In most cases, the full output demonstrating the error(s) you receive is useful too.

Please do not paste your input files directly into your email, but instead put them in a `.zip` or `.tar.gz` file. That way, we can see the exact same files you're using.

1.5 Method for comparative protein structure modeling by Modeller

Modeller implements an automated approach to comparative protein structure modeling by satisfaction of spatial restraints (Figure 1.1) [Šali & Blundell, 1993]. The method and its applications to biological problems are described in detail in references listed in Section 1.2. Briefly, the core modeling procedure begins with an alignment of the sequence to be modeled (target) with related known 3D structures (templates). This alignment is usually the input to the program. The output is a 3D model for the target sequence containing all mainchain and sidechain non-hydrogen atoms. Given an alignment, the model is obtained without any user intervention. First, many distance and dihedral angle restraints on the target sequence are calculated from its alignment with template 3D structures (Figure 1.2). The form of these restraints was obtained from a statistical analysis of the relationships between many pairs of homologous structures. This analysis relied on a database of 105 family alignments that included 416 proteins with known 3D structure [Šali & Overington, 1994]. By scanning the database, tables quantifying various correlations were obtained, such as the correlations between two equivalent $C_\alpha - C_\alpha$ distances, or between equivalent mainchain dihedral angles from two related proteins. These relationships were expressed as conditional probability density functions (pdf's) and can be used directly as spatial restraints. For example, probabilities for different values of the mainchain dihedral angles are calculated from the type of a residue considered, from mainchain conformation of an equivalent residue, and from sequence similarity between the two proteins. Another example is the pdf for a certain C_α–C_α distance given equivalent distances in two related protein structures (Figure 1.2). An important feature of the method is that the spatial restraints are obtained empirically, from a database of protein structure alignments. Next, the spatial restraints and Charmm energy terms enforcing proper stereochemistry [MacKerell *et al.*, 1998] are combined into an objective function. Finally, the model is obtained by optimizing the objective function in Cartesian space. The optimization is carried out by the use of the variable target function method [Braun & Gõ, 1985] employing methods of conjugate gradients and molecular dynamics with simulated annealing (Figure 1.3). Several slightly different models can be calculated by varying the initial structure. The variability among these models can be used to estimate the errors in the corresponding regions of the fold.

There are additional specialized modeling protocols, such as that for the modeling of loops (Section 2.3).

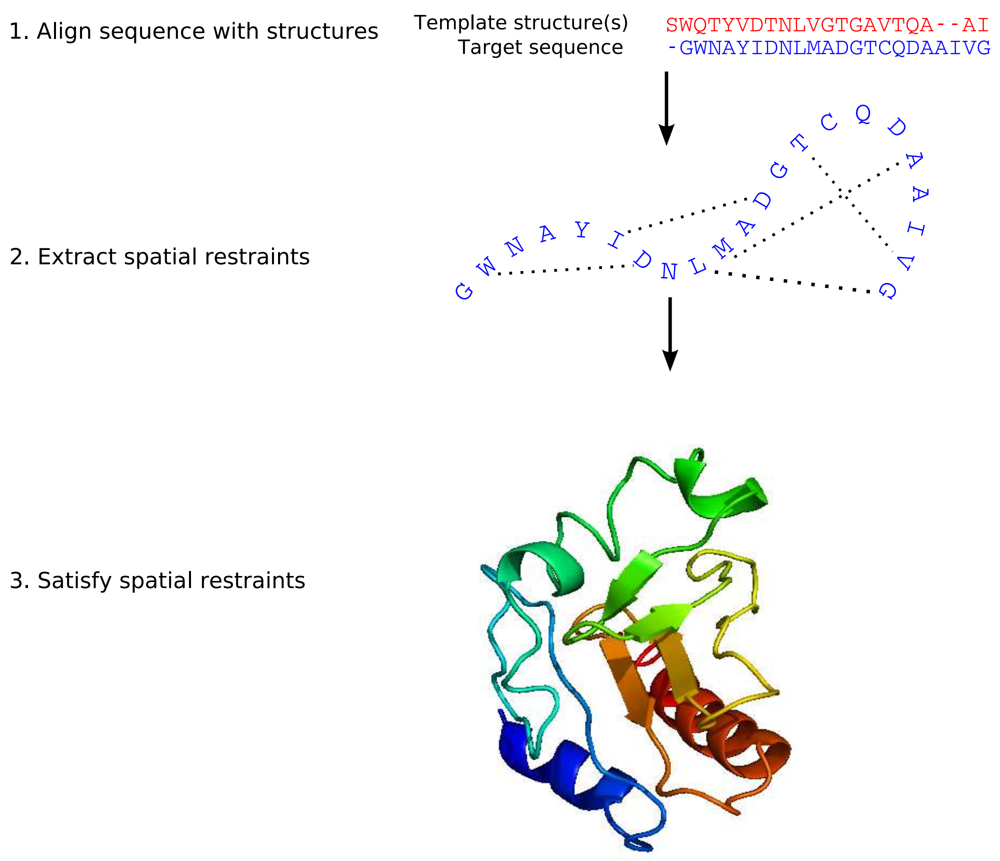
1. Align sequence with structures
Template structure(s)
Target sequence
SWQTYVDTNLVGTGAVTQA--AI
-GWNAYIDNLMADGTCQDAAIVG
2. Extract spatial restraints
G W N A Y I D N L M A D G T C Q D A A I V G
3. Satisfy spatial restraints

Figure 1.1: *Comparative protein modeling by satisfaction of spatial restraints.* First, the known, template 3D structures are aligned with the target sequence to be modeled. Second, spatial features, such as C_α–C_α distances, hydrogen bonds, and mainchain and sidechain dihedral angles, are transferred from the templates to the target. Thus, a number of spatial restraints on its structure are obtained. Third, the 3D model is obtained by satisfying all the restraints as well as possible.

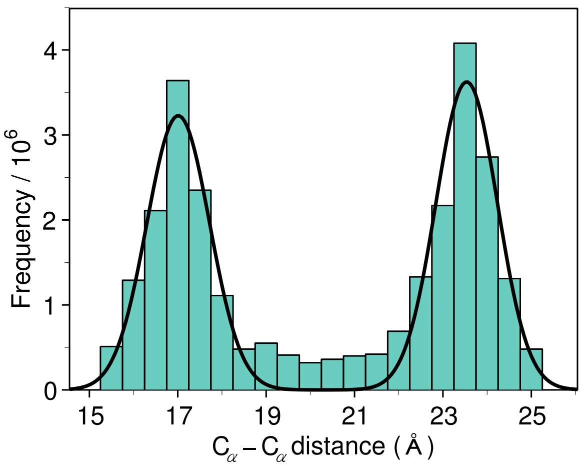

Figure 1.2: *Sample spatial restraint.* A restraint on a given C_α–C_α distance, d, is expressed as a conditional probability density function that depends on two other equivalent distances ($d' = 17.0$ and $d'' = 23.5$): $p(d/d', d'')$. The restraint (continuous line) is obtained by least-squares fitting a sum of two Gaussian functions to the histogram, which in turn is derived from the database of alignments of protein structures. In practice, more complicated restraints are used that depend on additional information, such as similarity between the proteins, solvent accessibility, and distance from a gap in the alignment [Šali & Blundell, 1993].

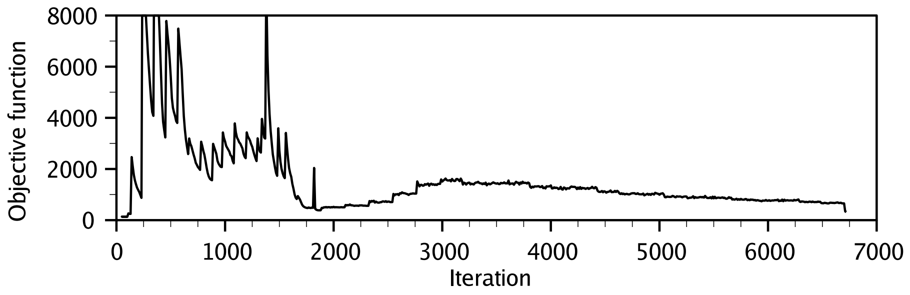

Figure 1.3: *Optimization of the objective function.* Optimization of the objective function (curve) starts with a distorted average of template structures. In this run, the first $\sim 2,000$ iterations correspond to the variable target function method relying on the conjugate gradients technique. This approach first satisfies sequentially local restraints and slowly introduces longer range restraints until the complete objective function is optimized. In the last 4,750 iterations for this model, molecular dynamics with simulated annealing is used to refine the model. Typically, a model is calculated in the order of minutes on a PC workstation.

1.6 Using Modeller for comparative modeling

Simple demonstrations of Modeller in all steps of comparative protein structure modeling, including fold assignment, sequence-structure alignment, model building, and model assessment, can be found in references listed at `https://salilab.org/modeller/documentation.html`. A number of additional tools useful in comparative modeling are listed at `https://salilab.org/bioinformatics_resources.shtml`. Specifically, users have access to ModBase, a comprehensive database of comparative models for all known protein sequences detectably related to at least one known protein structure; ModWeb, a web server for automated comparative protein structure modeling; and ModLoop, a web server for automated modeling of loops in protein structures. For "frequently-asked-questions" (FAQ), see Section 3.1.

The rest of this section is a 'hands on' description of the most basic use of Modeller in comparative modeling, in which the input are Protein Data Bank (PDB) atom files of known protein structures, and their alignment with the target sequence to be modeled, and the output is a model for the target that includes all non-hydrogen atoms. Although Modeller can find template structures as well as calculate sequence and structure alignments, it is better in the difficult cases to identify the templates and prepare the alignment carefully by other means. The alignment can also contain very short segments such as loops, secondary structure motifs, *etc*.

1.6.1 Preparing input files

The sample input files in this tutorial can be found in the `examples/automodel` directory of the Modeller distribution.

There are three kinds of input files: Protein Data Bank atom files with coordinates for the template structures, the alignment file with the alignment of the template structures with the target sequence, and Modeller commands in a script file that instruct Modeller what to do.

Atom files

Each atom file is named `code.atm` where `code` is a short protein code, preferably the PDB code; for example, *Peptococcus aerogenes* ferredoxin would be in a file `1fdx.atm`. If you wish, you can also use file extensions `.pdb` and `.ent` instead of `.atm`. The code must be used as that protein's identifier throughout the modeling.

Alignment file

One of the formats for the alignment file is related to the PIR database format; this is the preferred format for comparative modeling:

```
C; A sample alignment in the PIR format; used in tutorial

>P1;5fd1
structureX:5fd1:1    :A:106  :A:ferredoxin:Azotobacter vinelandii: 1.90: 0.19
AFVVTDNCIKCKYTDCVEVCPVDCFYEGPNFLVIHPDECIDCALCEPECPAQAIFSEDEVPEDMQEFIQLNAELA
EVWPNITEKKDPLPDAEDWDGVKGKLQHLER*

>P1;1fdx
sequence:1fdx:1    :A:54   :A:ferredoxin:Peptococcus aerogenes: 2.00:-1.00
AYVINDSC--IACGACKPECPVNIIQGS--IYAIDADSCIDCGSCASVCPVGAPNPED-----------------
-------------------------------*
```

See Section B.1 for a detailed description of the alignment file format. Influence of the alignment on the quality of the model cannot be overemphasized. To obtain the best possible model, it is important to understand how the alignment is used by Modeller [Šali & Blundell, 1993]. In outline, for the aligned regions, Modeller tries to derive a 3D model for the target sequence that is as close to one or the other of the template structures as possible while also satisfying stereochemical restraints (*e.g.*, bond lengths, angles, non-bonded atom contacts, ...); the inserted regions, which do not have any equivalent segments in any of the templates, are modeled in the context of the whole molecule, but using their sequence alone. This way of deriving a model means that whenever a user aligns a target residue with a template residue, he tells MODELLER to treat the aligned residues as **structurally** equivalent. Command **Alignment.check()** can be used to find some trivial alignment mistakes.

Script file

MODELLER is a command-line only tool, and has no graphical user interface; instead, you must provide it with a script file containing MODELLER commands. This is an ordinary Python script.

If you are not familiar with Python, you can simply adapt one of the many examples in the `examples` directory, or look at the code for the classes used by MODELLER itself, in the `modlib/modeller` directory. Finally, there are many resources for learning Python itself, such as a comprehensive tutorial at https://docs.python.org/3/tutorial/index.html.

A sample script file `model-default.py` to produce one model of sequence `1fdx` from the known structure of `5fd1` and from the alignment between the two sequences is

```
# Comparative modeling by the AutoModel class
from modeller import *              # Load standard Modeller classes
from modeller.automodel import *    # Load the AutoModel class

log.verbose()    # request verbose output
env = Environ()  # create a new MODELLER environment to build this model in

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env,
              alnfile  = 'alignment.ali',     # alignment filename
              knowns   = '5fd1',              # codes of the templates
              sequence = '1fdx')              # code of the target
a.starting_model= 1                 # index of the first model
a.ending_model  = 1                 # index of the last model
                                    # (determines how many models to calculate)
a.make()                            # do the actual comparative modeling
```

See Chapter 2 for more information about the `AutoModel` class, and a more detailed explanation of what this script does.

1.6.2 Running MODELLER

To run MODELLER with the script file `model-default.py` above, do the following:

1. Open a command line prompt:
   - **On Linux/Unix:** `ssh` to the machine, or open an xterm or GNOME Terminal.
   - **On Windows:** Click on the 'Modeller' link on your Start Menu. This will give you a Windows Command Prompt, set up for you to run MODELLER.
   - **On Mac OS X:** `ssh` to the machine, or open the Terminal application.
2. Change to the directory containing the script and alignment files you created earlier, using the `'cd'` command.
3. Run MODELLER itself by typing the following at the command prompt:
   ```
   python3 model-default.py > model-default.log
   ```

(Note that if you don't have Python installed on your machine, you can run the last step by typing '`mod10.7 model-default.py`' instead.)

A number of intermediary files are created as the program proceeds. After about 10 seconds on a modern PC, the final `1fdx` model is written to file `1fdx.B99990001.pdb`. Examine the `model-default.log` file for information about the run. In particular, one should always check the output of the **Alignment.check()** command, which you can find by searching for '`check_a`'. Also, check for warning and error messages by searching for '`W>`' and '`E>`', respectively. There should be no error messages; most often, there are some warning messages that can usually be ignored.

(MODELLER should work just like an ordinary Python module without any additional setup on Mac OS, Windows with any version of Python between 2.3 and 3.13, or using the RPM install on Linux with any version of Python between 2.3 and 3.13. On other systems, you may need to set the `PYTHONPATH` and `LD_LIBRARY_PATH` variables, or create symlinks to the relevant directories, so that your system can find the MODELLER Python modules and dynamic libraries, respectively.)

Chapter 2

Automated comparative modeling with AutoModel

The simplest way to build comparative models with MODELLER is to use the `AutoModel` class. This automates many of the steps required for simple modeling, and can be customized for more complex tasks. A related class, `LoopModel`, also allows for refinement of loop regions.

See section A.4 for a flowchart of the comparative modeling procedure, and section A.5 for an overview of the loop modeling algorithm.

This chapter gives an overview of simple applications of the `AutoModel` class. For more detailed information, see the comparative modeling class reference, in Chapter 4, or the comments in the Python files themselves, `modlib/modeller/automodel/automodel.py` and `modlib/modeller/automodel/loopmodel.py`.

2.1 Simple usage

The simple example below constructs a single comparative model for the `1fdx` sequence from the known `5fd1` structure, using `alignment.ali`, a PIR format alignment of `5fd1` and `1fdx`. The final model is written into the PDB file `1fdx.B99990001.pdb`. See Section 1.6.2 for instructions on how to run this script.

Example: examples/automodel/model-default.py

```
# Comparative modeling by the AutoModel class
from modeller import *              # Load standard Modeller classes
from modeller.automodel import *    # Load the AutoModel class

log.verbose()    # request verbose output
env = Environ()  # create a new MODELLER environment to build this model in

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env,
              alnfile  = 'alignment.ali',     # alignment filename
              knowns   = '5fd1',              # codes of the templates
              sequence = '1fdx')              # code of the target
a.starting_model= 1                 # index of the first model
a.ending_model  = 1                 # index of the last model
                                    # (determines how many models to calculate)
a.make()                            # do the actual comparative modeling
```

Example: examples/automodel/alignment.ali

```
C; A sample alignment in the PIR format; used in tutorial

>P1;5fd1
structureX:5fd1:1    :A:106  :A:ferredoxin:Azotobacter vinelandii: 1.90: 0.19
AFVVTDNCIKCKYTDCVEVCPVDCFYEGPNFLVIHPDECIDCALCEPECPAQAIFSEDEVPEDMQEFIQLNAELA
EVWPNITEKKDPLPDAEDWDGVKGKLQHLER*

>P1;1fdx
sequence:1fdx:1    :A:54   :A:ferredoxin:Peptococcus aerogenes: 2.00:-1.00
AYVINDSC--IACGACKPECPVNIIQGS--IYAIDADSCIDCGSCASVCPVGAPNPED-----------------
-------------------------------*
```

Stepping through the script, first we load the AutoModel class, using standard Python syntax to load a module. Next, we request verbose output (see Section 6.31) so that we can more easily spot errors. We then create an **Environ()** object (see Section 6.2) and call it env. This object holds the MODELLER 'environment', which comprises default values for many parameters, as well as the libraries used for comparative modeling (topology, parameters, dihedral classes, *etc*). An Environ object is needed to create most other MODELLER objects, but you can call it whatever you like (it doesn't have to be called env).

Once we have the Environ object, we can set some global parameters. In this case, we set IOData.atom_files_directory to set the directories to look for PDB files in.

Next, we create an AutoModel object, tell it which PIR alignment file to use, and which sequences are templates and which one we want to build a model for, and call it a. This doesn't actually build any models, but creates the object, ready to be tweaked for our purposes. In this case, we simply tell it to build a single model, by setting both AutoModel.starting_model and AutoModel.ending_model to 1. Finally, we actually build the model by running **AutoModel.make()**.

2.2 More advanced usage

2.2.1 Including water molecules, HETATM residues, and hydrogen atoms

If your template contains a ligand or other non-protein residues (e.g. DNA or RNA, or anything marked as HETATM in the PDB file) then MODELLERcan transfer this into your generated model. This is done by using the BLK ('.') residue type in your alignment (both in the template(s) *and* the model sequence) to copy the ligand residue(s) as a rigid body into the model. In most cases, you should also set env.io.hetatm to True, which instructs MODELLER to read HETATM records from your template PDB files; by default all HETATM records are ignored.

Example: examples/automodel/model-ligand.py

```
# Comparative modeling with ligand transfer from the template
from modeller import *              # Load standard Modeller classes
from modeller.automodel import *    # Load the AutoModel class

log.verbose()    # request verbose output
env = Environ()  # create a new MODELLER environment to build this model in

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

# Read in HETATM records from template PDBs
env.io.hetatm = True

a = AutoModel(env,
              alnfile  = 'align-ligand.ali',  # alignment filename
              knowns   = '5fd1',              # codes of the templates
              sequence = '1fdx')              # code of the target
a.starting_model= 4                 # index of the first model
a.ending_model  = 4                 # index of the last model
                                    # (determines how many models to calculate)
a.make()                            # do the actual comparative modeling
```

Example: examples/automodel/align-ligand.ali

```
C; Similar to alignment.ali, but with ligands included

>P1;5fd1
structureX:5fd1:1    :A:108  :A:ferredoxin:Azotobacter vinelandii: 1.90: 0.19
AFVVTDNCIKCKYTDCVEVCPVDCFYEGPNFLVIHPDECIDCALCEPECPAQAIFSEDEVPEDMQEFIQLNAELA
EVWPNITEKKDPLPDAEDWDGVKGKLQHLER..*

>P1;1fdx
sequence:1fdx:1    :A:56   :A:ferredoxin:Peptococcus aerogenes: 2.00:-1.00
AYVINDSC--IACGACKPECPVNIIQGS--IYAIDADSCIDCGSCASVCPVGAPNPED-----------------
-------------------------------..*
```

Note that by turning on env.io.hetatm, *all* HETATM records are read from your templates, so all of these must be listed in your alignment. Use a single '.' character for each HETATM residue in the *template* sequence in your alignment.[1] MODELLER always reads PDB residues in the order they're written in the PDB file, so if you have a ligand at the end of PDB file, put the '.' residue at the end of the sequence in the alignment too. You will also need to modify the residue range in the alignment header to tell MODELLER to read the ligands from the PDB file - in this case the range is changed from 106:A (in Section 2.1) to 108:A, as the two residues are numbered 107 and 108 in the A chain. You will often see a chain break ('/') immediately preceding '.' residues in example alignments. That's only necessary if you want to force the ligands to have a different chain ID to the amino acids. (If you want them in the same chain, leave out the chain break.)

To get the ligand into your model, you must align a residue in the model with the desired residue in the template. Use a single '.' residue in your *model* sequence in your alignment for each ligand you want in the model. This must be aligned with a suitable ligand in the template sequence. If you have extra HETATM ligands in the template which you *don't* want in the model, simply align them with a gap ('-') in the model sequence. If you have multiple templates, you can copy ligands from any suitable template — just align the '.' residue in the model with the desired template sequence ligand.

`AutoModel` builds restraints on these ligands to keep their geometry and environment reasonably similar to the template, by restraining some intra-ligand, inter-ligand, and ligand-protein distances to their template values. See **AutoModel.nonstd_restraints()** for more information.

You can also treat ligands flexibly by defining topology and parameter information. See section 5.2.1 for more information, and the example in the advanced modeling tutorial, at https://salilab.org/modeller/tutorial/advanced.html.

If you want to add ligands to your model which are not present in your template, you will need to do some docking studies, which are beyond the scope of the MODELLER program.

To read in water residues, set env.io.water to `True` and use the 'w' residue type in your alignment.

To read in hydrogen atoms, set env.io.hydrogen to `True`. This is not generally necessary, as if you want to build an all hydrogen model, it is easiest just to use the `AllHModel` class, which turns this on for you automatically; see section 2.2.5.

[1] If the residue type is defined in `'modlib/restyp.lib'` you can use the 1-letter code that is specified there, but if in doubt use '.', since that matches everything.

2.2.2 Changing the default optimization and refinement protocol

See Section A.4 for a detailed description of the optimization and refinement protocol used by `AutoModel`. To summarize, each model is first optimized with the variable target function method (VTFM) with conjugate gradients (CG), and is then refined using molecular dynamics (MD) with simulated annealing (SA) [Šali & Blundell, 1993]. Most of the time (70%) is spent on the MD&SA part. Our experience is that when MD&SA are used, if there are violations in the best of the 10 models, they probably come from an alignment error, not an optimizer failure (if there are no insertions longer than approximately 15 residues).

The VTFM step can be tuned by adjusting AutoModel.library_schedule, AutoModel.max_var_iterations, and AutoModel.max_molpdf.

The MD&SA step can be tuned by adjusting AutoModel.md_level.

The whole optimization can be repeated multiple times if desired (by default it is run only once) by adjusting AutoModel.repeat_optimization.

The energy function used in both VTFM and MD&SA can be scaled by setting Environ.schedule_scale. (Note that for VTFM, the function is additionally scaled by the factors set in AutoModel.library_schedule.)

Example: examples/automodel/model-changeopt.py

```
# Example of changing the default optimization schedule
from modeller import *
from modeller.automodel import *

log.verbose()
env = Environ()

# Give less weight to all soft-sphere restraints:
env.schedule_scale = physical.Values(default=1.0, soft_sphere=0.7)
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env, alnfile='alignment.ali', knowns='5fd1', sequence='1fdx')
a.starting_model = a.ending_model = 1

# Very thorough VTFM optimization:
a.library_schedule = autosched.slow
a.max_var_iterations = 300

# Thorough MD optimization:
a.md_level = refine.slow

# Repeat the whole cycle 2 times and do not stop unless obj.func. > 1E6
a.repeat_optimization = 2
a.max_molpdf = 1e6

a.make()
```

2.2.3 Getting a very fast and approximate model

To get an approximate model very quickly (to get a rough idea of what it looks like, or to confirm that the alignment is reasonable) call the **AutoModel.very_fast()** method before **AutoModel.make()**. This uses only a very limited amount of variable target function optimization with conjugate gradients, and thus is roughly 3 times faster than the default procedure.

Note that no randomization of the starting structure is done in this case, so only a single model can be produced.

This example also demonstrates the use of the assess_methods keyword, to request model assessment. In this case the GA341 method is requested. See section 4.1.1.

Example: examples/automodel/model-fast.py

```
# Very fast comparative modeling by the AutoModel class
from modeller import *
from modeller.automodel import *    # Load the AutoModel class
#from modeller import soap_protein_od

log.verbose()
env = Environ()
# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env,
              alnfile='alignment.ali',      # alignment filename
              knowns='5fd1',                # codes of the templates
              sequence='1fdx',              # code of the target
              assess_methods=assess.GA341)  # request GA341 model assessment
#             assess_methods=(assess.GA341, assess.DOPE))  # GA341 and DOPE
#             assess_methods=soap_protein_od.Scorer())  # assess with SOAP

a.very_fast()                       # prepare for extremely fast optimization

a.starting_model = 2
a.ending_model = 2
a.final_malign3d = True

a.make()                            # make the comparative model
```

2.2.4 Building a model from multiple templates

It is straightforward a to build a model using information from multiple templates — simply provide an alignment between all of the templates and your target sequence, and list all of the templates in the knowns argument, as demonstrated below. MODELLER will automatically combine the templates; there is no need to superpose the structures (although you can request that this is done by setting AutoModel.initial_malign3d).

Example: examples/automodel/model-multiple.py

```
# Comparative modeling with multiple templates
from modeller import *              # Load standard Modeller classes
from modeller.automodel import *    # Load the AutoModel class

log.verbose()    # request verbose output
env = Environ()  # create a new MODELLER environment to build this model in

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env,
              alnfile  = 'align-multiple.ali', # alignment filename
              knowns   = ('5fd1', '1bqx'),     # codes of the templates
              sequence = '1fdx')               # code of the target
```

```
a.starting_model= 1                 # index of the first model
a.ending_model  = 1                 # index of the last model
                                    # (determines how many models to calculate)
a.make()                            # do the actual comparative modeling
```

Example: examples/automodel/align-multiple.ali

```
C; A multiple alignment in the PIR format; used in tutorial

>P1;5fd1
structureX:5fd1:1    :A:106  :A:ferredoxin:Azotobacter vinelandii: 1.90: 0.19
AFVVTDNCIKCKYTDCVEVCPVDCFYEGPNFLVIHPDECIDCALCEPECPAQAIFSEDEVPEDMQEFIQLNAELA
EVWPNITEKKDPLPDAEDWDGVKGKLQHLER*

>P1;1bqx
structureN:1bqx:   1 :A: 77  :A:ferredoxin:Bacillus schlegelii:-1.00:-1.00
AYVITEPCIGTKCASCVEVCPVDCIHEGEDQYYIDPDVCIDCGACEAVCPVSAIYHEDFVPEEWKSYIQKNRDFF
KK-----------------------------*

>P1;1fdx
sequence:1fdx:1    : :54   : :ferredoxin:Peptococcus aerogenes: 2.00:-1.00
AYVINDSC--IACGACKPECPVNIIQGS--IYAIDADSCIDCGSCASVCPVGAPNPED-----------------
-------------------------------*
```

2.2.5 Building an all hydrogen model

This is done by using the AllHModel class rather than AutoModel. Otherwise, operation is identical. Note that the AllHModel class automatically turns on env.io.hydrogen for you and selects the all-atom topology and radii files.

Example: examples/automodel/model-default-allh.py

```
from modeller import *
from modeller.automodel import *

log.verbose()
env = Environ()

env.io.atom_files_directory = ['.', '../atom_files']

a = AllHModel(env, alnfile='alignment.ali', knowns='5fd1', sequence='1fdx')
a.starting_model = a.ending_model = 4

a.make()
```

2.2.6 Refining only part of the model

The AutoModel class contains a **AutoModel.select_atoms()** function which selects the atoms to be moved during optimization. By default, the routine selects all atoms, but you can redefine it to select any subset of atoms, and then only those atoms will be refined. (To redefine the routine, it is necessary to create a 'subclass' of `AutoModel`, here called `MyModel`, which has the modified routine within it. We then use `MyModel` in place of `AutoModel`. The `select_atoms` routine should return a `Selection` object; see Section 6.9 for further information.)

In this particular case, we use the **Model.residue_range()** function to select residues 1 and 2 from the first (A) chain. See Section 6.17.10 for ways to specify residues, and **Selection()** for other examples of selecting atoms or residues. Please note that the residue numbers and chain IDs refer to the built model, *not* to the template(s). This is because template PDB residue numbering can be inconsistent, and in the case where you have two or more templates, residues from different parts of the sequence coming from different templates could have the same number. MODELLER always names the model residues consistently, counting up from 1. Chain IDs A, B, C, *etc* are assigned[2]. If in doubt about residue numbering, first build a model using the simple script in section 2.1, and then look at the final model (or the initial unoptimized `.ini` model) for the residue numbering.

By default, the selected atoms will "feel" the presence of other atoms via all the static and possibly dynamic restraints that include both selected and un-selected atoms. However, you can turn off dynamic interactions between the selected and unselected regions by setting EnergyData.nonbonded_sel_atoms to 2 (by default it is 1).

The difference between this script and the one for loop modeling is that here the selected regions are optimized with the default optimization protocol and the default restraints, which generally include template-derived restraints. In contrast, the loop modeling routine does not use template-dependent restraints, but does a much more thorough optimization.

Example: examples/automodel/model-segment.py

```
# Comparative modeling by the AutoModel class
#
# Demonstrates how to refine only a part of the model.
#
# You may want to use the more exhaustive "loop" modeling routines instead.
#
from modeller import *
from modeller.automodel import *    # Load the AutoModel class

log.verbose()

# Override the 'select_atoms' routine in the 'AutoModel' class:
# (To build an all-hydrogen model, derive from AllHModel rather than AutoModel
# here.)
class MyModel(AutoModel):
    def select_atoms(self):
        # Select residues 1 and 2 in chain A (PDB numbering)
        return Selection(self.residue_range('1:A', '2:A'))

        # Residues 4, 6, 10 in chain A:
        # return Selection(self.residues['4:A'], self.residues['6:A'],
        #                  self.residues['10:A'])

        # All residues except 1-5 in chain A:
        # return Selection(self) - Selection(self.residue_range('1:A', '5:A'))


env = Environ()
# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']
# selected atoms do not feel the neighborhood
env.edat.nonbonded_sel_atoms = 2
```

[2]After uppercase letters A-Z are used, chain IDs 0 through 9 are assigned, then lowercase letters a-z. If your protein contains more than 62 chains, the remaining chains are given no IDs.

```
# Be sure to use 'MyModel' rather than 'AutoModel' here!
a = MyModel(env,
            alnfile  = 'alignment.ali',     # alignment filename
            knowns   = '5fd1',              # codes of the templates
            sequence = '1fdx')              # code of the target

a.starting_model= 3                # index of the first model
a.ending_model  = 3                # index of the last model
                                   # (determines how many models to calculate)
a.make()                           # do comparative modeling
```

2.2.7 Including disulfide bridges

If there is an equivalent disulfide bridge in any of the templates aligned with the target, `AutoModel` will automatically generate appropriate disulfide bond restraints[3] for you (by using the **Model.patch_ss_templates()** command).

Explicit manual restraints can be added by the **Model.patch()** command using the CHARMM topology file DISU patching residue. You must redefine the **AutoModel.special_patches()** routine to add these or other patches.

It is better to use **Model.patch_ss_templates()** rather than **Model.patch()** where possible because the dihedral angles are restrained more precisely by using the templates than by using the general rules of stereochemistry.

Some CHARMM parameter files have a multiple dihedral entry for the disulfide dihedral angle χ_3 that consists of three individual entries with periodicities of 1, 2 and 3. This is why you see three feature restraints for a single disulfide in the output of the **Selection.energy()** command.

Note that the residue numbers that you patch refer to the model, *not* the templates. See Section 2.2.6 for more discussion.

Example: examples/automodel/model-disulfide.py

```
# Comparative modeling by the AutoModel class
from modeller import *              # Load standard Modeller classes
from modeller.automodel import *    # Load the AutoModel class

# Redefine the special_patches routine to include the additional disulfides
# (this routine is empty by default):
class MyModel(AutoModel):
    def special_patches(self, aln):
        # A disulfide between residues 8 and 45 in chain A:
        self.patch(residue_type='DISU', residues=(self.residues['8:A'],
                                                  self.residues['45:A']))

log.verbose()    # request verbose output
env = Environ()  # create a new MODELLER environment to build this model in

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = MyModel(env,
            alnfile  = 'alignment.ali',     # alignment filename
            knowns   = '5fd1',              # codes of the templates
```

[3]The restraints include bond, angle and dihedral angle restraints. The SG — SG atom pair also becomes an excluded atom pair that is not checked for an atom–atom overlap. The χ_i dihedral angle restraints will depend on the conformation of the equivalent disulfides in the template structure, as described in [Šali & Overington, 1994].

```
              sequence = '1fdx')              # code of the target
a.starting_model= 1                   # index of the first model
a.ending_model  = 1                   # index of the last model
                                      # (determines how many models to calculate)
a.make()                              # do the actual comparative modeling
```

2.2.8 Generating new-style PDBx/mmCIF outputs

By default, the models generated are traditional format PDB files. These have the advantage that many viewers and tools that use these files exist. However, PDB files are being phased out in favor of the mmCIF format (also known as PDBx). mmCIF has a number of advantages, one of which is that it can store large structures that would otherwise need to be split between several PDB files. Another is that the file format supports a broader range of metadata than PDB (such as information on the templates and alignment used in the modeling) which can be useful if the models are deposited in a database.

To use mmCIF files as templates, you don't need to do anything special – Modeller will read templates in PDB, mmCIF, or BinaryCIF format.

To have Modeller output models in mmCIF format, simply call **AutoModel.set_output_model_format()**.

Example: examples/automodel/model-cif.py

```
# Comparative modeling by the AutoModel class, generating mmCIF outputs
from modeller import *              # Load standard Modeller classes
from modeller.automodel import *    # Load the AutoModel class

log.verbose()    # request verbose output
env = Environ()  # create a new MODELLER environment to build this model in

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env,
              alnfile  = 'alignment.ali',     # alignment filename
              knowns   = '5fd1',              # codes of the templates
              sequence = '1fdx')              # code of the target
a.starting_model= 1                   # index of the first model
a.ending_model  = 1                   # index of the last model
                                      # (determines how many models to calculate)
a.set_output_model_format("MMCIF")    # request mmCIF rather than PDB outputs
a.make()                              # do the actual comparative modeling
```

2.2.9 Providing your own restraints file

To force `AutoModel` not to construct restraints at all, but to instead use your own restraints file, simply use the `csrfile` keyword when creating the `AutoModel` class, as in the example below. Note that Modeller does only rudimentary checking on your restraints file, so you must be careful that it applies correctly to the generated model.

Example: examples/automodel/model-myrsr.py

```
# Modeling using a provided restraints file (csrfile)
from modeller import *
from modeller.automodel import *    # Load the AutoModel class
```

```
log.verbose()
env = Environ()

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env,
              alnfile  = 'alignment.ali',     # alignment filename
              knowns   = '5fd1',              # codes of the templates
              sequence = '1fdx',              # code of the target
              csrfile  = 'my.rsr')            # use 'my' restraints file
a.starting_model= 1                 # index of the first model
a.ending_model  = 1                 # index of the last model
                                    # (determines how many models to calculate)
a.make()                            # do comparative modeling
```

2.2.10 Using your own initial model

Normally, `AutoModel` generates an initial model by transferring coordinates from the templates. However, if you have a prepared PDB file containing an initial model, you can have `AutoModel` use this instead with the `inifile` keyword, as in the example below. (This automatically sets AutoModel.generate_method to `generate.read_xyz` for you, which is necessary for this to work.) This can be useful if the default initial model (`.ini` file) is so bad that the optimizer cannot efficiently optimize it. Of course, the primary sequence of this structure must match the target's exactly.

Note that when the initial model file is read, the range of residues to read from the PDB file is taken from the alignment file header for the sequence. Therefore, you should set that range accordingly (in the example below, the header for the `1fdx` sequence `alignment.ali` is set to instruct MODELLER to read residues 1 through 54 from the 'A' chain).

Example: examples/automodel/model-myini.py

```
# Comparative using a provided initial structure file (inifile)
from modeller import *
from modeller.automodel import *    # Load the AutoModel class

log.verbose()
env = Environ()

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env,
              alnfile  = 'alignment.ali',     # alignment filename
              knowns   = '5fd1',              # codes of the templates
              sequence = '1fdx',              # code of the target
              inifile  = 'my-initial.pdb')    # use 'my' initial structure
a.starting_model= 1                 # index of the first model
a.ending_model  = 1                 # index of the last model
                                    # (determines how many models to calculate)
a.make()                            # do comparative modeling
```

2.2.11 Adding additional restraints to the defaults

You can add your own restraints to the restraints file, with the homology-derived restraints, by redefining the **AutoModel.special_restraints()** routine (by default it does nothing). This can be used, for example, to add information from NMR experiments or to add regions of known secondary structure. Symmetry restraints, excluded pairs, or rigid body definitions can also be added in this routine (see Section 2.2.12 for a symmetry example). The example below enforces an additional restraint on a single CA-CA distance, adds some known secondary structure, and shows how to add restraints from a file. (See Section 5.3 for further information on how to specify restraints, and Section 6.8 for details on secondary structure restraints.)

Note that the residue numbers for any restraints refer to the model, *not* the templates. See Section 2.2.6 for more discussion.

Example: examples/automodel/model-addrsr.py

```
# Addition of restraints to the default ones
from modeller import *
from modeller.automodel import *    # Load the AutoModel class

log.verbose()
env = Environ()

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

class MyModel(AutoModel):
    def special_restraints(self, aln):
        rsr = self.restraints
        at = self.atoms
#       Add some restraints from a file:
#       rsr.append(file='my_rsrs1.rsr')

#       Residues 20 through 30 should be an alpha helix:
        rsr.add(secondary_structure.Alpha(self.residue_range('20:A', '30:A')))
#       Two beta-strands:
        rsr.add(secondary_structure.Strand(self.residue_range('1:A', '6:A')))
        rsr.add(secondary_structure.Strand(self.residue_range('9:A', '14:A')))
#       An anti-parallel sheet composed of the two strands:
        rsr.add(secondary_structure.Sheet(at['N:1:A'], at['O:14:A'],
                                          sheet_h_bonds=-5))
#       Use the following instead for a *parallel* sheet:
#       rsr.add(secondary_structure.Sheet(at['N:1:A'], at['O:9:A'],
#                                         sheet_h_bonds=5))

#       Restrain the specified CA-CA distance to 10 angstroms (st. dev.=0.1)
#       Use a harmonic potential and X-Y distance group.
        rsr.add(forms.Gaussian(group=physical.xy_distance,
                               feature=features.Distance(at['CA:35:A'],
                                                         at['CA:40:A']),
                               mean=10.0, stdev=0.1))

a = MyModel(env,
            alnfile  = 'alignment.ali',     # alignment filename
            knowns   = '5fd1',              # codes of the templates
            sequence = '1fdx')              # code of the target
a.starting_model= 1                 # index of the first model
a.ending_model  = 1                 # index of the last model
```

```
                                    # (determines how many models to calculate)
a.make()                            # do comparative modeling
```

2.2.12 Building multi-chain models

Modeller can build models of multi-chain proteins in exactly the same way as single-chain models; simply add one or more chain break ('/') characters to your alignment file in the appropriate locations.

(You can also build multimeric models from monomeric templates (just use gaps in your alignment for the missing chains in your templates). However, note that since Modeller will have no information about the interfaces between your monomers in this case, your models will probably be poor, so you will have to add additional distance restraints, or find a multimeric template.)

The example below builds a model of a homodimer, and also constrains the two chains to have similar conformations by use of symmetry restraints. Just as for the example in Section 2.2.11, this involves redefining the **AutoModel.special_restraints()** routine. In this case we also redefine the **AutoModel.user_after_single_model()** routine, to print some information about the symmetry restraints after building each model. To build a model of a heterodimer, simply omit this additional restraint.

Example: examples/automodel/model-multichain-sym.py

```
# Comparative modeling by the AutoModel class
#
# Demonstrates how to build multi-chain models, and symmetry restraints
#
from modeller import *
from modeller.automodel import *    # Load the AutoModel class

log.verbose()

# Override the 'special_restraints' and 'user_after_single_model' methods:
class MyModel(AutoModel):
    def special_restraints(self, aln):
        # Constrain the A and B chains to be identical (but only restrain
        # the C-alpha atoms, to reduce the number of interatomic distances
        # that need to be calculated):
        s1 = Selection(self.chains['A']).only_atom_types('CA')
        s2 = Selection(self.chains['B']).only_atom_types('CA')
        self.restraints.symmetry.append(Symmetry(s1, s2, 1.0))
    def user_after_single_model(self):
        # Report on symmetry violations greater than 1A after building
        # each model:
        self.restraints.symmetry.report(1.0)

env = Environ()
# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

# Be sure to use 'MyModel' rather than 'AutoModel' here!
a = MyModel(env,
            alnfile  = 'twochain.ali' ,     # alignment filename
            knowns   = '2abx',              # codes of the templates
            sequence = '1hc9')              # code of the target

a.starting_model= 1                 # index of the first model
```

```
a.ending_model  = 1                 # index of the last model
                                    # (determines how many models to calculate)
a.make()                            # do comparative modeling
```

Example: examples/automodel/twochain.ali

```
C; example for building multi-chain protein models

>P1;2abx
structureX:2abx:   1 :A:74 :B:bungarotoxin:bungarus multicinctus:2.5:-1.00
IVCHTTATIPSSAVTCPPGENLCYRKMWCDAFCSSRGKVVELGCAATCPSKKPYEEVTCCSTDKCNHPPKRQPG/
IVCHTTATIPSSAVTCPPGENLCYRKMWCDAFCSSRGKVVELGCAATCPSKKPYEEVTCCSTDKCNHPPKRQPG*

>P1;1hc9
sequence:1hc9:   1 :A:148:B:undefined:undefined:-1.00:-1.00
IVCHTTATSPISAVTCPPGENLCYRKMWCDVFCSSRGKVVELGCAATCPSKKPYEEVTCCSTDKCNPHPKQRPG/
IVCHTTATSPISAVTCPPGENLCYRKMWCDAFCSSRGKVVELGCAATCPSKKPYEEVTCCSTDKCNPHPKQRPG*
```

2.2.13 Residues and chains in multi-chain models

Just as MODELLER automatically assigns residue numbers starting from 1 for the model sequence, it automatically assigns chain IDs. Chain IDs are assigned alphabetically: A, B, *etc.* Thus, in the previous example (see Section 2.2.12) the model contains residues labeled 1 through 74 in chain A, and 75 through 148 in chain B. You can change this behavior and label the chains and residues yourself by calling **Model.rename_segments()** from within the **AutoModel.special_patches()** method.

You must *always* specify the chain ID when referring to an atom (see Model.atoms) or residue (see Sequence.residues). MODELLER will not 'guess' the chain for you if you leave it out. For example, the CA atom in residue 30 in chain B can be specified with the identifier 'CA:30:B'.

In the example below, the model is relabeled to contain residues 1 through 74 in chain X and 1 through 74 in chain Y. A user-defined restraint is also added between two atoms in the new chain Y. Note that in this example the two chains are *not* constrained to be symmetric; however, the symmetry restraint from the previous example can be added in if desired.

Example: examples/automodel/model-multichain.py

```
# Comparative modeling by the AutoModel class
#
# Demonstrates how to build multi-chain models
#
from modeller import *
from modeller.automodel import *    # Load the AutoModel class

log.verbose()

class MyModel(AutoModel):
    def special_patches(self, aln):
        # Rename both chains and renumber the residues in each
        self.rename_segments(segment_ids=['X', 'Y'],
                             renumber_residues=[1, 1])
        # Another way to label individual chains:
        self.chains[0].name = 'X'
        self.chains[1].name = 'Y'

    def special_restraints(self, aln):
        rsr = self.restraints
        at = self.atoms
#       Restrain the specified CB-CB distance to 8 angstroms (st. dev.=0.1)
#       Use a harmonic potential and X-Y distance group.
#       Note that because special_patches is called before special_restraints,
#       we must use the relabeled chain IDs and residue numbers here.
        rsr.add(forms.Gaussian(group=physical.xy_distance,
                               feature=features.Distance(at['CB:40:Y'],
                                                         at['CB:71:Y']),
                               mean=8.0, stdev=0.1))

env = Environ()
# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

# Be sure to use 'MyModel' rather than 'AutoModel' here!
a = MyModel(env,
            alnfile  = 'twochain.ali' ,     # alignment filename
            knowns   = '2abx',              # codes of the templates
            sequence = '1hc9')              # code of the target

a.starting_model= 2                 # index of the first model
a.ending_model  = 2                 # index of the last model
                                    # (determines how many models to calculate)
a.make()                            # do comparative modeling
```

2.2.14 Accessing output data after modeling is complete

After **AutoModel.make()** finishes building your model(s), the output data is accessible to your script as AutoModel.outputs. This variable is an ordinary Python list, one element for each model (so `a.outputs[0]` refers to the first model, and so on). Each list element is a Python dictionary of key:value pairs, the most important of which are:

- `'failure'`: the Python value `None` if no failure occurred (*i.e.*, the model was built successfully). Otherwise, it is the exception that was raised.
- `'name'`: the name of the output PDB file, if no error occurred.
- `'molpdf'`: the value of the MODELLER objective function, if no error occurred.
- `'pdfterms'`: the contributions to the objective function from all physical restraint types (see Section 6.10.1), if no error occurred.
- `'xxx score'`: the value of the assessment score `'xxx'` (*e.g.*, `'GA341 score'`, `'DOPE score'`).

If you are also building loop models, information for these is made available in LoopModel.loop.outputs.

Example: examples/automodel/model-outputs.py

```
from modeller import *
from modeller.automodel import *
import sys
```

```
log.verbose()
env = Environ()

env.io.atom_files_directory = ['.', '../atom_files']

# Build 3 models, and assess with both DOPE and GA341
a = AutoModel(env, alnfile = 'alignment.ali', knowns = '5fd1',
              sequence = '1fdx', assess_methods=(assess.DOPE, assess.GA341))
a.starting_model= 1
a.ending_model  = 3
a.make()

# Get a list of all successfully built models from a.outputs
ok_models = [x for x in a.outputs if x['failure'] is None]

# Rank the models by DOPE score
key = 'DOPE score'
if sys.version_info[:2] == (2,3):
    # Python 2.3's sort doesn't have a 'key' argument
    ok_models.sort(lambda a,b: cmp(a[key], b[key]))
else:
    ok_models.sort(key=lambda a: a[key])

# Get top model
m = ok_models[0]
print("Top model: %s (DOPE score %.3f)" % (m['name'], m[key]))
```

2.2.15 Fully automated alignment and modeling

If you do not have an initial alignment between your templates and target sequence, MODELLER can derive one for you, fully automatically. All MODELLER requires is a a PIR file containing the target sequence and the template PDB codes (their sequences are not required — just use a single '*' character — as MODELLER will read these from the PDBs). Use the `AutoModel` class as per usual, but call the **AutoModel.auto_align()** method before **AutoModel.make()**; see the example below. (MODELLER has a variety of other alignment methods which you can use instead for this purpose; see Section 6.16 for more details.)

Please be aware that the single most important factor that determines the quality of a model is the quality of the alignment. If the alignment is incorrect, the model will also be incorrect. **For this reason, automated alignment for comparative modeling should not be used unless the sequences are so similar that the calculated alignment is likely to be correct, which usually requires more than 50% sequence identity.** Instead, the alignment should be carefully inspected, optimized by hand, and checked by the **Alignment.check()** command before used in modeling. Moreover, several iterations of alignment and modeling may be necessary in general.

Example: examples/automodel/model-full.py

```
# A sample script for fully automated comparative modeling
from modeller import *
from modeller.automodel import *    # Load the AutoModel class

log.verbose()
env = Environ()
```

```
# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = AutoModel(env,
              # file with template codes and target sequence
              alnfile  = 'alignment.seg',
              # PDB codes of the templates
              knowns   = ('5fd1', '1fdn', '1fxd', '1iqz'),
              # code of the target
              sequence = '1fdx')
a.auto_align()                      # get an automatic alignment
a.make()                            # do comparative modeling
```

Example: examples/automodel/alignment.seg

```
>P1;1fdx
sequence:::::ferredoxin:Peptococcus aerogenes:-1.00:-1.00
AYVINDSCIACGACKPECPVNIIQGSIYAIDADSCIDCGSCASVCPVGAPNPED*
>P1;1fdn
structureX:1fdn:FIRST:@:55:@:ferredoxin:Clostrodium acidiurici: 1.84:-1.0
*
>P1;5fd1
structureX:5fd1:FIRST:@:60:@:ferredoxin:Azotobacter vinelandii: 1.90:0.192
*
>P1;1fxd
structureX:1fxd:FIRST:@:58:@:ferredoxin:Desolfovibrio gigas: 1.70:-1.0
*
>P1;1iqz
structureX:1iqz:FIRST:@:60:@:ferredoxin:Bacillus thermoproteolyticus: 2.30:-1.0
*
```

2.3 Loop optimization

MODELLER has several loop optimization methods, which all rely on scoring functions and optimization protocols adapted for loop modeling [Fiser *et al.*, 2000]. They are used to refine loop regions, either automatically after standard model building, or manually on an existing PDB file.

2.3.1 Automatic loop refinement after model building

To automatically refine loop regions after building standard `AutoModel` models, simply use the `LoopModel` class rather than `AutoModel`; see the example below.

In many cases, you can obtain better quality loops (at the expense of more computer time) by using the newer DOPE-based loop modeling protocol. In this case, just use the `DOPELoopModel` or `DOPEHRLoopModel` classes in place of `LoopModel` in each of the examples below. See Section 4.4 or Section 4.5 for more details.

Example: examples/automodel/model-loop.py

```
# Comparative modeling by the AutoModel class
from modeller import *
from modeller.automodel import *    # Load the AutoModel class
```

```
log.verbose()
env = Environ()

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

a = LoopModel(env,
              alnfile  = 'alignment.ali',     # alignment filename
              knowns   = '5fd1',              # codes of the templates
              sequence = '1fdx')              # code of the target
a.starting_model= 1                 # index of the first model
a.ending_model  = 1                 # index of the last model
                                    # (determines how many models to calculate)
a.md_level = None                   # No refinement of model

a.loop.starting_model = 1           # First loop model
a.loop.ending_model   = 4           # Last loop model
a.loop.md_level       = refine.fast # Loop model refinement level

a.make()                            # do comparative modeling
```

After generating the standard model(s), a number of loop models are generated for each model, from LoopModel.loop.starting_model to LoopModel.loop.ending_model. Each loop model is written out with the `.BL` extension. See section A.5 for more information.

2.3.2 Defining loop regions for refinement

By default, the `LoopModel` class selects all 'loop' regions in your model for refinement, defined as any insertion in the alignment (*i.e.*, a region of the target where template information is not available). You can override this and select any set of atoms of your choosing by redefining the **LoopModel.select_loop_atoms()** routine. (This routine should return a `Selection` object; see Section 2.2.6 or Section 6.9 for further information.)

This example also demonstrates how to automatically assess each generated loop model.

Example: examples/automodel/model-loop-define.py

```
from modeller import *
from modeller.automodel import *

log.verbose()
env = Environ()

env.io.atom_files_directory = ['.', '../atom_files']

# Create a new class based on 'LoopModel' so that we can redefine
# select_loop_atoms
class MyLoop(LoopModel):
    # This routine picks the residues to be refined by loop modeling
    def select_loop_atoms(self):
        # Two residue ranges (both will be refined simultaneously)
        return Selection(self.residue_range('19:A', '28:A'),
                         self.residue_range('45:A', '50:A'))

a = MyLoop(env,
           alnfile  = 'alignment.ali',      # alignment filename
           knowns   = '5fd1',               # codes of the templates
           sequence = '1fdx',               # code of the target
           loop_assess_methods=assess.DOPE) # assess each loop with DOPE
a.starting_model= 1                 # index of the first model
a.ending_model  = 1                 # index of the last model

a.loop.starting_model = 1           # First loop model
a.loop.ending_model   = 2           # Last loop model

a.make()                            # do modeling and loop refinement
```

2.3.3 Refining an existing PDB file

All of the loop modeling classes can also be used to refine a region of an existing PDB file, without comparative modeling, as in the example below. Note that it is necessary in this case to redefine the **LoopModel.select_loop_atoms()** routine, as no alignment is available for automatic loop detection.

Example: examples/automodel/loop.py

```
# Loop refinement of an existing model
from modeller import *
from modeller.automodel import *
#from modeller import soap_loop

log.verbose()
env = Environ()

# directories for input atom files
env.io.atom_files_directory = ['.', '../atom_files']

# Create a new class based on 'LoopModel' so that we can redefine
# select_loop_atoms (necessary)
class MyLoop(LoopModel):
    # This routine picks the residues to be refined by loop modeling
    def select_loop_atoms(self):
        # One loop in chain A from residue 19 to 28 inclusive
        return Selection(self.residue_range('19:A', '28:A'))
        # Two loops simultaneously
        #return Selection(self.residue_range('19:A', '28:A'),
        #                 self.residue_range('38:A', '42:A'))

m = MyLoop(env,
           inimodel='1fdx.B99990001.pdb',   # initial model of the target
           sequence='1fdx',                 # code of the target
           loop_assess_methods=assess.DOPE) # assess loops with DOPE
#          loop_assess_methods=soap_loop.Scorer()) # assess with SOAP-Loop

m.loop.starting_model= 20           # index of the first loop model
m.loop.ending_model  = 23           # index of the last loop model
m.loop.md_level = refine.very_fast  # loop refinement method

m.make()
```

Chapter 3

Frequently asked questions and history

3.1 Frequently asked questions (FAQ) and examples

Please also check the mailing list archives and the release notes.

1. **I want to build a model of a chimeric protein based on two known structures. Alternatively, I want to build a multi-domain protein model using templates corresponding only to the individual domains.**

   This can be accomplished using the standard `AutoModel` class (see Chapter 2). The alignment should be as follows when the chimera is a combination of proteins A and B:

   ```
   proteinA  aaaaaaaaaaaaaaaaaaaaaaaaaaaa----------------------------------
   proteinB  ----------------------------bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
   chimera   aaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
   ```

   If no additional information is available about the relative orientation of the two domains the resulting model will probably have an incorrect relative orientation of the two domains when the overlap between A and B is non-existing or short. To obtain satisfactory relative orientation of modeled domains in such cases, orient the two template structures appropriately before the modeling.

2. **I don't want to use one region of a template for construction of my model.**

   The easiest way to achieve this is to not align that region of the template with the target sequence. If region `'bbbbbbbb'` of the template should not be used as a template for region `'eeeee'` of the target sequence the alignment should be like this:

   ```
   template  aaaaaaaaaaaaaaaaaaaaaaaa-----bbbbbbbbcccccccccccccccccccccccccccccc
   target    ddddddddddddddddddddddddeeeee--------ffffffffffffffffffffffffffffff
   ```

   The effect of this alignment is that no homology-derived restraints will be produced for region `'eeeee'`.

3. **I want to explicitly force certain Pro residues to the *cis* ω conformation.**

   Modeller should usually be allowed to handle this automatically *via* the omega dihedral angle restraints, which are calculated by default.

   ```
   from modeller import *
   from modeller.automodel import *
   from modeller.scripts import cispeptide

   # Redefine the special_restraints routine to force Pro to cis conformation:
   # (this routine is empty by default):
   ```

```
class MyModel(AutoModel):
    def special_restraints(self, aln):
        a = self.atoms
        cispeptide(self.restraints,
                   atom_ids1=(a['O:4'], a['C:4'], a['N:5'], a['CA:5']),
                   atom_ids2=(a['CA:4'], a['C:4'], a['N:5'], a['CA:5']))

# This is as usual:
log.verbose()
env = Environ()

a = MyModel(env, alnfile='align1.ali', knowns='templ1', sequence='targ1')
a.make()
```

4. **How can I select/remove/add a set of restraints?**

   Restraints can be read from a file by **Restraints.append()**, calculated by **Restraints.make()** or **Restraints.make_distance()**, or added "manually" by **Restraints.add()**. **Restraints.pick()** picks those restraints for objective function calculation that restrain the selected atoms only. The 'AutoModel.homcsr()' routine contains examples of selecting atoms when generating restraints by **Restraints.make_distance()**. There are also commands for adding and unselecting single restraints, **Restraints.add()** and **Restraints.unpick()**, respectively. If you do **Restraints.condense()**, the unselected restraints will be deleted. This is useful for getting rid of the unwanted restraints completely.

5. **I want to change the default optimization or refinement protocol.**

   See Section 2.2.2.

6. **I want to build an all hydrogen atom model with water molecules and other non-protein atoms (atoms in the HETATM records in the PDB file).**

   See Sections 2.2.1 and 2.2.5 for some examples.

```
from modeller import *
from modeller.automodel import *

log.verbose()
env = Environ()
env.io.hydrogen = env.io.hetatm = env.io.water = True

a = AllHModel(env, alnfile='align1.ali', knowns='templ1', sequence='targ1')
a.make()
```

7. **How do I build a model with water molecules or residues that do not have an entry in the topology and/or parameter files?**

   See Section 2.2.1 for an example.

8. **How do I define my own residue types, such as D-amino acids, special ligands, and unnatural amino-acids?**

   This is a painful area in all molecular modeling programs. However, CHARMM and X-PLOR provide a reasonably straightforward solution *via* the residue topology and parameter libraries. MODELLER uses CHARMM topology and parameter library format and also extends the options by allowing for a generic "BLK" residue type (Section 5.2.1). This BLK residue type circumvents the need for editing any library files, but it is not always possible to use it. Due to its conformational rigidity, it is also not as accurate as a normal residue type. In order to define a new residue type in the MODELLER libraries, you have to follow the series of steps described below. As an example, we will define the ALA residue without any hydrogen atoms. You can add an entry to the MODELLER topology or parameter file; you can also use your own topology or parameter files. For more information, please see the CHARMM manual.

(a) Define the new residue entry in the residue topology file (RTF), say `'top_heav.lib'`.

```
RESI ALA      0.00000
ATOM N    NH1    -0.29792
ATOM CA   CT1     0.09563
ATOM CB   CT3    -0.17115
ATOM C    C       0.69672
ATOM O    O      -0.32328
BOND CB CA    N CA    O C    C CA    C +N
IMPR C CA +N O     CA N C CB
IC -C    N     CA    C         1.3551  126.4900  180.0000  114.4400    1.5390
IC N     CA    C     +N        1.4592  114.4400  180.0000  116.8400    1.3558
IC +N    CA    *C    O         1.3558  116.8400  180.0000  122.5200    1.2297
IC CA    C     +N    +CA       1.5390  116.8400  180.0000  126.7700    1.4613
IC N     C     *CA   CB        1.4592  114.4400  123.2300  111.0900    1.5461
IC N     CA    C     O         1.4300  107.0000    0.0000  122.5200    1.2297
PATC FIRS NTER LAST CTER
```

You can obtain an initial approximation to this entry by defining the new residue type using the residue type editor in Quanta and then writing it to a file.

The RESI record specifies the Charmm residue name, which can be up to four characters long and is usually the same as the PDB residue name (exceptions are the potentially charged residues where the different charge states correspond to different Charmm residue types). The number gives the total residue charge.

The ATOM records specify the IUPAC (*i.e.*, PDB) atom names and the Charmm atom types for all the atoms in the residue. The number at the end of each ATOM record gives the partial atomic charge.

The BOND records specify all the covalent bonds between the atoms in the residue (*e.g.*, there are bonds CB–CA, N–CA, O–C, etc.). In addition, symbol `'+'` is used to indicate the bonds to the subsequent residue in the chain (*e.g.*, C – +N). The covalent angles and dihedral angles are calculated automatically from the list of chemical bonds.

The IMPR records specify the improper dihedral angles, generally used to restrain the planarity of various groups (*e.g.*, peptide bonds and sidechain rings). See also below.

The IC (internal coordinate) records are used for constructing the initial Cartesian coordinates of a residue. An entry

$$IC \quad a \quad b \quad c \quad d \quad d_{ab} \quad \alpha_{abc} \quad \Theta_{abcd} \quad \alpha_{bcd} \quad d_{cd}$$

specifies distances d, angles α, and either dihedral angles or improper dihedral angles Θ between atoms a, b, c and d, given by their IUPAC names. The improper dihedral angle is specified when the third atom, c, is preceded by a star, `'*'`. As before, the `'-'` and `'+'` prefixes for the atom names select the corresponding atom from the preceding and subsequent residues, respectively. The distances are in angstroms, angles in degrees. The distinction between the dihedral angles and improper dihedral angles is unfortunate since they are the same mathematically, except that by convention when using the equations, the order of the atoms for a dihedral angle is *abcd* and for an improper dihedral angle it is *acbd*.

The PATC record specifies the default patching residue type when the current residue type is the first or the last residue in a chain.

(b) You have to make sure that all the Charmm atom types of the new residue type occur in the MASS records at the beginning of the topology library: Add your entry at the end of the MASS list if necessary. If you added any new Charmm atom types, you also have to add them to the radii library, `'modlib/radii.lib'` and to the solvation library, `'modlib/solv.lib'`. The radii library lists the atomic radii for the different topology models, for the long range non-bonded soft-sphere term. The full name of the file that is used during calculation is given by the environment variable `$RADII_LIB`. Note that Charmm atom type names can be no longer than four characters! Any name longer than that in these files will be silently truncated.

(c) Optionally, you can add the residue entry to the library of MODELLER topology models, `'modlib/models.lib'`. The runtime version of this library is specified by the environment variable `$MODELS_LIB`. This library specifies which subsets of atoms in the residue are used for each of the possible topologies. Currently, there are 10 topologies selected by Topology.submodel (3 is default):

1	ALLH	all atoms
2	POL	polar hydrogens only
3	HEAV	non-hydrogen atoms only
4	MCCB	non-hydrogen mainchain (N, C, CA, O) and CB atoms
5	MNCH	non-hydrogen mainchain atoms only
6	MCWO	non-hydrogen mainchain atoms without carbonyl O
7	CA	CA atoms only
8	MNSS	non-hydrogen mainchain atoms and disulfide bonds
9	CA3H	reduced model with a small number of sidechain interaction centers
10	CACB	CA and CB atoms only

The Ala entry is:

```
#
         ALLH POLH HEAV MCCB MNCH MCWO CA   MNSS CA3H CACB
*
RESI ALA
ATOM     NH1  NH1  NH1  NH1  NH1  NH1  #### NH1  #### ####
ATOM     H    HN   #### #### #### #### #### #### #### ####
ATOM     CT1  CT1  CT1  CT1  CT1  CT1  CT1  CT1  CAH  CT1
ATOM     HB   #### #### #### #### #### #### #### CH3E ####
ATOM     CT3  CT3  CT3  CT3  #### #### #### #### #### CT2
ATOM     HA   #### #### #### #### #### #### #### #### ####
ATOM     HA   #### #### #### #### #### #### #### #### ####
ATOM     HA   #### #### #### #### #### #### #### #### ####
ATOM     C    C    C    C    C    C    #### C    #### ####
ATOM     O    O    O    O    O    #### #### O    #### ####
```

The residue entries in this library are separated by stars. The `'####'` string indicates a missing atom. The atom names for the present atoms are arbitrary. The order of the atoms must be the same as in the CHARMM residue topology library. If a residue type does not have an entry in this library, all atoms are used for all topologies.

(d) You have to add the new residue type to the residue type library, `'modlib/restyp.lib'`. The execution version of this file is specified by the environment variable `$RESTYP_LIB`. See the comments in the file for further information.

Every residue in the CHARMM topology file has to have an entry in the `$RESTYP_LIB` library, but not every residue entry in the `$RESTYP_LIB` library needs an entry in the residue topology file. If you need to edit the `$RESTYP_LIB` file, it is recommended that you change a copy of it, and provide that file to the **Environ()** constructor.

(e) In general, when you add a new residue type, you also add new chemical bonds, angles, dihedral angles, improper dihedral angles, and non-bonded interactions, new in the sense that a unique combination of CHARMM atoms types is involved whose interaction parameters are not yet specified in the parameter library (see also Section 5.2.1). In such a case, you will get a number of warning and/or error messages when you generate the stereochemical restraints by the **Restraints.make()** command. These messages can sometimes be ignored because MODELLER will guess the values for the missing parameters from the current Cartesian coordinates of the model. When this is not accurate enough or if the necessary coordinates are undefined you have to specify the parameters explicitly in the parameter library. Search for BOND, ANGL, DIHE, and IMPR sections in the parameters library file and use the existing entries to guess your new entries. Note that you can use dummy atom types `'X'` to create general dihedral (*i.e.*, `X A A X`) and improper dihedral angle (*i.e.*, `A X X A`) entries, where `A` stands for any of the real CHARMM atom types. For the dihedral angle cosine terms, the CHARMM convention for the phase is different for 180° from MODELLER's (Eq. A.84). If you use non-bonded Lennard-Jones terms, you also have to add a

NONB entry for each new atom type. If you use the default soft-sphere non-bonded restraints, you have already taken care of it by adding the new atom types to the `$RADII_LIB` and `$RADII_LIB` libraries.

9. **How do I define my own patching residue types?**

   This is even messier than defining a new residue type. As an example, we will define the patching residue for establishing a disulfide bond between two CYS residues.

```
PRES DISU         -0.36 ! Patch for disulfides. Patch must be 1-CYS and 2-CYS.
ATOM 1:CB  CT2    -0.10 !
ATOM 1:SG  SM     -0.08 !           2:SG--2:CB--
ATOM 2:SG  SM     -0.08 !          /
ATOM 2:CB  CT2    -0.10 ! -1:CB--1:SG
DELETE ATOM 1:HG
DELETE ATOM 2:HG
BOND 1:SG 2:SG
IC 1:CA  1:CB  1:SG  2:SG      0.0000    0.0000  180.0000    0.0000    0.0000
IC 1:CB  1:SG  2:SG  2:CB      0.0000    0.0000   90.0000    0.0000    0.0000
IC 1:SG  2:SG  2:CB  2:CA      0.0000    0.0000  180.0000    0.0000    0.0000
```

   The PRES record specifies the CHARMM patching residue name (up to four characters).

   The ATOM records have the same meaning as for the RESI residue types described above. The extension is that the IUPAC atom names (listed first) must be prefixed by the index of the residue that is patched, if the patch affects multiple residues. In this example, there are two CYS residues that are patched, thus the prefixes 1 and 2. When using the **Model.patch()** command, the order of the patched residues specified by residues must correspond to these indices (this is only important when the patch is not symmetric, unlike the `'DISU'` patch in this example).

   DELETE records specify the atoms to be deleted, the two hydrogens bonded to the two sulfurs in this case.

   The BOND and IC (internal coordinate) records are the same as those for the RESI residues, except that the atom names are prefixed with the patched residue indices.

10. **Is it possible to restrain secondary structure in the target sequence?**

    Yes — see Section 2.2.11 for an example.

11. **I want to patch the N-terminal or (C-terminal) residue (*e.g.*, to model acetylation properly), but the Model.patch() command does not work.**

    This is probably because the N-terminus is patched by default with the NTER patching residue (corresponding to –NH3$^+$) and a patched residue must not be patched again. The solution is to turn the default patching off by `env.patch_default = False` before the **Model.generate_topology()** command is called.

12. **Is it possible to use templates with the coordinates for C$_\alpha$ atoms only?**

    Yes. You do not have to do anything special.

13. **How do I analyze the output `log` file?**

    First, check for the error messages by searching for string `'_E>''`. These messages can only rarely be ignored. Next, check for the warning messages by searching for string `'_W>''`. These messages can almost always be ignored. If everything is OK so far, the most important part of the `log` file is the output of the **Selection.energy()** command for each model. This is where the violations of restraints are listed. When there are too many too violated restraints, more optimization or a different alignment is needed. What is too many and too much? It depends on the restraint type and is best learned by doing **Selection.energy()** on an X-ray structure or a good model to get a feel for it. You may also want to look at the output of command **Alignment.check()**, which should be self-explanatory. I usually ignore the other parts of the `log` file.

14. **How do I prevent "knots" in the final models?**

    The best way to prevent knots is to start with a starting structure that is as close to the desired final model as possible. Other than that, the only solution at this point is to calculate independently many models and hope that in some runs there won't be knots. Knots usually occur when one or more neighboring long insertions (*i.e.*, longer than 15 residues) are modeled from scratch. The reason is that an insertion is built from a randomized distorted structure that is located approximately between the two anchoring regions. Under such conditions, it is easy for the optimizer to "fall" into a knot and then not be able to recover from it. Sometimes knots result from an incorrect alignment, especially when more than one template is used. When the alignment is correct, knots are a result of optimization not being good enough. However, making optimization more thorough by increasing the CPU time would not be worth it on the average as knots occur relatively infrequently. The excluded volume restraints are already included in standard comparative modeling with the `AutoModel` class (see Chapter 2).

15. **What is considered to be the minimum length of a sequence motif necessary to derive meaningful constraints from the alignment to use in modeling.. one, two, three, or more?**

    Usually more than that (dozens if you want just to detect reliable similarity, and even more if you want a real model). It is good to have at least 35-40% sequence identity to build a model. Sometimes even 30% is OK.

16. **Does Modeller have a graphical interface (GUI) ?**

    No; Modeller is run from the command line, and uses a Python script to direct it. Graphical interfaces to Modeller are commercially available from BIOVIA. Also, check the links page in the Modeller wiki for GUIs contributed by Modeller users.

17. **What do the 'Alignment sequence not found in PDB file' or 'Number of residues in the alignment and pdb files are different' errors mean?**

    When you give MODELLER an alignment, it also needs to read the structure of the known proteins (templates) from PDB files. In order to correctly match coordinates to the residues specified in the alignment, the sequences in the PDB file and the alignment file must be the same (although obviously you can add gap or chain break characters to your alignment). If they are not, you see this error. (Note that MODELLER takes the PDB sequence from the ATOM and HETATM PDB records, not the SEQRES records.) You should also check the header of your alignment file, to make sure that you are reading the correct chain and residue numbers from your PDB.

    To see the sequence that MODELLER reads from the PDB file `'1BY8.pdb'`, use this short script to produce a `'1BY8.seq'` sequence file:

    ```
    from modeller import *
    env = Environ()
    # If you also want to see HETATM residues, uncomment this line:
    #env.io.hetatm = True
    code = '1BY8'
    mdl = Model(env, file=code)
    aln = Alignment(env)
    aln.append_model(mdl, align_codes=code)
    aln.write(file=code+'.seq')
    ```

18. **Can I make a web interface or GUI for Modeller?**

    Certainly, although you should bear in mind that the Modeller license is non-transferable, and permits free usage only for academic purposes.

    For web interfaces, users must obtain their own Modeller license key directly from us; your web interface should provide a text box into which users should put their license key, and then use that input to set the `KEY_MODELLER10v7` environment variable, as is done by our own MODWEB and MODLOOP interfaces. (Note that you will first need to edit the file `modlib/modeller/config.py` in your Modeller installation to remove the line that sets the license, since this takes precedence over the environment variable setting.)

    For GUIs or other interfaces (*e.g.* frameworks), users should obtain and license Modeller directly from us, rather than it being bundled with your software.

    In all cases, please update the links page in the Modeller wiki, to advertise your software to Modeller users.

19. **I get warnings such as 'Could not find platform independent libraries', 'import site failed' or 'No module named socket'**

    These refer to missing Python modules on your system. In the first two cases, these are just warnings that can be safely ignored - most Modeller scripts do not need Python modules anyway, and will run successfully. However, some Modeller scripts, such as the parallel task support, *do* need modules (such as socket) and will not function without them. Please refer to the release notes for two possible solutions in this case.

3.2 Modeller updates

3.2.1 Changes since release 10.6

- mmCIF output now has better support for multiple chains, ligands, DNA/RNA and water, adds chemical component information and user-provided angle and dihedral restraints, and includes both Modeller ("author-provided") residue and chain IDs as well as the mmCIF identifiers (`'seq_id'`, `'asym_id'`).
- Models can now be read from BinaryCIF format files. Templates can also be provided in BinaryCIF format.
- The default file format for **Model.read()** has changed from `PDB_OR_MMCIF` to `PDB_ANY`. A file with a `'.bcif'` extension will now be read as a BinaryCIF file, one with a `'.cif'` extension will be read as mmCIF, while all other files will be read as PDB.
- The internal residue name length has been increased from 4 to 10 characters, to better handle the new 5-character PDB residue names.
- Bundled version of HDF5 updated to 1.14.6.
- Bugfix: atom files are now searched for with the 'pdb' prefix in each directory specified by IOData.atom_files_directory before moving on to the next directory; previously this behavior was reversed.
- Bugfix: avoid numerical instability in **Model.build()** when using internal coordinates that involve colocated atoms.
- Bugfix: correctly relabel sidechain atoms in chiral amino acids (VAL/THR/LEU/ILE) to get the correct chirality if necessary.
- Bugfix: author-provided chain IDs and residue numbers are now used when reading sequence from mmCIF or BinaryCIF, for consistency with reading coordinates.

3.2.2 Changes since release 10.5

- Add Python 3.13 support.
- **Alignment.append_sequence()** now adds a gap with every chain break by default. This makes it easier to construct multiple-sequence alignments. The old behavior can be requested using the new `zero_width_break` argument.
- Bugfix: **Model.get_deletions()** no longer erroneously reports a gap aligned with another gap as a deletion.
- Bugfix: don't put non-UTF8 computer names in the log file on Windows.

Chapter 4

Comparative modeling class reference

4.1 AutoModel reference

All of the functions and data members of the AutoModel class are listed for reference below. Please note that the AutoModel class is derived from the Model class, so all properties and commands of the Model class are also available. Please see section 6.6 for more information.

4.1.1 AutoModel() — prepare to build one or more comparative models

`AutoModel(env, alnfile, knowns, sequence, deviation=None, library_schedule=None, csrfile=None, inifile=None, assess_methods=None, root_name=None)`

alnfile is required, and usually specifies the name of the PIR file which contains an alignment between knowns (the templates) and sequence (the target sequence).

alnfile can instead be a readable file handle (see **modfile.File()**) from which the alignment will be read, or an existing Alignment object containing knowns and sequence. (Note that this is only supported with a subset of AutoModel functionality; in particular, it does not work with parallel jobs, AutoModel.initial_malign3d, or AutoModel.final_malign3d.)

deviation controls the amount of randomization done by randomize.xyz or randomize.dihedrals; see also AutoModel.rand_method. (This can also be set after the object is created, by assigning to 'AutoModel.deviation'. The default is 4Å.)

library_schedule, if given, sets an initial value for AutoModel.library_schedule

If csrfile is set, restraints are not constructed, but are instead read from the user-supplied file of the same name. See section 2.2.9 for an example.

If inifile is set, an initial model is read from the user-supplied file of the same name. See section 2.2.10 for an example.

If root_name is set, it is used to name any output files (see also **AutoModel.get_model_filename()**). By default, files are named using sequence.

assess_methods allows you to request assessment of the generated models (by default, none is done). You can provide a function (or callable), or list of functions, for this purpose, including any of the SOAP potentials (*e.g.*, **soap_loop.Scorer()**, **soap_protein_od.Scorer()**), or any of the standard functions provided in the assess module:

- assess.GA341, which uses the GA341 method (see **Model.assess_ga341()**).
- assess.DOPE, which uses the DOPE method (see **Selection.assess_dope()**).
- assess.DOPEHR, which uses the DOPE-HR method (see **Selection.assess_dopehr()**).

- `assess.normalized_dope`, which uses the normalized DOPE method (see **Model.assess_normalized_dope()**).

(This can also be set after the object is created, by assigning to `'AutoModel.assess_methods'`.) See Section 2.2.3 for an example. Only the region selected by **AutoModel.select_atoms()** is assessed, although most assessment functions take the interaction with the rest of the system into account. Note that only standard models are assessed in this way; if you are also building loop models, see LoopModel.loop.assess_methods.

By default, models are built using heavy atom-only parameters and topology. If you want to use different parameters, read them in before creating the `AutoModel` object with **Topology.read()** and **Parameters.read()**.

See section 2.1 for a general example of using this class.

4.1.2 AutoModel.library_schedule — select optimization schedule

This allows the degree of VTFM optimization to be adjusted. This is usually one of the schedules from the `autosched` module (`autosched.slow`, `autosched.normal`, `autosched.fast`, `autosched.very_fast`, or `autosched.fastest`) or you can provide your own `Schedule` object (see Section 6.12). The default is `autosched.normal`. See the `modlib/modeller/automodel/autosched.py` file for more information.

See Section 2.2.2 for an example.

4.1.3 AutoModel.md_level — control the model refinement level

This allows the degree of MD refinement of the models to be adjusted. You can define your own function for this purpose, set it to the special Python value `None` (in which case no additional refinement is done) or use one of the predefined functions in the `refine` module — `refine.very_fast`, `refine.fast`, `refine.slow`, `refine.very_slow` or `refine.slow_large` — which perform different amounts of MD annealing. See the `modlib/modeller/automodel/refine.py` file for more information.

See Section 2.2.2 for an example.

4.1.4 AutoModel.outputs — all output data for generated models

This is produced after **AutoModel.make()** is finished. It contains filenames and model scores for every generated model. This information is provided for your own post-processing (*e.g.*, ranking and further refinement of the models) although a summary of it is printed at the end of the model run. See Section 2.2.14 for an example.

4.1.5 AutoModel.rand_method — control initial model randomization

This is used to randomize the initial model before producing each final model. If set to `None` then no randomization is done, and every model will be the same. You can set it to one of the functions in the `randomize` module — `randomize.xyz` (the default) to randomize all coordinates of standard residues, or `randomize.dihedrals` to randomize dihedral angles. (The amount of randomization carried out by these two functions can be tuned by changing AutoModel.deviation.) Finally, you could provide your own Python function; see the `randomize` module in `modlib/modeller/automodel/randomize.py` for examples to get you started.

4.1.6 AutoModel.generate_method — control initial model generation

This is used to build the initial model. It is usually set to `generate.transfer_xyz`, which builds the model based on the templates, but you can also set it to `generate.generate_xyz` to build it purely from the internal coordinates, or to `generate.read_xyz` to read it from a file (see section 2.2.10 for the easiest way to do this).

4.1.7 AutoModel.max_var_iterations — select length of optimizations

This is used to set max_iterations for every call to **ConjugateGradients()**. Smaller numbers may lead to more rapid (but less accurate) model building.

4.1.8 AutoModel.repeat_optimization — number of times to repeat optimization

The entire optimization cycle is repeated this many times; increase the value from 1 (the default) to request more thorough optimization.

4.1.9 AutoModel.max_molpdf — objective function cutoff

VTFM optimization of each model is automatically aborted (continuing with the next model, if any) if the objective function becomes larger than this value.

4.1.10 AutoModel.initial_malign3d — initial template alignment

If set to `True`, then an initial structural alignment of all templates is done.

4.1.11 AutoModel.starting_model — first model to build

This determines the number of the first model to build; models are built from AutoModel.starting_model through to AutoModel.ending_model.

4.1.12 AutoModel.ending_model — last model to build

This determines the number of the last model to build; see AutoModel.starting_model.

4.1.13 AutoModel.final_malign3d — final template-model alignment

If set to `True`, then all of the generated models (and loop models, if using `LoopModel`) are fit to the templates, and written out with the `_fit.pdb` extension.

4.1.14 AutoModel.write_intermediates — write intermediate files during optimization

If set to `True`, then PDB or mmCIF files are written out during the optimization (containing intermediate, partially optimized coordinates) in addition to the final model file(s).

4.1.15 AutoModel.trace_output — control optimization output

For every `.B` model file produced during modeling, a corresponding `.D` optimization trace file is produced, using the **actions.Trace()** periodic action. (When loop modeling with `LoopModel`, a `.DL` file is produced for each `.BL` loop model.) These files contain information about the progress of optimization, such as the iteration step, atomic shifts in space, energies and gradients. By default, this is written out every 10 steps, but you can change the frequency by assigning to this variable, or turn it off completely by setting it to zero.

See also **AutoModel.get_optimize_actions()** and **AutoModel.get_refine_actions()**.

4.1.16 AutoModel.max_ca_ca_distance — Distance cutoff for CA-CA homology-derived restraints

This is the cutoff distance for C_α-$C_\alpha$homology-derived restraints (it is passed to **Restraints.make_distance()** as the maximal_distance parameter when these restraints are generated). No C_α-$C_\alpha$distances greater than this value will be used in building homology-derived restraints; thus, reducing this value from the default will typically reduce the number of restraints and increase the speed of optimization.

The default value of this parameter is 14.0 angstroms.

See also AutoModel.max_n_o_distance, AutoModel.max_n_o_distance, AutoModel.max_sc_mc_distance, and AutoModel.max_sc_sc_distance for similar cutoffs for the other kinds of homology-derived distance restraints.

See also **AutoModel.very_fast()**, which decreases all of these distances from their default values.

4.1.17 AutoModel.max_n_o_distance — Distance cutoff for N-O homology-derived restraints

No homology-derived restraints will be generated using template N-O distances greater than this value. The default value of this parameter is 11.0 angstroms.

See AutoModel.max_ca_ca_distance for more details.

4.1.18 AutoModel.max_sc_mc_distance — Distance cutoff for sidechain-mainchain homology-derived restraints

No homology-derived restraints will be generated using template sidechain-mainchain distances greater than this value. The default value of this parameter is 5.5 angstroms.

See AutoModel.max_ca_ca_distance for more details.

4.1.19 AutoModel.max_sc_sc_distance — Distance cutoff for sidechain-sidechain homology-derived restraints

No homology-derived restraints will be generated using template sidechain-mainchain distances greater than this value. The default value of this parameter is 5.0 angstroms.

See AutoModel.max_ca_ca_distance for more details.

4.1.20 AutoModel.blank_single_chain — Control chain ID for single-chain models

If set `False` (the default), the chain in any generated single-chain model will be labeled 'A'. If set `True`, it will be given an empty (blank) chain ID. This parameter has no effect when building multi-chain models.

4.1.21 AutoModel.set_output_model_format() — set format for output models

```
set_output_model_format(fmt)
```

This allows the format for output models to be set. Valid values are `PDB` or `MMCIF`. Normally output models are written in PDB format; however, mmCIF files support larger structures and have better support for metadata (such as information on the templates and alignment used in the modeling).

The templates for modeling can be mmCIF, BinaryCIF or PDB; they do not affect the output format.

Note that the initial model and any loop models are written in the same format (and if an initial model is provided, it will be read in that format).

4.1.22 AutoModel.get_optimize_actions() — get actions to carry out during the initial optimization

```
get_optimize_actions()
```

This returns a list of optimizer actions which are carried out during the initial optimization (and for LoopModel, also during loop modeling). (By default, only the trace file is written (see AutoModel.trace_output.) You can override this method to perform other actions (see Section 6.11) during the optimization — *e.g.* writing a CHARMM trajectory file using **actions.CHARMMTrajectory()**.

4.1.23 AutoModel.get_refine_actions() — get actions to carry out during the refinement

```
get_refine_actions()
```

This returns a set of optimizer actions which are carried out during the molecular dynamics refinement part of the optimization. By default, it does the same thing as **AutoModel.get_optimize_actions()**.

4.1.24 AutoModel.select_atoms() — select region for optimization and assessment

```
select_atoms()
```

By default, this selects all atoms in the system. Only the selected atoms are optimized in model building, so you can redefine this routine to select the region of interest, if so desired. See section 2.2.6 for an example.

4.1.25 AutoModel.auto_align() — generate an automatic initial alignment

```
auto_align(matrix_file='family.mat', overhang=0, write_fit=False)
```

This generates an initial alignment between the templates and the target sequence. See section 2.2.15 for an example.

4.1.26 AutoModel.very_fast() — request rapid optimization

```
very_fast()
```

This sets parameters to request very fast optimization of the model(s). It reduces the cutoff distances for homology-derived restraints (*e.g.* AutoModel.max_ca_ca_distance), turns off model randomization (AutoModel.rand_method) and refinement (AutoModel.md_level), selects a very fast optimization schedule (AutoModel.library_schedule) and reduces the number of optimization steps (max_var_iterations). See section 2.2.3 for an example.

4.1.27 AutoModel.make() — build all models

```
make(exit_stage=0)
```

You should call this command after creating an AutoModel object and setting any desired parameters, to then go ahead and build all models.

If exit_stage is 2, then this routine exits after generating the initial model; no optimized models are built. If it is 1, then it also creates the restraints file before exiting. If it is 0 (the default) then the full comparative modeling procedure is followed.

4.1.28 AutoModel.cluster() — cluster all built models

`cluster(cluster_cut=1.5)`

This can be called after model building is complete. It clusters all of the output models, and outputs an averaged cluster structure, both optimized (in the file `cluster.opt`) and unoptimized (in `cluster.ini`). cluster_cut gives the cluster cutoff distance, as used in **Model.transfer_xyz()**.

4.1.29 AutoModel.special_restraints() — add additional restraints

`special_restraints(aln)`

This method is called automatically by `AutoModel` after the default restraints are generated. By default, it does nothing. However, you can redefine it to add additional user-defined restraints to those calculated by `AutoModel`. Symmetry restraints, excluded atom pairs, and rigid body definitions can also be set up in this routine. The routine is passed aln, which is the alignment between the templates and the target sequence. See section 2.2.11 for an example.

4.1.30 AutoModel.nonstd_restraints() — add restraints on ligands

`nonstd_restraints(aln)`

This method is called automatically by `AutoModel` after it generates the standard protein restraints. It adds restraints to keep non-standard residues (anything treated as a HETATM or BLK residue, such as ligands or metal ions) in a reasonable conformation. You can override this method if you need to change these restraints.

By default, four sets of restraints are built:

- For each residue that has no defined topology (generally BLK residues, used to transfer ligands directly from templates, as described in Section 2.2.1), intra-residue distances are all constrained to their template values. This causes each residue to behave as a rigid body.
- Inter-residue distances are constrained to template values if these are 7Å or less. This has the effect of preserving multiple-HETATM structures such as DNA chains. If the distances are 2.3Å or less they are assumed to be bonds and so are restrained more strongly and also excluded from the nonbonded list.
- Residue-protein atom distances are strongly constrained to template values (and excluded from the nonbonded list) if these are 2.3Å or less. This preserves chemical bonds between ligands and the protein.
- Residue-protein C_α distances are constrained to template values if these are 10Å or less and are not already bonded by the previous restraints. This preserves the ligand position.

4.1.31 AutoModel.special_patches() — add additional patches to the topology

`special_patches(aln)`

This routine, which is usually empty, can be redefined by the user to make additional changes to the model topology, such as:

- adding patches by calling **Model.patch()**, such as user-defined disulfides (see Section 2.2.7 for an example)
- renumbering the residues or renaming the chains (see Section 2.2.13 for an example)
- changing CHARMM atom types, by assigning to Atom.type or by calling **AutoModel.guess_atom_types()**

4.1.32 AutoModel.user_after_single_model() — analyze or refine each model

```
user_after_single_model()
```

This routine is called after building each model, before the output PDB or mmCIF file is written. It usually does nothing, but can be redefined by the user to perform analysis or additional refinement on each model. See Section 2.2.12 for an example. For loop models, see **LoopModel.user_after_single_loop_model()**.

4.1.33 AutoModel.get_model_filename() — get the model PDB/mmCIF name

```
get_model_filename(root_name, id1, id2, file_ext)
```

This routine returns the PDB or mmCIF file name of each generated model, usually of the form `'foo.B000X000Y.pdb'` (where root_name (or sequence)=foo, id1=X, id2=Y, and file_ext=.pdb). You can redefine this routine if you don't like the standard naming scheme. For typical **AutoModel** usage, id1 is a constant and id2 is the model number, running from AutoModel.starting_model to AutoModel.ending_model.

See also **LoopModel.get_loop_model_filename()** and **AutoModel.set_output_model_format()**.

4.1.34 AutoModel.use_parallel_job() — parallelize model building

```
use_parallel_job(job)
```

This uses a parallel job object (see Section 6.35) to speed up model building. When building multiple models, the optimization process is spread over all nodes in the parallel job (for example, if building 10 models, and you are running on 2 nodes, each node will build 5 models). This feature is still experimental.

4.1.35 AutoModel.guess_atom_types() — automatically assign CHARMM atom types

```
guess_atom_types()
```

When using BLK residues to represent ligands or other non-standard residues as rigid bodies (see Section 5.2.1) all atoms in these residues are assigned the CHARMM `'undf'` type, which behaves similarly to carbon or nitrogen. This can lead to inaccurate soft-sphere or Lennard-Jonesinteractions for hydrogens or metal ions. This function will attempt to assign suitable CHARMM atom types to all residues without defined topology (usually BLK residues) in the model. To use it, call it from **AutoModel.special_patches()**. Any atom type assignments are shown in the log file, which should be carefully inspected for mistakes. It calls **AutoModel.guess_atom_type()** for each atom, so its guesses can be improved by overriding that function.

4.1.36 AutoModel.guess_atom_type() — automatically assign CHARMM atom type

```
guess_atom_type(atom)
```

This assigns the CHARMM atom type (see Section 6.24) for the given Atom object (see Section 6.23). By default, this is done by guessing the type based on the atom name, but the method can be overridden to improve the guesses. It is usually called from **AutoModel.guess_atom_types()**.

4.2 `AllHModel` reference

The `AllHModel` class is derived from `AutoModel`, so all properties and commands of both the `AutoModel` and `Model` classes are available in addition to those listed below.

4.2.1 AllHModel() — prepare to build all-hydrogen models

```
AllHModel(env, alnfile, knowns, sequence, deviation=None, library_schedule=None, csrfile=None,
inifile=None, assess_methods=None)
```

This creates a new object for building all-hydrogen models. All of the arguments are the same as those for **AutoModel()**.

See section 2.2.5 for an example.

4.3 `LoopModel` reference

The `LoopModel` class is derived from `AutoModel`, so all properties and commands of both the `AutoModel` and `Model` classes are available in addition to those listed below.

4.3.1 LoopModel() — prepare to build models with loop refinement

```
LoopModel(env, sequence, alnfile=None, knowns=[], inimodel=None, deviation=None,
library_schedule=None, csrfile=None, inifile=None, assess_methods=None,
loop_assess_methods=None, root_name=None)
```

This creates a new object for loop modeling. It can either build standard comparative models (in identical fashion to the `AutoModel` class) and then refine each of them, in which case you should set the alnfile and knowns arguments appropriately (see the **AutoModel()** documentation) or it can refine a given region of a PDB or mmCIF file, in which case you should set inimodel to the name of the PDB or mmCIF file instead. In both cases, sequence identifies the code of the target sequence.

All other arguments are the same as those for **AutoModel()**, with the exception of those below:

loop_assess_methods is the analog of AutoModel.assess_methods for loop modeling, and allows you to request assessment of the generated loop models. (This can also be set after the object is created, by assigning to `'LoopModel.loop.assess_methods'`.) Only the region selected by **LoopModel.select_loop_atoms()** is assessed, although most assessment functions take the interaction with the rest of the system into account.

See section 2.3 for examples.

4.3.2 LoopModel.loop.md_level — control the loop model refinement level

This is the analog of AutoModel.md_level for loop modeling, and allows the loop model refinement to be customized.

4.3.3 LoopModel.loop.max_var_iterations — select length of optimizations

This is the analog of AutoModel.max_var_iterations for loop modeling.

4.3.4 LoopModel.loop.library_schedule — select optimization schedule

This is the analog of AutoModel.library_schedule for loop modeling.

4.3.5 LoopModel.loop.starting_model — first loop model to build

This is the analog of AutoModel.starting_model and determines the number of the first loop model to build for each regular model.

4.3.6 LoopModel.loop.ending_model — last loop model to build

This is the analog of AutoModel.ending_model and determines the number of the last loop model to build for each regular model.

4.3.7 LoopModel.loop.write_selection_only — write PDB/mmCIFs containing only the loops

If set to `True`, then the generated PDB or mmCIF files contain only the loops themselves, not the rest of the protein. The default value is `False`, which generates complete protein PDBs.

Note that if set to `True`, AutoModel.final_malign3d cannot also be set to `True`, since the final structural alignment requires the full structure of each loop model.

4.3.8 LoopModel.loop.write_defined_only — only write non-loop atoms present in the input model

When constructing the initial model for loop refinement, MODELLER fills in any missing atoms in both the loop and non-loop regions. These are needed in order to accurately determine the interaction between the loop and the rest of the protein. The coordinates for the missing atoms are constructed automatically using internal coordinates, so clashes between the atoms and the rest of the protein may exist (note, however, that the score of the loop does not include protein-protein internal interactions, so will not be affected).

If set to `False`, the default, the written-out models will contain the atoms present in the original non-loop region plus those added automatically. If set to `True`, the models will contain only the non-loop atoms that were present in the input model.

Note that if LoopModel.loop.write_selection_only is set to `True`, only the loop region is written out, so this option has no effect.

4.3.9 LoopModel.loop.outputs — all output data for generated loop models

This is the analog of AutoModel.outputs for loop modeling; it contains filenames and model scores for every generated loop model.

4.3.10 LoopModel.select_loop_atoms() — select region for loop optimization and assessment

```
select_loop_atoms()
```

By default, this selects all atoms near gaps in the alignment for loop optimization (by calling **Model.loops()**). You should redefine this routine if you do not have an alignment, or you wish to set a different region for loop optimization. See section 2.3 for an example.

4.3.11 LoopModel.get_loop_model_filename() — get the model PDB/mmCIF name

`get_loop_model_filename(root_name, id1, id2, file_ext)`

This routine returns the PDB or mmCIF file name of each generated loop model, usually of the form `'foo.BL000X000Y.pdb'` (where root_name (or sequence)=foo, id1=X, id2=Y, and file_ext=.pdb). You can redefine this routine if you don't like the standard naming scheme. For typical **LoopModel** usage, id1 is the loop model number, running from LoopModel.loop.starting_model to LoopModel.loop.ending_model, and id2 is the comparative model number, running from AutoModel.starting_model to AutoModel.ending_model.

See also **AutoModel.get_model_filename()**.

4.3.12 LoopModel.user_after_single_loop_model() — analyze or refine each loop model

`user_after_single_loop_model()`

This is the analog of **AutoModel.user_after_single_model()**, and is called after building each loop model, before the output PDB/mmCIF file is written. It can be redefined by the user to perform analysis or additional refinement on each loop model.

4.3.13 LoopModel.read_potential() — read in the loop modeling potential

`read_potential()`

This reads in the `<GroupRestraints>` object which defines the statistical potential for loop modeling. Redefine this routine if you want to use a different potential.

4.3.14 LoopModel.build_ini_loop() — create the initial conformation of the loop

`build_ini_loop(atmsel)`

This creates the initial conformation of the loop. By default all atoms are placed on a line between the loop termini, but you may want to use a different conformation, in which case you should redefine this routine. For example, if you want to leave the initial PDB/mmCIF file untouched, use a one-line 'pass' routine.

4.4 `DOPELoopModel` reference

The `DOPELoopModel` class is derived from `LoopModel`, and is very similar in operation, except that a newer DOPE-based loop modeling protocol is used (see **Selection.assess_dope()**). The main differences are:

- DOPE (see **Selection.assess_dope()**) potential used in combination with GB/SA implicit solvent interaction (see Section 6.14).
- Lennard-Jones potential used rather than soft-sphere.
- Improved handling of ligand-loop interactions.

To use, simply use '`DOPELoopModel`' rather than '`LoopModel`' in your Python scripts (see Section 2.3 for examples). Note that it will be significantly slower than the regular loop modeling procedure, primarily due to the GB/SA interaction.

4.4.1 DOPELoopModel() — prepare to build models with DOPE loop refinement

```
DOPELoopModel(env, sequence, alnfile=None, knowns=None, inimodel=None, deviation=None,
library_schedule=None, csrfile=None, inifile=None, assess_methods=None,
loop_assess_methods=None, root_name=None)
```

See **LoopModel()** for all arguments.

4.5 `DOPEHRLoopModel` reference

This class is identical to `DOPELoopModel`, except that the higher precision DOPE-HR method (see **Selection.assess_dopehr()**) is used.

Chapter 5

Modeller general reference

Sections in this Chapter describe technical aspects of Modeller.

5.1 Miscellaneous rules and features of Modeller

This Section describes several features of the program, including file naming conventions, various file types, and the control of the amount of output.

5.1.1 Modeller system

One of the main aims of Modeller is to allow for flexible exploration of various modeling protocols to facilitate the development of better modeling methods. Modeller can be seen as an interpreted language that is specialized for modeling of protein 3D structure.

See section 1.6.2 for basic information on writing and running scripts.

5.1.2 Controlling breakpoints and the amount of output

Some errors are recoverable. For those errors, a Python exception exception is raised. It is then up to your script to deal sensibly with the failure of the preceding command using a standard `'except'` clause. For example, this flexibility allows derivation of multiple models and searching for many sequences, even if some cases abort due to convergence problems. This exception could be a generic `ModellerError` exception, a more specific subclass (`FileFormatError`), or a standard Python exception[1].

There are five kinds of messages that Modeller writes to the `log` file: long output from the Modeller commands, short notes to do with the execution of the program (files opened, *etc.*), warnings identified by `'_W>'`, errors identified by `'_E>'`, and the messages about the status of dynamic memory allocation. To control how much of this output is displayed, use the `log` object; for more information, see Section 6.31.

5.1.3 File naming

There are several filename generating mechanisms that facilitate file handling. Not all of them apply to all file types.

Environment variables

There can be Unix shell environment variables in any input or output filename. The environment variables have to be in the format `${VARNAME}` or `$(VARNAME)`. Also, two predefined macros are available for string variables:

[1]Modeller can raise the following Python exceptions: `ZeroDivisionError`, `IOError`, `MemoryError`, `EOFError`, `TypeError`, `NotImplementedError`, `UnicodeError`, `IndexError`, and `ValueError`.

- `${LIB}` is expanded into the `$LIB_MODELLER` variable defined in `modlib/libs.lib` (equal to `$MODINSTALL10v7/modlib`);
- `${JOB}` is expanded into the root of the script filename, or `'(stdin)'` if instructions are being read from standard input;

If the `MODELLER_DEPRECATION` environment variable is set to `ERROR`, use of any deprecated class name[2] in the input script will cause a fatal error (normally this will only trigger a warning).

Reading or writing files

Any input file for MODELLER (alignments, PDB files, *etc*) can be compressed. If the name of an input file ends with a `'.Z'`, `'.gz'`, `'.bz2'`, or `'.7z'` extension, or the specified input file cannot be found but a compressed version (with extension) does, then MODELLER automatically uncompresses the file before reading it. (Note that it uses the `gzip`, `bzip2` and `7za` programs to do this, so they must be installed on your system in order for this to work. Also, any `'.7z'` archives must contain only a single member, which is the file to be uncompressed, just as with `'.gz'` or `'.bz2'` files.) The uncompressed copy of the file is created in the system temporary directory (deduced by checking the `'MODELLER_TMPDIR'`, `'TMPDIR'`, `'TMP'` and `'TEMP'` environment variables in that order, falling back to `/tmp` on Unix and `C:\` on Windows), or the current working directory if the temporary directory is read-only.

Any files written out by MODELLER can also be compressed. If the output file name ends in `'.gz'` or `'.bz2'` extension, a temporary uncompressed copy is created in the same way as above, and when the file is closed, the file is compressed with `gzip` or `bzip2` and placed in the final location. (Writing out files in `'.Z'` or `'.7z'` format is not currently supported.)

Many MODELLER functions that take file names can also be given file handles; these can either be **modfile.File()** objects or Python filelike objects such as `sys.stdout`.

Coordinate files and derivative data

When accessing an atom file, if MODELLER cannot find the specified filename or a compressed version of it (see above) it tries adding the extensions `'.atm'`, `'.pdb'`, `'.ent'`, `'.cif'`, `'.bcif'`, and `'.crd'` in this order, then also with the `'pdb'` prefix. If the filename is not an absolute path (*i.e.*, it does not start with `'/'`) then this search is then repeated through all the directories in IOData.atom_files_directory. PDB-style subdirectories (the last two but one characters in the PDB code) are also searched for each directory *e.g.*, 1abc is searched for in the 'ab' subdirectory, pdb4xyz.ent in the 'xy' subdirectory, and the new-style 12-character extended PDB ID PDB_12345678 in the '67' subdirectory. Atom files can be read in PDB, mmCIF, or BinaryCIF format.

Any derivative data that MODELLER may need, including residue solvent accessibilities, hydrogen bonding information, dihedral angles, residue neighbors, *etc.*, are calculated on demand from the atomic coordinates. The most time consuming operation is calculating solvent accessibility, but even this calculation takes less than 1 sec for a 200 residue protein on a Pentium III workstation.

MODELLER stores the filenames of coordinate sets in the alignment arrays. These arrays are used by **Alignment.compare_structures()**, **Restraints.make()**, **Alignment.malign3d()**, **Alignment.align2d()**, and several other commands. If these filenames do not change when the structures are needed for the second time, the coordinate files are not re-read because they should already be in memory. This creates a problem only when the contents of a structure file changes since it was last read during the current job.

Unicode

MODELLER supports Unicode for file naming, so files named using non-English characters can be accessed. If you wish to access such a file, specify the file name in your data file (*e.g.* alignment file) or Python 2 script in UTF-8 encoding. MODELLER will raise a `UnicodeError` if your filenames are not valid UTF-8. (If using Python 3, you

[2]All MODELLER Python classes are named using CamelCase, such as `Model`, `Alignment`, or `AutoModel`. Previous versions of MODELLER used lowercase names, such as `model`, `alignment`, or `automodel`. These lowercase names are still available but are deprecated.

need do nothing special, since it already understands Unicode.) Since UTF-8 is a superset of ASCII, if you are using only English characters you need do nothing special. [3]

Modeller input files are assumed to be UTF-8 encoded. However, most of the data Modeller handles is not Unicode-enabled (for example, PDB files and one letter residue types have to be ASCII, not Unicode), so you should not use non-English characters, except in filenames.

5.1.4 File types

Modeller uses a number of standard filename extensions to indicate the type of data stored in a file (Table 5.1). The extensions are generally not mandatory, only very helpful.

Extension	Description
`.top`	TOP script with instructions for a Modeller job
`.log`	log output produced by a Modeller run
`.ali`	alignment or sequences in the PIR format
`.pap`	alignment or sequences in the PAP format
`.aln`	alignment or sequences in the Quanta format
`.aln`	alignment or sequences in the InsightII format
`.seq`, `.chn`	sequence(s) in the PIR alignment format
`.cod`	list of sequence codes
`.grp`	list of families in PDB
`.atm`, `.pdb`, `.ent`	atom coordinates in the PDB or Grasp format
`.crd`	atom coordinates in the Charmm format
`_fit.pdb`	fitted protein structures in the PDB format
`.ini`	initial Modeller model
`.B*`	Modeller model in the PDB format
`.D*`	the progress of optimization
`.BL*`	Modeller model in the PDB format, in loop modeling
`.DL*`	the progress of optimization, in loop modeling
`.IL*`	initial Modeller model, in loop modeling
`.V*`	violations profile
`.E*`	energy profile
`.rsr`	restraints in MODELLER format
`.lrsr`	restraints in MODELLER format, in loop modeling
`.sch`	schedule file for the variable target function optimization
`.mat`	matrix of pairwise protein distances from an alignment
`.mat`	matrix of pairwise residue type–residue type distance scores
`.sim.mat`	matrix of pairwise residue type–residue type similarity scores
`.lib`	various Modeller libraries
`.psa`	residue solvent accessibilities
`.sol`	atomic solvent accessibilities
`.ngh`	residue neighbors
`.dih`	mainchain and sidechain dihedral angles
`.ssm`	secondary structure assignment
`.var`	sequence variability profile from multiple alignment
`.asgl`	data for plotting by Asgl

Table 5.1: *List of file types.*

[3]On Unix/Linux systems, Modeller assumes that your filesystem also stores filenames in UTF-8. This is usually the case on modern systems; however, you can change the encoding by setting the `G_FILENAME_ENCODING` environment variable.

5.2 Stereochemical parameters and molecular topology

All molecular modeling programs generally need to know what are the atoms in all residue types, what are the atom pairs that are covalently bonded to each other (*i.e.*, molecular topology), and what are the ideal bond lengths, angles, dihedral angles, and improper dihedral angles (*i.e.*, internal coordinates and stereochemical restraints). This information is stored in the residue topology and parameter libraries, which are manipulated by MODELLER scripts using the `Libraries` class (see Section 6.5.

For commands dealing with generating, patching, and mutating molecular topology, see the `Model` class reference, in section 6.6.

5.2.1 Modeling residues with non-existing or incomplete entries in the topology and parameter libraries

Defining new residue types is generally one of the more painful areas in developing and using a molecular modeling program. MODELLER has two quick-and-dirty solutions described in the next two sections that are often sufficient for comparative modeling involving new residue types. On the other hand, if you are willing to spend some time and define a new entry or complete an incomplete entry in the residue topology or parameter libraries, see the FAQ Section 3.1, Question 8.

Residues with defined topology, but with missing parameters

The parameter library is used by the **Restraints.make()** command to construct bond, angle, dihedral angle, improper dihedral angle, and non-bonded Lennard-Jones restraints. If some parameters for these restraints are missing, they are guessed on the fly from the current Cartesian coordinates of the MODEL. Thus, when there are missing parameters, the MODEL coordinates must be defined before calling **Restraints.make()**. The coordinates can be defined by the **Model.build()** command (from the IC entries in the residue topology library), by the **Model.read()** command (from an existing coordinate file for MODEL), or by the **Model.transfer_xyz()** command (from template coordinate files aligned with MODEL). The bonds, angles, and improper dihedral angles are restrained by a harmonic potential with the mean equal to the value in the current structure and a force constant typical for chemical bonds, angles, and improper dihedral angles, respectively. The dihedral angles are restrained by a tri-modal cosine term with the mean equal to the angle in the current structure. A message detailing MODELLER's improvisation is written to the `log` file.

Block (BLK) residues with undefined topology and parameters

The second relatively easy way of dealing with missing entries in the residue topology and/or parameters libraries is to use a "block" residue. These residues are restrained more or less as rigid bodies to the conformation of the equivalent residue(s) in the template(s). No chemical information is used. The template residues can themselves be defined as block residues. The symbol for the block residues is 'BLK' in the four- and three-letter codes and '.' in the single-letter code. The atoms in a BLK residue include all uniquely named atoms from the equivalent residues in all the templates. The atom type of all BLK atoms is the CHARMM type `'undf'` (but note that this can be changed by assigning to Atom.type). The IUPAC atom names (as opposed to the atom types) are the same as in the templates.

BLK atoms are treated differently from the other atoms during preparation of dynamic restraints: No pairs of intra-BLK atoms are put on the dynamic non-bonded list. Only the "inter-BLK" atom pairs and "BLK–other" atom pairs are considered for the dynamic non-bonded restraints. The radius of all block atoms (for soft-sphere restraints) is that of the CHARMM `'undf'` atom type. All intra-BLK and inter-residue BLK restraints other than the non-bonded restraints have to be derived separately and explicitly by the **Restraints.make_distance()** command. See **AutoModel.nonstd_restraints()** for the routine that makes block restraints for comparative modeling with the `AutoModel` class. Lennard-Jones terms use the CHARMM parameters for `'undf'` atoms (`par.lib` assumes these are similar to those for typical heavy atoms). Coulomb terms involving `'undf'` atoms are ignored by MODELLER. Non-bonded spline restraints (see EnergyData.dynamic_modeller and **GroupRestraints()**) derive their atom classes simply from the atom and residue names, so will function with BLK atoms only if the names of the BLK atoms and residues are given in the spline restraints atom class file. GB/SA restraints (see **gbsa.Scorer()**) will treat all BLK atoms as uncharged and with the same radius (that given in `solv.lib` for the `'undf'` atom type).

See also **AutoModel.guess_atom_types()**, to obtain improved interaction parameters for BLK residues.

Please note that if you use 'BLK' residues, you should set IOData.hetatm to `True`, as most 'BLK' residues are PDB HETATM residues (note, however, that 'BLK' residues can be either HETATM or ATOM; for example, any DNA or RNA residues handled as 'BLK' will be ATOM residues).

For an example of how to use block residues, see Section 2.2.1.

5.3 Spatial restraints

The objective function used by MODELLER is a sum over all of the restraints. See Section A.3 for equations defining the restraints and their derivatives with respect to atomic positions. See Section 6.6 for commands for calculating the objective function and Section A.2 for optimization methods. See the original papers for the most detailed definition and description of the restraints [Šali & Blundell, 1993, Šali & Overington, 1994].

5.3.1 Specification of restraints

Static and dynamic restraints

Dynamic restraints are created on the fly, and currently include:

- Soft-sphere overlap restraints (see EnergyData.dynamic_sphere).
- Lennard-Jones restraints (see EnergyData.dynamic_lennard).
- Coulomb restraints (see EnergyData.dynamic_coulomb).
- Non-bond spline restraints (see EnergyData.dynamic_modeller).
- GBSA solvent restraints (see Section 6.14).
- EM density restraints (see EnergyData.density).
- SAXS restraints (see EnergyData.saxsdata).
- User-defined energy terms (see Section 7.1.3).

Dynamic restraints are *not* written into the restraints file by **Restraints.write()** (only static restraints are).

Static restraints can be added with the **Restraints.add()** command, or can be read from a restraints file (see Section B.2). Collections of static restraints useful for various purposes (*e.g.* for restraining all bond lengths or angles, or for using template information) can also be automatically generated with the **Restraints.make()** command.

Each static restraint is formulated as a mathematical form (*e.g.* a Gaussian function) which acts on one or more 'features' of the model (*e.g.* a bond length). Any feature can be used with any mathematical form, with the exception of `forms.MultiBinormal`, which generally only works properly with `features.Dihedral`. Both feature types and mathematical forms are described below.

Feature types

Each feature is a Python class, which takes a defined number of atom ids as input. Each of these atom ids can be:

- An `Atom` object, from the current model (*e.g.*, `m.atoms['CA:1']`; see Model.atoms).
- A `Residue` object, from the current model (*e.g.*, `m.residues['3']`; see Sequence.residues), in which case all atoms from the residue are used.

- A list of atoms or residues returned by **Model.atom_range()** or **Model.residue_range()**, in which case all atoms from the list are used.
- A `Model` object, in which case all atoms in the model are used.
- A `Selection` object, in which case all atoms in the selection are used.

Features can be any of the classes in the `features` module (see below) or you can create your own classes; see Section 7.1.

`features.Distance(*atom_ids)`
Distance in angstroms between the given two atoms.

`features.Angle(*atom_ids)`
Angle in radians between the given three atoms.

`features.Dihedral(*atom_ids)`
Dihedral angle in radians between the given four atoms.

`features.MinimalDistance(*atom_ids)`
Given an even number of atoms, this calculates the distance between the first two atoms, the third and fourth, and so on, and returns the shortest such pair distance, in angstroms.

`features.SolventAccess(*atom_ids)`
Area (in $\mathring{A}^2$) exposed to solvent of the given atom. Note that this feature cannot be used in optimization, as first derivatives are always returned as zero. Note also that **Model.write_data()** should first be called with `OUTPUT='PSA'` to calculate the accessibility values.

`features.Density(*atom_ids)`
Atomic density (number of atoms within contact_shell of the given atom). Note that this feature cannot be used in optimization, as first derivatives are always returned as zero.

`features.XCoordinate(*atom_ids)`
Value of the x coordinate (in angstroms) of the given atom.

`features.YCoordinate(*atom_ids)`
Value of the y coordinate (in angstroms) of the given atom.

`features.ZCoordinate(*atom_ids)`
Value of the z coordinate (in angstroms) of the given atom.

`features.DihedralDiff(*atom_ids)`
Difference in radians between two dihedral angles (defined by the first four and last four atoms).

Mathematical forms of restraints

Each mathematical form is a Python class, which takes one or features (above) as arguments to act on. group is used to group restraints into "physical feature types" for reporting purposes in **Selection.energy()**, *etc*, and should be a Python object from the `physical` module (see Table 6.1 and Section 6.10.1). You can also create your own mathematical forms by creating new Python classes; see Section 7.1.

Each of the mathematical forms is depicted in Figure 5.1.

`forms.LowerBound(group, feature, mean, stdev)`
Harmonic lower bound (left Gaussian). The given feature is harmonically restrained to be greater than mean with standard deviation stdev. See Eq. A.82.

`forms.UpperBound(group, feature, mean, stdev)`
Harmonic upper bound (right Gaussian). The given feature is harmonically restrained to be less than mean with standard deviation stdev. See Eq. A.83.

`forms.Gaussian(group, feature, mean, stdev)`
Single Gaussian (harmonic potential). The given feature is harmonically restrained to be around mean with standard deviation stdev. See Eq. A.63.

`forms.MultiGaussian(group, feature, weights, means, stdevs)`
Multiple Gaussian. The given feature is restrained by a linear combination of Gaussians. weights, means and stdevs

should all be lists (of the same size) specifying the weights of each Gaussian in the linear combination, their means, and their standard deviations, respectively. See Eq. A.66.

`forms.Factor(group, feature, factor)`
Simple scaling. The given feature value is simply multiplied by factor to yield the objective function contribution.

`forms.LennardJones(group, feature, A, B)`
Lennard-Jones potential. The given feature is restrained by means of a Lennard-Jones potential, with control parameters A and B. See Eq. A.90.

`forms.Coulomb(group, feature, q1, q2)`
Coulomb point-to-point potential. The given feature is restrained by means of an inverse square Coulomb potential created by charges q1 and q2. See Eq. A.87.

`forms.Cosine(group, feature, phase, force, period)`
Cosine potential. The given feature is restrained by a CHARMM-style cosine function, with the given phase shift, force constant and periodicity. See Eq. A.84.

`forms.MultiBinormal(group, features, weights, means, stdevs, correls)`
The given two features (generally both `features.Dihedral`) are simultaneously restrained by a multiple binormal restraint. weights, means, stdevs and correls should all be lists (of the same size). weights specifies the weights of each term in the function. means and stdevs give the mean and standard deviation of each feature for each term, and each element should thus be a 2-element list. correls gives the correlation between the two features for each term. See Eq. A.76.

`forms.Spline(group, feature, open, low, high, delta, lowderiv, highderiv, values)`
Cubic spline potential. The given feature is restrained by an interpolating cubic spline, fitted to values, which should be a list of objective function values. The first element in this list corresponds to feature value low, the last to feature value high, and points in the list are taken to be equally spaced by delta in feature space. The spline can either be open (open = `True`) in which case the first derivatives of the function at the first and last point in values are given by lowderiv and highderiv respectively, or closed (open = `False`) in which case lowderiv and highderiv are ignored. A closed spline 'wraps around' in such a way that feature values low and high are taken to refer to the same point, and is useful for periodic features such as angles. See Eq. A.97.

`forms.NDSpline(group, values)`
Multi-dimensional cubic spline potential. The given feature is restrained by an interpolating multi-dimensional cubic spline, fitted to values, which should be an N-dimensional list of objective function values. (For example, for a 2D spline, it should be a list of lists. The outer list goes over the second feature, and contains one or more rows, each of which is a list which goes over the first feature.) After creating the object, you should then call the `'add_dimension'` function N times:

`NDSpline.add_dimension(feature, open, low, high, delta, lowderiv, highderiv)`
This initializes the next dimension of the multi-dimensional cubic spline. Parameters are as for `'forms.Spline()'`, above. Note that lowderiv and highderiv are used for every spline, for efficiency. (For example, in an x-by-y 2D spline, there will be 'x' splines in the second dimension, each of which could have its own lowderiv and highderiv, but one pair of values is actually used for all 'x' of these splines.)

Restraint violations

When MODELLER optimizes the objective function, the aim is to fulfill all of the restraints as well as possible. In complex cases, this will be difficult or impossible to do, and some of the restraints will not be optimal. In this case, MODELLER reports the deviation of each restraint from the optimum as a 'violation'. There are four kinds of restraint violation used by MODELLER:

- The *heavy violation* is defined as the difference between the current value of the feature, and the global minimum of the same feature according to the restraint's mathematical form.
- The *relative heavy violation* is the heavy violation normalized by dividing by the standard deviation of the global minimum.
- The *minimal violation* is defined as the difference between the current value of the feature, and the nearest minimum of the same feature according to the mathematical form. Where this minimum corresponds to the

global minimum (or for forms which have no well-defined local minimum, such as cubic splines), the minimal violation is the same as the heavy violation.

- The *relative minimal violation* is the minimal violation normalized by dividing by the standard deviation of the local minimum.

Equations for relative heavy violations for most mathematical forms are given in Section A.3.2.

5.3.2 Specification of pseudo atoms

There are virtual and pseudo atoms. A virtual atom is an atom that occurs in the actual molecule, but whose position is not represented explicitly in the MODEL and topology file. A pseudo atom is a position that does not correspond to an actual atom in a molecule, but is some sort of an average of positions of real atoms. Pseudo atoms can be added to the list of restraints by adding the objects below to the Restraints.pseudo_atoms list. Atom ids are as for features, above. The MODELLER pseudo and virtual atom types follow closely the GROMOS definitions.

`pseudo_atom.GravityCenter(*atom_ids)`
Gravity center of all of the supplied atoms.

`pseudo_atom.CH2(*atom_ids)`
Pseudo aliphatic proton on a tetrahedral carbon (>CH2). Not assigned stereospecifically; its position is between the two real protons; defined by the central C and the other two substituents (specified by atom_ids).

`pseudo_atom.CH31(*atom_ids)`
Pseudo aliphatic proton on a tetrahedral carbon (-CH3), defined by the central C and the heavy atom X in X-CH3 (specified by atom_ids); its position is the average of the three real protons.

`pseudo_atom.CH32(*atom_ids)`
Pseudo aliphatic proton between two unassigned -CH3 groups; defined by X in CH3 - X - CH3 and the two C atoms from the two CH3 groups (specified by atom_ids). Its position is the average of the six real protons.

`virtual_atom.CH1(*atom_ids)`
Virtual aliphatic proton on a tetrahedral carbon (->CH), defined by the central C and the three other substituents (specified by atom_ids).

`virtual_atom.CH1A(*atom_ids)`
Virtual aromatic proton on a trigonal carbon (=CH), defined by the central C and the two C atoms bonded to the central C (specified by atom_ids).

`virtual_atom.CH2(*atom_ids)`
Virtual aliphatic proton on a tetrahedral carbon (>CH2) assigned stereospecifically; defined by the central tetrahedral atom and the other two substituents on it (specified by atom_ids).

Example: examples/python/pseudo_atoms.py

```
from modeller import *
from modeller.scripts import complete_pdb
from modeller.optimizers import ConjugateGradients

env = Environ()

env.io.atom_files_directory = ['../atom_files']
log.verbose()
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read in the model
mdl = complete_pdb(env, "1fdn")
rsr = mdl.restraints

# Select all C-alpha atoms
allat = Selection(mdl)
```

```
allca = allat.only_atom_types('CA')

# Create a pseudo atom that is the center of all C-alphas, and activate it
center = pseudo_atom.GravityCenter(allca)
rsr.pseudo_atoms.append(center)

# Constrain every C-alpha to be no more than 10 angstroms from the center
for at in allca:
    r = forms.UpperBound(group=physical.xy_distance,
                         feature=features.Distance(at, center),
                         mean=10.0, stdev=0.1)
    rsr.add(r)

# Constrain the gravity center to the x=0 plane
r = forms.Gaussian(group=physical.xy_distance,
                   feature=features.XCoordinate(center),
                   mean=0.0, stdev=0.1)
rsr.add(r)

# Keep sensible stereochemistry
rsr.make(allat, restraint_type='stereo', spline_on_site=False)

# Optimize with CG
cg = ConjugateGradients()
cg.optimize(allat, max_iterations=100, output='REPORT')
mdl.write(file='1fas.ini')
```

5.3.3 Excluded pairs

You can also exclude certain pairs of atoms from the nonbonded list. These Python objects are added to the Restraints.excluded_pairs list.

`ExcludedPair(atom_id1, atom_id2)`
Excludes the given two atoms from the nonbonded list.

Example: examples/scoring/excluded_pair.py

```
# Demonstrate the use of excluded pairs.

# In this example we approximate a disulfide linkage by creating a distance
# restraint between two SG atoms in CYS residues. Since these atoms are in
# different residues, ordinarily Modeller will calculate a van der Waals
# (soft sphere) interaction between them. We use an excluded pair to prevent
# this interaction from being calculated, as otherwise it will conflict
# with the new distance restraint.

# Note that this is an example only; ordinarily a DISU patch would be used
# to create a disulfide linkage. The DISU patch has the advantage that it
# restrains the angles and dihedrals involved with the SG-SG bond, and also
# excludes these atom pairs from van der Waals interaction.

from modeller import *
from modeller.scripts import complete_pdb
from modeller.optimizers import ConjugateGradients
```

```
env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.edat.dynamic_sphere = True

env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

code = '1fas'
mdl = complete_pdb(env, code)

atom1 = mdl.atoms['SG:3:A']
atom2 = mdl.atoms['SG:22:A']
mdl.restraints.add(forms.Gaussian(group=physical.xy_distance,
                                  mean=2.0, stdev=0.1,
                                  feature=features.Distance(atom1, atom2)))
mdl.restraints.excluded_pairs.append(ExcludedPair(atom1, atom2))

# Retain stereochemistry
atmsel = Selection(mdl)
mdl.restraints.make(atmsel, restraint_type='stereo', spline_on_site=False)

# Optimize the model with CG
cg = ConjugateGradients(output='REPORT')
cg.optimize(atmsel, max_iterations=100)

mdl.write(file=code+'.expair.pdb')
```

5.3.4 Rigid bodies

You can mark groups of atoms as belonging to a rigid body. They will be moved together during optimization, such that their relative orientations do not change. These are created by making a `RigidBody` object and adding it to the Restraints.rigid_bodies list.

Note that all intra-body atom pairs are removed from the nonbonded list, since those distances cannot change. Thus these atom pairs will no longer contribute to any nonbonded interactions, such as Coulomb or Lennard-Jones interactions. See also **Selection.assess_dope()**.

`RigidBody(*atom_ids)`
Creates a new rigid body which contains all of the specified atoms. You can also tune the scale_factor member of the resulting object, which is used to scale the system state vector (used by **ConjugateGradients()** and **QuasiNewton()** optimizations) to rigid body orientation Euler angles (in radians). (Note that no scaling is done for the position of the rigid body; thus the units of this factor are effectively radians/Å.) This can improve optimization convergence in some cases. By default the scaling factor is 1.0; values larger than 1 increase the rotational sampling, while values less than 1 will decrease it.

Example: examples/python/rigid_body.py

```
from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']
mdl = Model(env, file='1fas')

# Keep residues 1-10 in chain A rigid:
```

```
r = RigidBody(mdl.residue_range('1:A', '10:A'))
mdl.restraints.rigid_bodies.append(r)

# Randomize the coordinates of the whole model; the rigid body remains rigid
sel = Selection(mdl)
sel.randomize_xyz(deviation=4.0)
mdl.write(file='1fas.ini')
```

5.3.5 Symmetry restraints

You can restrain two groups of atoms to be the same during optimization of the objective function. This is achieved by adding the sum of squares of the differences between the equivalent distances (similar to distance RMS deviation) to the objective function being optimized. See Equation A.99.

After creating a `Symmetry` object, you can call its `append` function to add additional pairs of groups. This allows some equivalent atoms to be weighted more strongly than others. Finally, add the `Symmetry` object to the Restraints.symmetry list.

`Symmetry(set1, set2, weight)`
Creates a new symmetry restraint which will constrain the interatomic distances in `set1` to be the same as in `set2`. (The `append` function takes the same parameters.) Both sets are just lists of atoms or objects which contain atoms, such as `Residue` or `Selection` objects. Note that each set must contain the same number of atoms. Note also that the order is important. (If using `Selection` objects, the atoms are always sorted in the same order as seen in the PDB file.)

See Section 2.2.12 for an example of using symmetry restraints with the `AutoModel` class.

Example: examples/commands/define_symmetry.py

```
# Example for: Model.symmetry.define()

# This will force two copies of 1fas to have similar mainchain
# conformation.

from modeller import *
from modeller.scripts import complete_pdb
from modeller.optimizers import ConjugateGradients, MolecularDynamics

log.level(1, 1, 1, 1, 0)
env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

def defsym(mdl, seg1, seg2):
    sel1 = Selection(mdl.residue_range(*seg1)).only_mainchain()
    sel2 = Selection(mdl.residue_range(*seg2)).only_mainchain()
    mdl.restraints.symmetry.append(Symmetry(sel1, sel2, 1.0))

# Generate two copies of a segment:
mdl = complete_pdb(env, '2abx', model_segment=('1:A', '74:B'))
mdl.rename_segments(segment_ids=('A', 'B'), renumber_residues=(1, 1))

myedat = EnergyData(dynamic_sphere = False)
atmsel = Selection(mdl)
atmsel.energy(edat=myedat)
atmsel.randomize_xyz(deviation=6.0)
# Define the two segments (chains in this case) to be identical:
defsym(mdl, seg1=('1:A', '74:A'), seg2=('1:B', '74:B'))

# Create optimizer objects
cg = ConjugateGradients()
md = MolecularDynamics(md_return='FINAL')

# Make them identical by optimizing the initial randomized structure
# without any other restraints:
atmsel.energy(edat=myedat)
mdl.write(file='define_symmetry-1.atm')
cg.optimize(atmsel, max_iterations=300, edat=myedat)
mdl.write(file='define_symmetry-2.atm')
atmsel.energy(edat=myedat)

# Now optimize with stereochemical restraints so that the
# result is not so distorted a structure (still distorted
# because optimization is not thorough):
myedat.dynamic_sphere = True
mdl.restraints.make(atmsel, restraint_type='stereo', spline_on_site=False,
                    edat=myedat)
atmsel.randomize_xyz(deviation=3.0)
for method in (cg, md, cg):
    method.optimize(atmsel, max_iterations=300, edat=myedat, output='REPORT')
mdl.write(file='define_symmetry-3.atm')
atmsel.energy(edat=myedat)

# Report on symmetry violations
mdl.restraints.symmetry.report(0.3)

# Create a blank alignment so that superpose uses its 1:1 default
aln = Alignment(env)

mdl = Model(env, file='define_symmetry-3.atm', model_segment=('1:A', '74:A'))
mdl2 = Model(env, file='define_symmetry-3.atm', model_segment=('1:B', '74:B'))
atmsel = Selection(mdl).only_mainchain()
atmsel.superpose(mdl2, aln)
```

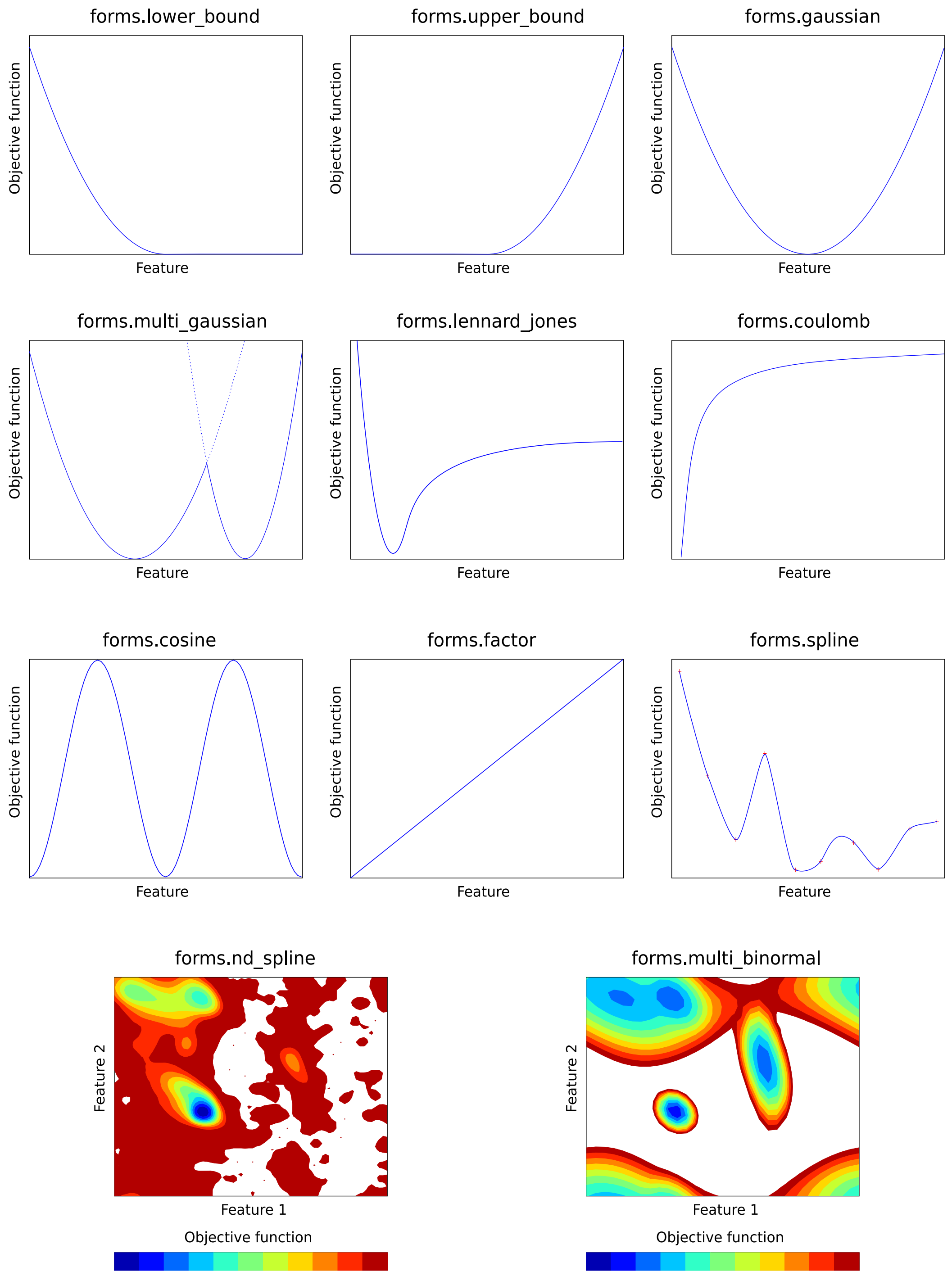

Figure 5.1: *Mathematical forms of restraints.* Each mathematical form generates a contribution to the objective function as a function of one or more features. Note that this contribution is the negative log of the probability density.

Chapter 6

Modeller command reference

6.1 Key for command descriptions

Most commands in Modeller take one or more arguments, usually as Pythonkeywords. For convenience, many of these arguments take a default value if you do not specify them.

Each argument must be of a specific type, either a Modeller class object (*e.g.*, a `Model` or `Alignment`) or one of the standard Python types (integers, strings, floating point numbers, or lists of these types).

Many commands take optional io and/or edat arguments. io should always be an **IOData()** object, and is used by commands that need to read coordinate files, while edat should be an **EnergyData** object, and is used by commands that need to use the energy function. For convenience, if these arguments are not specified, default values are taken from the Modeller environment (Environ.io and Environ.edat respectively).

6.2 The `Environ` class: Modeller environment

The `Environ` class contains most information about the Modeller environment, such as the energy function and parameter and topology libraries (see section 6.1 for more information). Usually it is the first class to be used in a Modeller script, as it provides methods to create the other main classes. In addition, some miscellaneous commands are also provided as methods of the `Environ` class.

6.2.1 Environ() — create a new Modeller environment

```
Environ(rand_seed=-8123, restyp_lib_file='$(LIB)/restyp.lib', copy=None)
```

This creates a new `Environ` object. rand_seed is used to seed the random number generator used throughout Modeller, and should be set to a negative integer between −2 and −50000 if you do not want to use the default value. restyp_lib_file specifies the file to read the residue type library from; it should be a filename or readable file handle (see **modfile.File()**). If unspecified, the default (`'restyp.lib'`) file is used. This file contains the mapping between one-letter residue types and CHARMM and PDB names; see the FAQ Section 3.1, Question 8 for the format of this file.

You can assign the new `Environ` object to the Python variable `'env'` with the following:

```
env = Environ()
```

You can release the object from memory when you no longer need it in standard Python fashion, either by an explicit `del(env)` or by reassigning `env` to some other object.

When you create new Modeller objects (such as `Model` or `Alignment` objects) they require an `Environ` object, which they use for their own default values. Note that each object gets a *copy* of the environment, so it is not affected by any changes you make to the global environment *after* its creation. You can, however, modify the object's own environment directly, by assigning to its `.env` member:

```
env = Environ()
env.io.hetatm = True # New objects will read HETATM records from PDB by default
mdl = model(env) # Create new model object (with hetatm=True)
mdl.env.io.hetatm = False # hetatm is now False, but only for 'mdl'
```

If in doubt, set anything you need to set within `Environ` *before* you create any objects.

6.2.2 Environ.io — default input parameters

This is an `IOData` object, which is used as the default by all routines which take an `io` argument (used for reading coordinate files). See Section 6.4.

6.2.3 Environ.edat — default objective function parameters

This is an `EnergyData` object, which is used as the default by all routines which take an `edat` argument (used to configure the energy function). See Section 6.3.

6.2.4 Environ.libs — MODELLER libraries

This is a `Libraries` object, which contains all of the MODELLER topology and parameter libraries. See Section 6.5.

6.2.5 Environ.schedule_scale — energy function scaling factors

This is used to scale the various contributions to the energy function (if not specified explicitly when calling the energy function or an optimizer). It is a **physical.Values()** object. See Section 2.2.2 for an example.

6.2.6 Environ.dendrogram() — clustering

`dendrogram(matrix_file, cluster_cut)`

This command calculates a clustering tree from the input matrix of pairwise distances. This matrix must be in the PHYLIP format and can be produced by the **Alignment.id_table()**, **Alignment.compare_sequences()**, or **Alignment.compare_structures()** commands. The weighted pair-group average clustering method (as described at **Model.transfer_xyz()**) is used.

The tree is written to the `log` file.

This command is useful for deciding about which known 3D structures are to be used as templates for comparative modeling.

Example: See **Alignment.id_table()** command.

6.2.7 Environ.principal_components() — clustering

`principal_components(matrix_file, file)`

This command calculates principal components clustering for the input matrix of pairwise distances. This matrix must be in the PHYLIP format and can be produced by the **Alignment.id_table()**, **Alignment.compare_sequences()**, or **Alignment.compare_structures()** commands.

The projected coordinates p and q are written to file file. The output file can be used with ASGL to produce a principal components plot.

This command is useful for deciding about which known 3D structures are to be used as templates for comparative modeling.

Example: See **Alignment.id_table()** command.

6.2.8 Environ.system() — execute system command

```
system(command)
```

This command executes the specified operating system command, for example 'rm' or 'ls' on a Unix system, or 'dir' on a Windows machine. This should be avoided in portable scripts, precisely because the available commands differ between operating systems.

6.2.9 Environ.make_pssmdb() — Create a database of PSSMs given a list of profiles

```
make_pssmdb(profile_list_file, pssmdb_name, profile_format='TEXT', rr_file='$(LIB)/as1.sim.mat',
matrix_offset=0.0, matrix_scaling_factor=0.0069, pssm_weights_type='HH1')
```

This command takes a list of profiles, specified in profile_list_file, to calculate their Position Specific Scoring Matrices (PSSM) and create a database of these PSSMs for use in **Profile.scan()**.

The profiles listed in profile_list_file should be in a format that is understood by **Profile.read()**. For instance, like those created by **Profile.build()** or **Alignment.to_profile**. See documentation under **Profile.read()** for help on profile_format.

rr_file is the residue-residue substitution matrix to use when calculating the position-specific scoring matrix (PSSM). The current implementation is optimized only for the BLOSUM62 matrix.

matrix_offset is the value by which the scoring matrix is offset during dynamic programming. For the BLOSUM62 matrix use a value of -450.

pssmdb_name is the name for the output PSSM database.

Example: examples/commands/ppscan.py

```
# Example for: Profile.scan()

from modeller import *

env = Environ()

# First create a database of PSSMs
env.make_pssmdb(profile_list_file = 'profiles.list',
                matrix_offset     = -450,
                rr_file           = '${LIB}/blosum62.sim.mat',
                pssmdb_name       = 'profiles.pssm',
                profile_format    = 'TEXT',
                pssm_weights_type = 'HH1')

# Read in the target profile
prf = Profile(env, file='T3lzt-uniprot90.prf', profile_format='TEXT')

# Read the PSSM database
psm = PSSMDB(env, pssmdb_name = 'profiles.pssm', pssmdb_format = 'text')
```

```
# Scan against all profiles in the 'profiles.list' file
# The score_statistics flag is set to false since there are not
# enough database profiles to calculate statistics.
prf.scan(profile_list_file = 'profiles.list',
         psm               = psm,
         matrix_offset     = -450,
         ccmatrix_offset   = -100,
         rr_file           = '${LIB}/blosum62.sim.mat',
         gap_penalties_1d  = (-700, -70),
         score_statistics  = False,
         output_alignments = True,
         output_score_file = None,
         profile_format    = 'TEXT',
         max_aln_evalue    = 1,
         aln_base_filename = 'T3lzt-ppscan',
         pssm_weights_type = 'HH1',
         summary_file      = 'T3lzt-ppscan.sum')
```

6.3 The EnergyData class: objective function parameters

The EnergyData class is used to configure the objective function, selecting which types of dynamic restraints (see Section 5.3.1) to calculate (*e.g.*, soft-sphere, Coulomb), and how to calculate them (*e.g.*, distance cutoffs).

6.3.1 EnergyData() — create a new set of objective function parameters

```
EnergyData(copy=None, **kwargs)
```

This creates a new EnergyData object. The object will have the default parameters. You can, however, specify any of these parameters when you create the object:

```
edat = EnergyData(contact_shell=7.0)
```

Alternatively, you can set parameters in an existing object:

```
edat.contact_shell = 7.0
```

Many commands use EnergyData objects. However, for convenience, the Environ class also contains an EnergyData object, as Environ.edat. This is used as the default if you do not give an EnergyData parameter, so you can set this to change the objective function used by all functions:

```
env = Environ()
env.edat.contact_shell = 7.0
```

6.3.2 EnergyData.contact_shell — nonbond distance cutoff

This defines the maximal distance (in angstroms) between atoms that flags a non-bonded atom pair. Such pairs are stored in the list of non-bonded atom pairs. Only those non-bonded pairs that are sufficiently close to each other will result in an actual non-bonded restraint. The default value is 4.0 Å.

If undefined (-999), the distance is the maximum of:

- If EnergyData.dynamic_sphere is True, twice the radius of the largest atom multiplied by EnergyData.radii_factor (in the case of the all non-hydrogen atoms model, this is 3.2 Å).
- If EnergyData.dynamic_lennard is True, EnergyData.lennard_jones_switch[1].
- If EnergyData.dynamic_coulomb is True, EnergyData.coulomb_switch[1].
- The maximum cutoff distance requested by any user-defined energy term, if the scaling factor of that term is non-zero (see Section 7.1.3).

The best value for EnergyData.contact_shell must be found in combination with EnergyData.update_dynamic (see also below). Good values are 4Å for EnergyData.contact_shell and 0.39Å for EnergyData.update_dynamic when no Lennard-Jones and Coulomb terms are used; if EnergyData.contact_shell is larger, there would be many pairs in the non-bonded pairs list which would slow down the evaluation of the molecular pdf. If it is too small, however, the increased frequency of the pair list recalculation may slow down the optimization.

This distance is also used for the calculation of atomic density; see section A.3.1.

6.3.3 EnergyData.update_dynamic — nonbond recalculation threshold

This sets the cumulative maximal atomic shift (in angstroms) during optimization that triggers recalculation of the list of atom–atom non-bonded pairs. It should be set in combination with EnergyData.contact_shell. The default value is 0.39 Å.

For soft-sphere overlap, to be absolutely sure that no unaccounted contacts occur, EnergyData.update_dynamic has to be equal to (EnergyData.contact_shell – maximal_overlap_distance) / 2. maximal_overlap_distance is equal to the diameter of the largest atom in the model; it is 3.2 Å in the case of the all non-hydrogen atoms model. This distance is the EnergyData.contact_shell value if a default is requested. Factor 2 comes from the fact that the moves of both atoms can reduce the distance between them.

6.3.4 EnergyData.sphere_stdv — soft-sphere standard deviation

This sets the standard deviation (in angstroms) of the lower bound harmonic potential used for the soft-sphere restraints. The default value is 0.05 Å. See EnergyData.dynamic_sphere.

6.3.5 EnergyData.dynamic_sphere — calculate soft-sphere overlap restraints

If set to `True` (the default), the dynamic soft-sphere overlap restraints are calculated. Note that they are only calculated if the scaled standard deviation of the soft-sphere overlap restraints is greater than zero. It is simpler not to pre-calculate any soft-sphere overlap restraints and to use the dynamically generated restraints alone, although this may be slower. The soft-sphere potential is simply a lower bound harmonic restraint (see Equation A.82), with standard deviation EnergyData.sphere_stdv, dropping to zero at the sum of the two atoms' van der Waals radii.

Soft sphere restraints require the model topology to first be generated with **Model.generate_topology()**.

6.3.6 EnergyData.dynamic_lennard — calculate Lennard-Jones restraints

If set to `True`, dynamic Lennard-Jones restraints are calculated, using equation A.90. The default value is `False`.

Lennard-Jones restraints require the model topology to first be generated with **Model.generate_topology()**.

6.3.7 EnergyData.dynamic_coulomb — calculate Coulomb restraints

If set to `True`, dynamic Coulomb (electrostatic) restraints are calculated, using equation A.87. The default value is `False`.

6.3.8 EnergyData.dynamic_modeller — calculate non-bonded spline restraints

If set to `True`, dynamic MODELLER non-bonded spline restraints are calculated. These include the loop modeling potential and DOPE. The actual library of spline restraints is selected for a model by setting Model.group_restraints. The default value is `False`.

6.3.9 EnergyData.excl_local — exclude certain local pairs of atoms

This specifies whether or not the atoms in a chemical bond, chemical angle, dihedral angle, and in the excluded pairs list respectively are considered in the construction of the non-bonded atom pairs list, and distance restraints. This is especially useful when simplified protein representations are used; *e.g.*, when non-bonded restraints need to be used on $C_{\alpha i}$ – $C_{\alpha i+2}$ terms.

6.3.10 EnergyData.radii_factor — scale atomic radii

This is the scaling factor for the atom radii as read from the `$RADII_LIB` library file. The scaled radii are used only for the calculation of violations of the soft-sphere overlap restraints and by **Model.write_data()**. Note that which radii to first read from the library file are determined by the Topology.submodel variable. The default value is 0.82.

6.3.11 EnergyData.lennard_jones_switch — Lennard-Jones switching parameters

These are the parameters f_1 and f_2 to the Lennard-Jones switching function, which smoothes the potential down to zero; see equation A.90. The default values are

$$6.5, 7.5$$

.

6.3.12 EnergyData.coulomb_switch — Coulomb switching parameters

These are the parameters f_1 and f_2 to the Coulomb switching function, which smoothes the potential down to zero; see equation A.87. The default values are

$$6.5, 7.5$$

.

6.3.13 EnergyData.relative_dielectric — relative dielectric

This sets the relative dielectric ϵ_r, used in the calculation of the Coulomb energy (equation A.87). The default value is 1.0.

6.3.14 EnergyData.covalent_cys — use disulfide bridges in residue distance

This modulates the effect of residue_span_range; if `True`, the disulfide bridges are taken into account when calculating the residue index difference between two atoms (*i.e.*, disulfides make some atom pairs closer in sequence). EnergyData.covalent_cys = `True` is slow and only has an effect when certain statistical non-bonded potentials are used (*i.e.*, EnergyData.dynamic_modeller is `True` and the non-bonded library has been derived considering the disulfide effect). Thus, it should generally be set to `False` (the default).

6.3.15 EnergyData.nonbonded_sel_atoms — control interaction with picked atoms

This specifies the number of atoms that must be selected in each nonbonded pair, in order for the energy term to be calculated. Thus, when only a subset of all atoms is used in energy evaluation, this variable controls the interaction between the picked atoms and the rest of the system. When it is 2, the non-bonded pairs will contain only selected atoms. This means that the optimized atoms will not "feel" the rest of the protein through the non-bonded terms at all.

If EnergyData.nonbonded_sel_atoms is 1 (the default), only one of the atoms in the non-bonded pair has to be a selected atom. This means that the selected region feels the rest of the system through the non-bonded terms, at the expense of longer CPU times.

See Section 2.2.6 for an example.

When all atoms are selected, this variable has no effect.

6.3.16 EnergyData.nlogn_use — select non-bond list generation algorithm

Before calculating dynamic non-bonded restraints, MODELLER determines which of the several routines is most appropriate and efficient for calculating the non-bonded atom pairs list. The user can use this variable to influence the selection, of either a straightforward $\mathcal{O}(n^2)$ search or a cell-based algorithm which has $n \log n$ dependency of CPU time *versus* size n. The latter algorithm is used when the maximal difference in residue indices of the atoms in the current dynamic restraints is larger than EnergyData.nlogn_use, EnergyData.contact_shell is less than 8Å, the necessary number of cells is less than EnergyData.max_nlogn_grid_cells and fits in an integer without overflow, and there is sufficient system memory to store the cells. The default value is 15.

6.3.17 EnergyData.max_nlogn_grid_cells — maximum number of grid cells for NlogN nonbond pairs routine

This sets the maximum number of grid cells that can be used for the NlogN nonbond pairs routine (see EnergyData.nlogn_use). The default value (26214400) prevents the grid sorting algorithm from using more than approximately 200MiB of memory. In most cases this is more than sufficient memory; a grid larger than this is usually a symptom of a badly-performing optimization in any case (the system is blowing apart with huge distances between protein domains).

6.3.18 EnergyData.energy_terms — user-defined global energy terms

This holds a list of Python objects, each of which is used to add user-defined terms to the energy function. See Section 7.1.3.

6.4 The IOData class: coordinate file input parameters

The IOData class stores information useful in reading coordinate files. This is used both for reading models, and for reading coordinate files used for templates.

6.4.1 IOData() — create a new input parameters object

```
IOData(copy=None, **kwargs)
```

This creates a new IOData object, with default parameters. You can, however, specify any of these parameters when you create the object:

```
io = IOData(hetatm=True)
```

Alternatively, you can set parameters in an existing object:

```
io.hetatm = True
```

Many commands use IOData objects. However, for convenience, the Environ class also contains an IOData object, as Environ.io. This is used as the default if you do not give an IOData parameter, so you can set this to change the input configuration used by all functions:

```
env = Environ()
env.io.hetatm = True
```

Please note that IOData.hetatm, IOData.hydrogen, IOData.convert_modres, IOData.hybrid36, IOData.two_char_chain and IOData.water are ignored when reading UHBD files. When reading CHARMM files, IOData.hetatm, IOData.convert_modres, IOData.hybrid36 and IOData.two_char_chain are ignored.

6.4.2 IOData.hetatm — whether to read HETATM records

If set to True, then all non-water 'HETATM' records are read from PDB, mmCIF or BinaryCIF files. (By default, only 'ATOM' records are read. To read waters, use IOData.water.) Note that you will usually need to turn this on when using BLK residues, or if you want to use a PDB, mmCIF, or BinaryCIF file containing ligands.

6.4.3 IOData.hydrogen — whether to read hydrogen atoms

If set to True, then hydrogen atoms are read from PDB, mmCIF, BinaryCIF, or CHARMM files. (By default, only heavy atoms are read.) You will need to turn this on when building all-atom models, although note that the AllHModel class does this for you automatically.

6.4.4 IOData.water — whether to read water molecules

If set to True, then water molecules are read. (For PDB, mmCIF, or BinaryCIF files, this is regardless of whether they are in 'ATOM' or 'HETATM' records.) Ordinarily, they are ignored. (See '${LIB}/restyp.lib' for the definition of a water molecule used by MODELLER.)

6.4.5 IOData.convert_modres — whether to convert modified residues

By default, some special handling is done for certain commonly-used modified residues. The MSE residue type is mapped to the regular MET amino acid, and the SE atom in this residue is mapped to SD. MSE residues are treated as regular amino acids ('ATOM' records) regardless of whether they are marked as 'ATOM' or 'HETATM' in the PDB, mmCIF, or BinaryCIF file (thus, even if the modified residue is marked as 'HETATM' and IOData.hetatm is False, it will still be read in). This behavior can be disabled by setting this variable to False.

6.4.6 IOData.hybrid36 — whether to read PDB files conformant with hybrid-36

Models containing 100,000 or more atoms, or residues numbered 10,000 or higher, cannot be represented in traditional PDB format (as the atom serial number is a fixed 5 digits, and the residue number 4 digits). If this variable is set `True`, the default, MODELLER will work around this problem by reading a slightly-modified PDB format, hybrid-36, which allows for up to 87,440,031 atoms and residue numbers up to 2,436,111. (See also IOData.two_char_chain for similar handling of chain IDs longer than a single character.) This can be set `False` to strictly conform to the traditional PDB format.

Note that MODELLER will always use hybrid-36 to write out PDB files, as hybrid-36 is the same as traditional PDB for small numbers of atoms and residues. (Thus, to maximize compatibility, avoid large atom or residue numbers if possible.)

Use of the mmCIF format is encouraged when working with large systems.

6.4.7 IOData.two_char_chain — whether to read PDB files with two-character chain IDs

Models containing chain names longer than a single character cannot be represented in traditional PDB format as only a single column is allocated for the chain ID. If this variable is set `True`, the default, MODELLER will support two-character chain IDs, by using the otherwise-unused column 21 (between the residue name and chain ID). (See also IOData.hybrid36 for similar handling of large residue and atom numbers.) This can be set `False` to strictly conform to the traditional PDB format, for example when dealing with software such as `AMBER` that uses column 21 for another purpose, such as longer residue names.

Note that MODELLER will always use this scheme to write out PDB files, as it is the same as traditional PDB for single-character chain IDs. (Thus, to maximize compatibility, avoid two-character chain names.)

Use of the mmCIF format is encouraged when working with large systems.

6.4.8 IOData.atom_files_directory — search path for coordinate files

This is a Python list of directories in which to search for coordinate files. (By default, only the current directory is searched.) PDB-style subdirectories (the last two but one characters in the PDB code) are also searched for each directory *e.g.*, 1abc is searched for in the 'ab' subdirectory, pdb4xyz.ent in the 'xy' subdirectory, and the new-style 12-character extended PDB ID PDB_12345678 in the '67' subdirectory.

6.5 The `Libraries` class: stereochemical parameters and molecular topology

6.5.1 Libraries.topology — topology library information

This contains the current topology library information. See **Topology.append()** for more information.

6.5.2 Libraries.parameters — parameter library information

This contains the current parameter library information. See **Parameters.append()** for more information.

6.5.3 Topology.append() — append residue topology library

```
append(file)
```

This command reads residue topologies from the topology library given by file, such as the CHARMM 22 topology file [MacKerell *et al.*, 1998]. This file must include atomic connectivities of residues and patching residues, and the internal coordinates for minimum energy residue conformations. Patching residues modify residues; for example, N-terminus, C-terminus and disulfide bonds are defined by patching the original topology. This information is used for generating the molecular topology and possibly for calculating an initial conformation. To define your entries in the topology library, see the FAQ Section 3.1, Questions 8 and 9.

file can be a filename or a readable file handle (see **modfile.File()**).

This command also sets Topology.submodel appropriately to match the topology library, assuming a suitable header is found in the library file (see Topology.submodel). For example, the default topology for comparative modeling by MODELLER includes only non-hydrogen atoms (Topology.submodel = 3).

The new residue topologies are added to the existing residue topologies. (To replace the old topology, call **Topology.clear()** first.) If the topology for a residue is duplicated, only the last definition is kept.

Not all the features of the CHARMM 22 topology library are implemented in MODELLER, although a CHARMM file should be read in successfully. A variety of topology files for different kinds of models can be prepared by the **Topology.make()** command.

Example: See **Model.patch()** command.

6.5.4 Topology.clear() — clear residue topology library

```
clear()
```

This deletes all topology information from the library.

6.5.5 Topology.read() — read residue topology library

```
read(file)
```

This is shorthand for calling **Topology.clear()** followed by **Topology.append()**.

6.5.6 Parameters.append() — append parameters library

```
append(file)
```

This command reads the parameters from the parameter library given by file, such as the CHARMM 22 parameter file for proteins with all atoms [MacKerell *et al.*, 1998]. The parameters are added to any already in memory. This file contains the values for bond lengths, angles, dihedral angles, improper dihedral angles, and non-bonded interactions. MODELLER relies on slightly modified CHARMM-22 parameters to reproduce the protein geometry in the MODELLER environment. For example, for the default non-hydrogen atoms model, the ω dihedral angle restraints are stronger than the original CHARMM 22 values which apply to the all-hydrogen model. For a sparse discussion of the parameter library, see the FAQ Section 3.1, Question 8.

Note that, in contrast to older versions of MODELLER, the non-bonded spline parameters used in loop modeling are not read by this function. See instead the documentation for the separate `GroupRestraints` class, in section 6.13, for more information.

file can be a filename or a readable file handle (see **modfile.File()**).

Example: See **Model.patch()** command.

6.5.7 Parameters.clear() — clear parameters library

```
clear()
```

This deletes all parameter information from the library.

6.5.8 Parameters.read() — read parameters library

```
read(file)
```

This is shorthand for calling **Parameters.clear()** followed by **Parameters.append()**.

6.5.9 Topology.make() — make a subset topology library

```
make(submodel)
```

This command makes a residue topology library from the most detailed CHARMM topology library, which contains all atoms, including all hydrogens (corresponding to Topology.submodel = 1). There are currently ten residue topologies, all of which are defined in library `$MODELS_LIB`, which is also read in by this function. For example, the default non-hydrogen atom topology is selected by submodel = 3. For each submodel and residue type, the `$MODELS_LIB` library lists those atoms in the full atom set that are part of the specified topology.

This command works by deleting all the entries that contain non-existing atoms from the original topology file. The charge of each removed atom is redistributed equally between the atoms directly bonded to it (if any of these atoms is in turn marked for deletion, the charge is instead placed on that atom's bonded neighbors, and so on). Any remaining charge is then spread around the entire residue, in proportion to the absolute charge of each atom.

One must carefully test topology files produced in this way. Library `$RADII_LIB` must specify atomic radii for each atom in each residue type for each topology model. submodel must be an integer from 1 to 10. On exit from this routine, Topology.submodel is set to submodel.

For more information about the topology library, see the FAQ Section 3.1, Questions 8 and 9.

Example: examples/commands/make_topology_model.py

```
# Example for: topology.make(), topology.write()

# This creates a topology library for heavy atoms from the
# CHARMM all-atom topology library:

from modeller import *

env = Environ()

tpl = env.libs.topology
# Read CHARMM all-atom topology library:
tpl.read(file='${LIB}/top.lib')

# Keep only heavy atoms (TOPOLOGY_MODEL = 3)
tpl.make(submodel=3)

# Write the resulting topology library to a new file:
tpl.write(file='top_heav.lib')
```

6.5.10 Topology.submodel — select topology model type

This is used to select the type of topology model currently in use; see **Topology.make()** for more information. Note that it is not usually necessary to explicitly set Topology.submodel, since it is set for you automatically when you read a topology file (all of the standard MODELLER topology files include a `'* MODELLER topology.submodel'` header from which this information is derived). See **Topology.read()**.

6.5.11 Topology.write() — write residue topology library

```
write(file)
```

This command writes a residue topology library to the specified file. It is usually used after **Topology.make()**.

file can be either a file name or a **modfile.File()** object open in write mode.

Example: See **Topology.make()** command.

6.6 The `Model` class: handling of atomic coordinates, and model building

The `Model` class holds all information about a 3D model (such as its Cartesian coordinates, topology, and optimization information). It also provides methods for reading, writing and transforming the model. Models are also sequences, so all methods and attributes of `Sequence` objects (see Section 6.17) are also available for models (*e.g.*, Sequence.residues, Sequence.chains).

6.6.1 Model() — create a new 3D model

```
Model(env, **vars)
```

This creates a new `Model` object. If used with no parameters, the new model is empty (*i.e.*, it contains no atoms). However, if any keyword arguments are given, they are passed to the **Model.read()** function to read in an initial model. See the **Model.to_iupac()** example.

6.6.2 Model.seq_id — sequence identity between the model and templates

This is set by MODELLER during model building when **Model.transfer_xyz()** is called. It is also read from or written to MODELLER-produced PDB files or mmCIF files (see **Model.write()**).

You should set Model.seq_id explicitly if you want to carry out GA341 assessment on a model not produced by MODELLER, as the sequence identity is not known in this case. See **Model.assess_ga341()**.

6.6.3 Model.resolution — resolution of protein structure

This is the resolution of the protein structure, as read from a PDB `'REMARK 2 RESOLUTION'` line or mmCIF/BinaryCIF file. For NMR structures, and models built by MODELLER, the resolution is set to -1 (undefined).

6.6.4 Model.last_energy — last objective function value

This is the last value of the objective function. It is written into PDB files and mmCIF files (and is read back from MODELLER-produced PDB or mmCIF files); see **Model.write()**.

6.6.5 Model.remark — text remark(s)

This is a text remark, written by MODELLER into PDB files verbatim. You can add multiple remarks by assigning a multi-line Python string to Model.remark. It is up to you to add a suitable `'REMARK'` prefix to your remark so that it is in valid PDB format.

Note that Model.remark is not read in from PDB files by **Model.read()**; it is used only for writing out your own custom remarks.

Example: See **Model.build()** command.

6.6.6 Model.restraints — all static restraints which act on the model

This provides the static restraints themselves, and methods to manipulate them. See section 6.7 for more information.

6.6.7 Model.group_restraints — all restraints which act on atom groups

This can be assigned to a `GroupRestraints` object in order for dynamic restraints on atom groups to be calculated. (If set to `None`, the default, then no such restraints are evaluated.) Note that EnergyData.dynamic_modeller must also be set to `True`.

Example: See **GroupRestraints()** command.

6.6.8 Model.atoms — all atoms in the model

This is a standard Python list of all the atoms in the model. This can be used to query individual atom properties (*e.g.* coordinates) or to specify atoms for use in restraints, *etc.*

Atoms can be individually accessed in two ways:

- A string of the form 'ATOM_NAME:RESIDUE_#[:CHAIN_ID]', where ATOM_NAME is the four character IUPAC atom name as found in a PDB file, RESIDUE_# is a five character residue number as it occurs in the PDB file of a model, and the optional CHAIN_ID is the single character chain id as it occurs in the PDB file. For example, if `'m'` is a `Model` object, the carbonyl oxygen (O) in residue '10A' in chain 'A' is given by `'m.atoms['O:10A:A']'`; if the chain has no chain id, `'m.atoms['O:10A']'` would be sufficient.
- By numeric index, starting from zero, in standard Python fashion. For example, if `'m'` is a `Model` object, `'m.atoms[0]'` is the first atom, `'m.atoms[-1]'` is the last atom, and `'m.atoms[0:10]'` is a list of the first 10 atoms (0 through 9). (Note that the atom numbers in PDB files and MODELLER restraint files start from 1, so will always be 1 larger than any index you use here.)

See Section 6.23 for more information about `Atom` objects. See also Sequence.residues and Sequence.chains for equivalent lists of residues and chains. See also **Model.atom_range()**, for getting a contiguous range of atoms.

6.6.9 Model.point() — return a point in Cartesian space

```
point(x, y, z)
```

This returns an object defining a point in the Cartesian space of this model. See Section 6.22.

Example: See **Selection()** command.

6.6.10 Model.atom_range() — return a subset of all atoms

```
atom_range(start, end)
```

This returns a list of a subset of atoms from the model, from start to end inclusive. Both start and end must be valid atom indices (see Model.atoms). For example, if `'m'` is a `Model` object, `'m.atom_range('CA:1', 'CB:10')'` returns a list of all atoms from the 'CA' atom in residue '1' to the 'CB' atom in residue '10', inclusive. This sublist can be accessed in just the same way as Model.atoms.

6.6.11 Model.residue_range() — return a subset of all residues

```
residue_range(start, end)
```

This returns a list of a subset of residues from the model, from start to end inclusive. Both start and end must be valid residue indices (see Sequence.residues). For example, if `'m'` is a `Model` object, `'m.residue_range('1', '10')'` returns a list of all residues from PDB residue '1' to PDB residue '10', inclusive. This sublist can be accessed in just the same way as Sequence.residues.

6.6.12 Model.get_insertions() — return a list of all insertions

```
get_insertions(aln, minlength, maxlength, extension, include_termini=True)
```

This returns a list of all insertions (*i.e.*, residue ranges in which the model sequence, which must be the last in the alignment aln, is aligned only with gaps in the other sequences). Each residue range is extended by extension residues either side of the insertion, and is only returned if it is at least minlength residues long, but not longer than maxlength.

If include_termini is `False`, any residue range that includes a chain terminus is excluded from the output.

See also **Model.get_deletions()**, **Model.loops()**.

Example: See **Selection()** command.

6.6.13 Model.get_deletions() — return a list of all deletions

```
get_deletions(aln, extension, include_termini=True)
```

This returns a list of all deletions (*i.e.*, residue ranges in which gaps in the model sequence, which must be the last in the alignment aln, are aligned to residues in the other sequences). Each residue range is extended by extension residues either side of the deletion.

If include_termini is `False`, any residue range that includes a chain terminus is excluded from the output.

See also **Model.get_insertions()**, **Model.loops()**.

Example: See **Selection()** command.

6.6.14 Model.loops() — return a list of all loops

```
loops(aln, minlength, maxlength, insertion_ext, deletion_ext, include_termini=True)
```

This returns a list of all loops, by calling **Model.get_insertions()** and **Model.get_deletions()**.

If include_termini is `False`, any residue range that includes a chain terminus is excluded from the output.

Example: See **Selection()** command.

6.6.15 Model.read() — read coordinates for MODEL

```
read(file, model_format='PDB_ANY', model_segment=('FIRST:@', 'LAST:'), io=None,
keep_disulfides=False)
```

This command reads the atomic coordinates, atom names, residue names, residue numbers[1], isotropic temperature factors and segment specifications for MODEL, assigns residue types, and defines the dihedral angles listed in the `$RESDIH_LIB` library. For `CHARMM` and `UHBD` file formats, it also reads the atomic charges. However, it does not assign Charmm and Modeller atom types, internal coordinates, charges (in the case of the `'PDB'`, `'MMCIF'` or `'BCIF'`formats), or patches (such as disulfides); to make these assignments, which are necessary for almost all energy commands, use **Model.generate_topology()**.

file can be a filename or a readable file handle (see **modfile.File()**).

[1]for mmCIF or BinaryCIF format, author-provided residue numbers (`_atom_site.auth_seq_id`) are used if available rather than the canonical sequence id (`_atom_site.label_seq_id`)

The 'PDB_ANY' file format (the default) will read any PDB-like file; it will read an mmCIF file if the filename ends in '.cif', or a BinaryCIF file if the filename ends in '.bcif'; otherwise it will read a PDB file. (The 'PDB_OR_MMCIF' format is similar except that it will not read BinaryCIF.)

The PDB residue type 'HIS' is assigned the CHARMM residue type 'HSD', which is the neutral His with H on ND1. The PDB types 'ASP' and 'GLU' are assigned the corresponding charged CHARMM residue types, as are 'LYS' and 'ARG'. These conventions are relevant only if Coulomb terms and/or hydrogens are used.

Certain noncanonical residues (such as MSE) are automatically mapped to their closest canonical types by default; see IOData.convert_modres for more details.

model_segment sets the beginning and ending residue identifiers for the contiguous sequence of residues to be read from the PDB, mmCIF or BinaryCIF file (this option does not work yet for the other file formats). The format of residue identifiers is described in Section B.1.

keep_disulfides, if set to True, will preserve any disulfide information already stored in the model. (The default is False, which discards it.) This should only be used if the model being read in is guaranteed to have the same sequence and atom ordering as the model in memory. This is primarily used by the AutoModel class to copy SSBOND records from the initial model (.ini file) to final output models.

Note that this command reads in the model file directly, and does no special handling to ensure the file is suitable for energy evaluations (*e.g.*, that it has no missing atoms). If you want to read in a PDB file from PDB or generated from an experiment or some other program, it is recommended that you use the **complete_pdb()** script instead.

For PDB, mmCIF or BinaryCIF files with alternate locations (characters in column 17 of PDB ATOM or HETATM records; '_atom_site.label_alt_id' in mmCIF or BinaryCIF), MODELLER reads only the first alternate location encountered for each residue. (This differs from older versions of MODELLER, which would read only alternate locations marked with A or 1.)

For PDB files that contain multiple models (each starting with a 'MODEL' record and ending with an 'ENDMDL' record), only the first model is read. For mmCIF or BinaryCIF files, only models in the first 'data_' block and with '_atom_site.pdbx_PDB_model_num' equal to 1 are read.

MODELLER understands hybrid-36 notation, so can read PDB files containing residues numbered 10,000 or higher, or 2-character chain names. See IOData.hybrid36 and IOData.two_char_chain.

The model's R value is read from appropriately formatted PDB REMARKs or the mmCIF/BinaryCIF '_refine' or '_pdbx_refine' categories and is available as Sequence.rfactor. The first valid R value (not free R) found in the PDB file is used.

Some MODELLER-specific REMARKs are also read from PDB files if they are present. See **Model.write()** for a list of such REMARKs. The same information is read from mmCIF or BinaryCIF files from MODELLER-specific categories.

Certain residues (*e.g.*, waters or HETATM records) can be skipped when reading the file, and some modified residues are automatically converted to the nearest equivalent standard amino acids; see **IOData()**.

This command can raise a FileFormatError if the atom file format is invalid.

Example: examples/commands/read_model.py

```
# Example for: Model.read(), Model.write()

# This will read a PDB file and write both CHARMM and mmCIF atom files without
# atomic charges or radii. For assigning charges and radii, see the
# all_hydrogen.py script.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']

mdl = Model(env)
mdl.read(file='1fas')
mdl.write(file='1fas.crd', model_format='CHARMM')
mdl.write(file='1fas.cif', model_format='MMCIF')
```

6.6.16 Model.build_sequence() — build model from a sequence of one-letter codes

`build_sequence(sequence, special_patches=None, patch_default=True, blank_single_chain=False)`

This builds a new model (overwriting any existing one) of the given sequence, specified as one-letter codes. The sequence can also contain '/' characters to build multi-chain models. The coordinates of the model are automatically constructed using **Model.build()**.

special_patches, blank_single_chain, and patch_default can be used to adjust the topology; see **Model.generate_topology()** for more details.

See also **Alignment.append_sequence()**.

Example: examples/commands/build_sequence.py

```
# This demonstrates the use of Alignment.append_sequence() and
# Model.build_sequence() to build residue sequences from one-letter codes

from modeller import *
env = Environ()

# Read parameters (needed to build models from internal coordinates)
env.libs.topology.read('${LIB}/top_heav.lib')
env.libs.parameters.read('${LIB}/par.lib')

# Create a new empty alignment and model:
aln = Alignment(env)
mdl = Model(env)

# Build a model from one-letter codes, and write to a PDB file:
mdl.build_sequence("AFVVTDNCIK/CKYTDCVEVC")
mdl.write(file='sequence.pdb')

# Build an alignment from one-letter codes
aln.append_sequence("AF---VVTDN---CIKCK------")
aln.append_sequence("-------AFVVTDN--CI--K-CK")
# Set alignment information, and write to file:
aln[0].code = 'seq1'
aln[1].code = 'seq2'
aln.write(file='sequence.ali')
```

6.6.17 Model.write() — write MODEL

`write(file, model_format='PDB', no_ter=False, extra_data='')`

This command writes the current MODEL to a file in the selected format.

file can be a filename or a writeable file handle (see **modfile.File()**).

If you want to only write out a subset of the atoms, see **Selection.write()**.

Setting model_format to 'PDB' writes out files in the Protein Data Bank (PDB) format. Note that the isotropic temperature factor (B_{iso}) field can be set by **Selection.energy()** or **Model.write_data()**.

'MMCIF' writes out files in the new PDBx or Macromolecular Crystallographic Information File (mmCIF) format.

Note that if the model contains 100,000 or more atoms, residues numbered 10,000 or higher, or chain names longer than a single character, it cannot be represented in traditional PDB format (as the atom serial number is a fixed 5 digits, the residue number 4 digits, and the chain ID a single character). MODELLER will work around this by using hybrid-36 notation, which is understood by many PDB readers. (Note that this is always done for large systems, regardless of the setting of IOData.hybrid36 and IOData.two_char_chain.) In general, however, mmCIF output is recommended for large systems.

The 'GRASP' format is the same as the 'PDB' format, except that it includes two special lines at the top of the file and the atomic radii and charges in the columns following the Cartesian coordinates of atoms. This format is useful for input to program GRASP, written by Anthony Nicholls in the group of Barry Honig at Columbia University [Nicholls *et al.*, 1991]. For GRASP output, the atomic radii are needed. This usually means using the **complete_pdb()** script rather than **Model.read()** to read in any original PDB file.

If the model contains any disulfide bridges, CONECT and SSBOND records for each bridge are included when writing out PDB files (note that CONECT records for other contacts, such as between HETATM residues, are not included). For mmCIF files, equivalent _struct_conn records are written.

A number of MODELLER-specific REMARKs are added when writing out PDB files (and are parsed when reading such files back in with **Model.read()**). For mmCIF files, the same information is stored in the '_modeller' or '_modeller_blk' categories (shown in parentheses below):

- REMARK 6 MODELLER OBJECTIVE FUNCTION (_modeller.objective_function): the value of Model.last_energy.
- REMARK 6 MODELLER BEST TEMPLATE % SEQ ID (_modeller.best_template_pct_seq_id): the value of Model.seq_id.
- REMARK 6 MODELLER BLK RESIDUE (_modeller_blk.resid): for each residue treated as a BLK, the residue number and chain ID. (This ensures that when the model is read back in, those BLK residues are still treated as BLK, even if the topology file contains a definition for that residue type.)

The contents of Model.remark are also added to PDB files verbatim.

If no_ter is set to True then no TER records are written to PDB files (normally these are added at the end of each amino acid chain).

If extra_data is given, its contents are also added to PDB or mmCIF files verbatim (so it must be formatted correctly for each format).

Example: See **Model.read()** command.

6.6.18 Model.clear_topology() — clear model topology

```
clear_topology()
```

This removes any information from the model about covalent topology and primary sequence. See also **Model.generate_topology()**.

6.6.19 Model.generate_topology() — generate MODEL topology

```
generate_topology(alnseq, patch_default=None, blank_single_chain=False, io=None)
```

This command calculates the model's covalent topology (*i.e.*, atomic connectivity) and internal coordinates, and assigns Charmm atom types, Modeller atom types for non-bonded spline restraints, atomic charges, and atomic radii.

The residue sequence to generate is taken from the alignment sequence given by alnseq.

The sequence is added to the model as a new chain; if you want to first remove any existing chains, call **Model.clear_topology()** prior to this command.

A sequence in the alignment can use any residue listed in the single-character code column of the `$RESTYP_LIB` library (`'modlib/restyp.lib'`). Examples of non-standard residue types include water (`'w'`), zinc (`'z'`), calcium (`'3'`), heme (`'h'`), and many others. If you wish to use patch residues, use **Model.patch()** subsequently. You can generate multiple chains by including chain break characters '/' in the alignment. A chain break prevents Modeller from connecting two otherwise adjacent residues.

By default, the chains are labeled `'A'`, `'B'`, `'C'` and so on. However, if only one chain is generated, and blank_single_chain is set `True`, it is given a blank chain ID. Residues in each chain are numbered sequentially, starting at 1. If you want to use the chain IDs and residue numbers from an existing PDB file, use **Model.res_num_from()**.

If patch_default is `True`, each chain in the sequence is patched with the `'NTER'` and `'CTER'` patches (see **Model.patch()**). These patches are applied to the first and last residue respectively in a connected chain of residues (generally this excludes residues that are not in the amino acid chain, such as HETATM residues).

The **Model.generate_topology()** command generates only the topology of the model, not its Cartesian coordinates; the Cartesian coordinates are assigned by the **Model.build()**, **Model.transfer_xyz()**, or **Model.read()** commands.

In general, the **Model.generate_topology()** command has to be executed before any energy commands (**Selection.energy()**, **Selection.hot_atoms()**) or optimizations (Section 6.11).

The variables Sequence.atom_file, io and Topology.submodel are necessary only when residues lacking topology information (*e.g.*, `'BLK'` residues) are present in the sequence. In that case, the template PDB files are read in.

Example: See **Model.patch()** command.

6.6.20 Model.make_valid_pdb_coordinates() — make coordinates fit in PDB format

```
make_valid_pdb_coordinates()
```

This command ensures that all of the model's coordinates can be written out to a PDB file. Since the PDB format is fixed width, there is a maximum size beyond which x, y or z coordinates cannot be represented; they cannot be smaller than -999.999 or larger than 9999.999. This command ensures that the coordinates lie in this range by simply reflecting them if necessary.

6.6.21 Model.write_psf() — write molecular topology to PSF file

```
write_psf(file, xplor=True)
```

This command writes the current model topology (which must have already been created via **Model.generate_topology()** or **complete_pdb**) to a Charmm or X-PLOR format PSF file or file handle (see **modfile.File()**).

PSF files contain information on all atoms in the model and their types, plus all the connectivity (bonds, angles, *etc*). By default X-PLOR format PSF files are written, in which each atom has a defined type name. These are more flexible than Charmm format PSF files, in which each atom type is specified numerically.

PSF files are generally required in combination with binary format trajectory files, as written by **actions.CHARMMTrajectory()**.

Example: See **actions.CHARMMTrajectory()** command.

6.6.22 Model.patch() — patch MODEL topology

patch(residue_type, residues)

This command uses a CHARMM patching residue to patch the topology of the MODEL. CHARMM patch rules are observed.

residue_type is the type of the patching residue (PRES entry in the topology library), such as 'DISU', 'NTER', 'CTER', *etc.* You do not have to apply explicitly the N- and C-terminal patches to protein chains because the 'NTER' and 'CTER' patches are applied automatically to the appropriate residue types at the termini of each chain at the end of each **Model.generate_topology()** command.

residues should be one or more Residue objects to be patched. The first residue is the patched residue 1, the second residue is the patched residue 2, *etc*; for example, the 'DISU' patching residue has two patched Cys residues while the 'ACE' patching residue has only one patched residue. The order of the residue identifiers here has to match the definition of the patching residue in the topology library.

It is not allowed to patch an already patched residue. Since the N- and C-terminal residues of each chain are automatically patched with the 'NTER' and 'CTER' patching residues, respectively, a user who wants to patch the N- or C-terminal residues with other patches, should turn the default patching off before executing **Model.generate_topology()**. This is achieved by setting patch_default = False.

Example: examples/commands/patch.py

```
# Example for: Model.patch(), topology(), parameters.read()

# This will define a CYS-CYS disulfide bond between residues 3 and 22.

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Create the disulfide bond:
def patches(mdl):
    mdl.patch(residue_type='DISU', residues=(mdl.residues['3:A'],
                                             mdl.residues['22:A']))
# Read the sequence:
code = '1fas'
mdl = complete_pdb(env, code, special_patches=patches)

# Create the stereochemical restraints
sel = Selection(mdl)
mdl.restraints.make(sel, restraint_type='stereo', spline_on_site=False)

# Calculate the energy to test the disulfide:
sel.energy()
```

6.6.23 Model.patch_ss_templates() — guess MODEL disulfides from templates

patch_ss_templates(aln, io=None)

This command defines and patches disulfide bonds in the MODEL using an alignment of the MODEL sequence with one or more template structures. The MODEL sequence has to be the last sequence in the alignment, aln. The template structures are all the other proteins in the alignment. All Cys–Cys pairs in the target sequence that are aligned with at least one template disulfide are defined as disulfide bonds themselves. (Template disulfide bridges are assumed to exist between all pairs of Cys residues whose SG–SG distances are less than 2.5Å. PDB annotations such as SSBOND are not used.) The covalent connectivity is patched accordingly.

This command should be run after **Model.generate_topology()** and before **Restraints.make()** to ensure that the disulfides are restrained properly by the bond length, angle, and dihedral angle restraints and that no SG–SG non-bonded interactions are applied.

The disulfide bond, angle and dihedral angle restraints have their own physical restraint type separate from the other bond, angle and dihedral angle restraints (Table 6.1).

Example: examples/commands/patch_disulfides.py

```
# Example for: Model.patch_ss_templates() and Model.patch_ss()

# This will patch CYS-CYS disulfide bonds using disulfides in aligned templates:

from modeller import *

log.verbose()
env = Environ()
env.io.atom_files_directory = ['.', '../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read the sequence, calculate its topology, and coordinates:
aln = Alignment(env, file='toxin.ali', align_codes=('2ctx', '2abx'))
# Superpose the two template structures without changing the alignment.
# This is for TRANSFER_XYZ to work properly. It relies on not reading
# the atom files again before TRANSFER_XYZ.
aln.malign3d(fit=False) # This is for TRANSFER_XYZ to work properly.
# Restore the alignment, and add in the model sequence, 1fas:
aln.clear()
aln.append(file='toxin.ali', align_codes=('2ctx', '2abx', '1fas'))
mdl = Model(env)
mdl.generate_topology(aln['1fas'])
mdl.transfer_xyz(aln)
mdl.build(initialize_xyz=True, build_method='INTERNAL_COORDINATES')
mdl.write(file='1fas.noSS')
# Create the disulfide bonds using equivalent disulfide bonds in templates:
mdl.patch_ss_templates(aln)

# Create the stereochemical restraints
sel = Selection(mdl)
mdl.restraints.make(sel, restraint_type='stereo', spline_on_site=False)

# Calculate energy to test the disulfide restraints (bonds, angles, dihedrals):
sel.energy()

mdl.read(file='1fas.noSS')
# Create the disulfide bonds guessing by coordinates
mdl.patch_ss()

# Create the stereochemical restraints
```

```
mdl.restraints.make(sel, restraint_type='stereo', spline_on_site=False)

# Calculate energy to test the disulfide restraints (bonds, angles, dihedrals):
sel.energy()
```

6.6.24 Model.patch_ss() — guess MODEL disulfides from model structure

`patch_ss()`

This command defines and patches disulfide bonds in MODEL using MODEL's current structure. A disulfide bridge is declared between all pairs of Cys residues whose SG–SG distances are less than 2.5Å. The covalent connectivity is patched accordingly.

This command should be run after **Model.read()** and before optimization to ensure that the disulfides are fixed properly and that no SG–SG non-bonded interactions are applied.

Topology.submodel is needed to make sure the correct atomic radii are used in CYS–CYS patching.

Example: See **Model.patch_ss_templates()** command.

6.6.25 Model.build() — build MODEL coordinates from topology

`build(build_method, initialize_xyz)`

Requirements: topology file & parameters file & MODEL topology

This command builds Cartesian coordinates of the MODEL.

If initialize_xyz is `True`, all coordinates are built. Otherwise only the undefined coordinates are built. The latter is useful because some coordinates may be undefined after the **Model.read()** or **Model.transfer_xyz()** command. The undefined coordinates have a value of $-999.$ when written to a PDB file.

If build_method is `'INTERNAL_COORDINATES'`, the Cartesian coordinates are built from the ideal values of the internal coordinates as obtained from the IC entries in the residue topology library[2] If an appropriate IC entry does not exist, the ideal value of the internal coordinate is calculated from the corresponding energy term in the parameter library. If some coordinates still cannot be built, the N and C mainchain atoms are placed near a point 1/3 of the way along the vector between CA atoms in adjacent residues, if possible, and then internal coordinate generation is tried again (this helps when using structures containing only CA atoms). If this still fails, mainchain atoms are then 'invented' (placed close to neighboring atoms, or near the origin if there are no neighbors) and then internal coordinate generation is tried again (this helps to create stereochemically correct sidechains, which would otherwise be greatly distorted). If even this fails, any remaining atoms are 'invented'.

If build_method is `'3D_INTERPOLATION'`, the Cartesian coordinates are built by linearly interpolating between the two defined atoms that span the contiguous undefined segment of atoms. In this mode, both the mainchain and sidechain conformations of all inserted residues are random and distorted. This build-up mode is useful because it may eliminate a knot and minimize the extended nature of the insertion obtained by build_method = `'INTERNAL_COORDINATES'`. In the end, the coordinates of each of the interpolated atoms are slightly randomized ($\pm 0.2\AA$) to prevent numerical problems with colinear angles and colinear dihedral angles. If one or both of the spanning atoms are undefined, the `'ONE_STICK'` option (below) is used.

[2]If no atoms in the first chain have coordinates, internal coordinate generation is seeded by placing the first atom at the origin, the second along the x axis, and the third in the xy plane. If no atoms in subsequent chains have coordinates, the first atom is placed 2 angstroms along each of the x, y and z axes from the last atom in the previous chain, and the second and third atoms placed in the same way as for the first chain.

If build_method is 'ONE_STICK', the Cartesian coordinates are built by "growing" them linearly out of the N-terminal spanning atom (C-terminal atom for the undefined N-terminal), away from the gravity center of all the defined atoms. If there are no spanning atoms, the spanning atom is defined randomly.

If build_method is 'TWO_STICKS', the loop is broken into two equal pieces and the 'ONE_STICK' algorithm is applied to both halves of the loop separately.

Example: examples/commands/build_model.py

```
# Example for: Model.build()
# This will build a model for a given sequence in an extended conformation.

from modeller import *
env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read the sequence from a file (does not have to be part of an alignment):
aln = Alignment(env, file='toxin.ali', align_codes='1fas')
# Calculate its molecular topology:
mdl = Model(env)
mdl.generate_topology(aln['1fas'])
# Calculate its Cartesian coordinates using internal coordinates and
# parameters if necessary:
mdl.build(initialize_xyz=True, build_method='INTERNAL_COORDINATES')

# Add PDB remarks for readability
mdl.remark = """REMARK   4 Extended-chain model of 1fas
REMARK   4 Built from internal coordinates only"""

# Write the coordinates to a PDB file:
mdl.write(file='1fas.ini')
```

6.6.26 Model.transfer_xyz() — copy templates' coordinates to MODEL

`transfer_xyz(aln, cluster_cut=-1.0, cluster_method='RMSD', io=None)`

This command transfers coordinates of the equivalent atoms and their isotropic temperature factors (B_{iso}) from the template structures to the model.

The target sequence must be the last protein in the alignment, aln, and has to be the same as the model sequence. The template structures are all the other proteins in the alignment.

Before transferring coordinates, the template structures generally have to be explicitly least-squares superposed onto each other. This is most conveniently achieved with the **Alignment.malign3d()** command called just before **Model.transfer_xyz()**. This is an important difference relative to MODELLER-3, which did not require explicit superposition by the user. Note, however, that the `AutoModel` class script does this superposition automatically if you set AutoModel.initial_malign3d to `True`.

If cluster_cut is greater than 0, the transferred coordinates and $B_{iso}$are the average of the templates in the largest cluster of the atoms. This cluster is obtained as follows (it only works when all templates and the target have exactly the same topology): For each residue position separately, calculate the maximal inter-template equivalent atom–atom distances (cluster_method = 'MAXIMAL_DISTANCE') or atomic RMS deviation (cluster_method = 'RMSD') for all template–template comparisons. Use the weighted pair-group average clustering method (the same as in the **Environ.dendrogram()** command) to obtain the clustering tree for the given residue position. Find the clusters that contain residues joined above cluster_cut angstroms (1Å is a good value). Use the largest cluster in the averaging for the target coordinates. The number of residue positions at which each template contributes to the consensus is written to the `log` file (`'The largest cluster occupancy'`). Sometimes the first template contributes many more times than the rest of the templates. This results from having many residue positions where all "clusters" have one template only (the first cluster/template is then picked by default). This artifact can be corrected by specifying a larger cluster_cut. Two additional data files are also produced: `nmemb.dat` contains one line for each residue in the model, which lists the residue number, the number of clusters detected, and the number of templates in the largest cluster. `occupancy.dat` lists, for each residue, the indices of the templates in the largest cluster.

If cluster_cut is less than or equal to 0, the transferred coordinates and (B_{iso}) for a given target atom are the average of the coordinates of all the equivalent template atoms. cluster_method is ignored.

Both kinds of averaging, but especially the cluster averaging, are useful for deriving a consensus model from an ensemble of models of the same sequence. If the consensus model is optimized by the conjugate gradients method, it frequently has a significantly lower value of the objective function than any of the contributing models. Thus, the construction of a consensus model can also be seen as part of an efficient optimization. The reason why consensus construction frequently results in better models is that the consensus model generally picks the best (*i.e.*, most frequent) conformation for the regions that are variable in the individual models, while it is very unlikely that a single model will have optimal conformation in all of the variable regions. The consensus construction may not work when two or more locally optimal conformations are inconsistent with each other (*e.g.*, because of the atom overlaps).

Two atoms are equivalent if they have exactly the same name and are in the equivalent residues. Note that the `$ATMEQV_LIB` library of equivalent residue–residue atom pairs, which is used in the construction of homology-derived distance restraints, is not used here. The atom names in the target may not correspond to the atom names in the template files. In such a case, if you want to copy the template atoms' coordinates, you have to edit the atom names in the template atom files so that they correspond to the MODELLER atom names (which you can see in the `.ini` atom file). At least for water molecules, this is usually better than letting the optimizer deal with grossly incorrect starting positions.

Atoms which do not have an equivalent in any template are flagged on exit from this method as 'undefined' (by setting their coordinates to -999, and their B_{iso} to 0.0). The coordinates of the undefined atoms of the model can be set with the **Model.build()** command, which relies on the internal coordinates specified in the residue topology library or on various types of geometric interpolation and extrapolation.

Example: examples/commands/transfer_xyz.py

```
# Example for: Model.transfer_xyz()

# This will build a model for a given sequence by copying
# coordinates from aligned templates. When the templates
# have the same sequence as the target, this procedure ensures
# that the new model corresponds to the MODELLER topology library.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['.', '../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read the sequence and calculate its topology:
aln = Alignment(env, file='toxin.ali', align_codes=('2ctx', '2nbt'))
aln.malign3d(fit=False)
aln.append(file='toxin.ali', align_codes='1fas')
mdl = Model(env)
mdl.generate_topology(aln['1fas'])
```

```
# Assign the average of the equivalent template coordinates to MODEL:
mdl.transfer_xyz(aln)
# Get the remaining undefined coordinates from internal coordinates:
mdl.build(initialize_xyz=False, build_method='INTERNAL_COORDINATES')

# Write the final MODEL coordinates to a PDB file:
mdl.write(file='1fas.ini')
```

6.6.27 Model.res_num_from() — residue numbers from MODEL2 to MODEL

`res_num_from(mdl, aln)`

This command transfers residue numbers and chain ids from mdl to the model. mdl and the current model must correspond to the first and second protein in the alignment, aln, respectively.

Example: examples/commands/transfer_res_numb.py

```
# Example for: Model.res_num_from()

# This will transfer residue numbers and chain ids from model2 to model.

from modeller import *

log.level(output=1, notes=1, warnings=1, errors=1, memory=0)
env = Environ()
env.io.atom_files_directory = ['../atom_files']

# Read an alignment for the transfer
aln = Alignment(env, file='toxin.ali', align_codes=('2ctx', '1fas'))
# Read the template and target models:
mdl2 = Model(env, file='2ctx')
mdl  = Model(env, file='1fas')
# Transfer the residue and chain ids and write out the new MODEL:
mdl.res_num_from(mdl2, aln)
mdl.write(file='1fas.ini')
```

6.6.28 Model.rename_segments() — rename MODEL segments

`rename_segments(segment_ids, renumber_residues=[])`

This assigns single character PDB chain IDs (from segment_ids; there should be as many elements in segment_ids as there are chains in the current MODEL). The residues in each chain are also renumbered consecutively, starting with the corresponding element from renumber_residues if provided, or the existing first residue number otherwise.

See also Chain.name for assigning chain IDs individually without also renumbering residues.

Example: examples/commands/rename_segments.py

```
# Example for: Model.rename_segments()

# This will assign new PDB single-character chain id's to all the chains
# in the input PDB file (in this example there are two chains).

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']
mdl = Model(env, file='2abx')

# Assign new segment names and write out the new model:
mdl.rename_segments(segment_ids=('X', 'Y'))
mdl.write(file='2abx-renamed.pdb')
```

6.6.29 Model.to_iupac() — standardize certain dihedral angles

`to_iupac()`

This routine swaps specific pairs of atoms within some residues of MODEL so that certain dihedral angles are within ±90°, satisfying the IUPAC convention [IUPAC-IUB, 1970,Kendrew *et al.*, 1970]. These residues, pairs of atoms, and dihedral angles are:

- Phe, Tyr: (CD1, CD2), (CE1, CE2); χ_2;
- Asp: (OD1, OD2); χ_2;
- Glu: (OE1, OE2); χ_3;
- Arg: (NH1, NH2); χ_4.

It is possible that for distorted sidechains, neither of the two possibilities satisfies the IUPAC convention. In such a case, a warning message is written to the `log` file.

Example: examples/commands/iupac_model.py

```
# This will swap certain atom names in some planar sidechains to satisfy
# the IUPAC convention.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']
log.level(1, 1, 1, 1, 0)

mdl = Model(env, file='2abx')
mdl.to_iupac()
mdl.write(file='2abx.iup')
```

6.6.30 Model.reorder_atoms() — standardize order of MODEL atoms

`reorder_atoms()`

Requirements: topology library

This routine reorders atoms within the residues of MODEL so that they follow the order in the current residue topology library.

Example: examples/commands/reorder_atoms.py

```
# Example for: Model.reorder_atoms()

# This will standardize the order of atoms in the model.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']

# Order the atoms according to a topology library:
env.libs.topology.read(file='$(LIB)/top_heav.lib')

mdl = Model(env, file='1fas')
mdl.reorder_atoms()
mdl.write(file='1fas.ini1')
```

6.6.31 Model.orient() — center and orient MODEL

`orient()`

Output: OrientData object

This command translates the MODEL so that its gravity center is at the origin of the coordinate system and that the three principal axes of the model's inertia ellipsoid correspond to the x, y, and z axes of the coordinate system. It may even be used for approximate superposition if molecules have a similar non-spherical shape. Information about the principal axes is written to the `log` file.

On successful completion, an OrientData object is returned; for instance, if you save this in a variable 'r', the following data are available:

- `r.translation`; the translation used to transform mdl to the center of mass
- `r.rotation`; the rotation matrix used to transform mdl (applied after the translation)

Example: examples/commands/orient_model.py

```
# Example for: Model.orient()

# This will orient the model along the principal axes of the inertia ellipsoid:

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']
mdl = Model(env)
mdl.read(file='1fas')
r = mdl.orient()
mdl.write(file='1fas.ini')

print("Translation: " + str(r.translation))
```

6.6.32 Model.write_data() — write derivative model data

```
write_data(output, file=None, surftyp=1, neighbor_cutoff=6.0, accessibility_type=8,
probe_radius=1.4, psa_integration_step=0.1, dnr_accpt_lib='$LIB/donor_acceptor.lib', edat=None)
```

Requirements: topology file

This command calculates and writes out the selected type(s) of data (from the list below) about the model. If file is specified and is non-empty, each property is written out to a file using file as the root of the file name. The last such property is also assigned to the B_{iso} field of each atom of the model (in the case of residue properties, each atom in the residue gets the value of the property). This can be accessed as Atom.biso, or written out with **Model.write()** to a PDB file, where it appears as the temperature factor.

The data to be calculated are specified by concatenating the corresponding keywords in the output variable:

- `'ALL'`: All types of data are written.
- `'PSA'`: The atomic and residue solvent accessibilities are written to `.sol` and `.psa` files, respectively. The algorithm for the solvent contact areas is described in [Richmond & Richards, 1978], and can be tuned by changing the probe_radius and psa_integration_step parameters. The normalization for the fractional areas is carried out as described in [Hubbard & Blundell, 1987], with the normalization factors courtesy of Simon Hubbard (personal communication)[3]. The single reference is Šali & Overington, 1994. Accessibilities are calculated with scaled radii from the `$MODELS_LIB` library, as specified by Topology.submodel. The radii are scaled by EnergyData.radii_factor, which should usually be set to 1. If output also contains `ATOMIC_SOL`, atomic accessibilities in $\mathring{A}^2$ are assigned to B_{iso}, otherwise residue accessibility of type accessibility_type (from 1 to 10, for the columns in the `.psa` file) is assigned. (In either case, atomic accessibilities in Atom.accessibility are updated.) If surftyp is 1, contact accessibility is calculated; if 2, surface accessibility is returned. Atoms with undefined coordinates are assigned zero accessibility.
- `'NGH'`: Residue neighbors of each residue are listed to a `.ngh` file. The MODELLER definition of a residue–residue contact used in restraints derivation is applied [Šali & Blundell, 1993]: Any pair of residues that has any pair of atoms within neighbor_cutoff Å of each other are in contact. The number of neighbors for each residue is assigned to B_{iso}.
- `'DIH'`: All the dihedral angle types defined in the `$RESDIH_LIB` library (mainchain, sidechain, and the virtual dihedral between four successive C_α atoms, starting with the previous residue) are written to a `.dih` file. One column from this file, as selected by accessibility_type, is also assigned to B_{iso}.
- `'SSM'`: Secondary structure assignments are written to a `.ssm` file, and also to B_{iso} (0 for unknown, 1 for strand, 2 for helix, and -2 for kink). The algorithm for secondary structure assignment depends on the C_α positions only and is based on the distance matrix idea described in [Richards & Kundrot, 1988]. For each secondary structure type, a 'library' C_α distance matrix was calculated by averaging distance matrices for several secondary structure segments from a few high resolution protein structures. Program DSSP was used to assign these secondary structure segments [Kabsch & Sander, 1983]. Outlier distances were omitted from the averaging. Currently, there are only two matrices: one for the α-helix and one for the β-strand. The algorithm for secondary structure assignment is as follows:
  1. For each secondary structure type (begin with a helix, which can thus overwrite parts of strand if they overlap):
     - Define the degree of the current secondary structure fit for each C_α atom by DRMS deviation (P_1) and maximal distance difference (P_2) obtained by comparing the library distance matrix with the distance matrix for a segment starting at the given C_α position;
     - Assign the current secondary structure type to all C_α's in all segments whose DRMS deviation and maximal distance difference are less than some cutoffs ($P_1 < cut_1$, $P_1 < cut_2$) and are not already assigned to 'earlier' secondary structure types;

[3] Fractional surface area of a residue X is given as the calculated surface area divided by that of the residue in an extended tripeptide Ala-X-Ala conformation.

2. Split kinked contiguous segments of the same type into separate segments:
   Kinking residues have both DRMS and maximal distance difference beyond their respective cutoffs ($P_1 > cut_3$, $P_2 > cut_4$). The actual single kink residue separating the two new segments of the same type is the central kinking residue. Note: we are assuming that there are no multiple kinks within one contiguous segment of residues of the same secondary structure type.
3. If the current secondary structure type is β-strand: Eliminate those runs of strand residues that are not close enough to other strand residues separated by at least two other residues: P_3 is minimal distance to a non-neighboring residue of the strand type ($P_3 < cut_3$). Currently, only one pass of this elimination is done, but could be repeated until self-consistency.
4. Eliminate those segments that are shorter than the cutoff (cut_6) length (*e.g.*, 5 or 6).
5. Remove the isolated kinking residues (those that occur on their own or begin or end a segment).

- `'CRV'`: Local mainchain curvatures, multiplied by 0.1, are assigned to the B_{iso} field. Local mainchain curvature at residue i is defined as the angle (expressed in degrees, between 0 and 180) between the least-squares lines through C_α atoms $i-3$ to i and i to $i+3$.
- `'HBONDS'`: Hydrogen bonds between amino acid residues are written to a `.hbnds` file. A list of donors and acceptors in the 20 naturally occurring amino acids, specified by dnr_accpt_lib, is utilized in the computation of H-bonds. Hydrogen bonds are reported if the donor - acceptor distance is between 2.5 and 3.5Å and the donor-acceptor-acceptor antecedent angle is larger than 120 degrees. The model's B_{iso} field is not changed by this property.

Example: examples/commands/write_data.py

```
# Example for: Model.write_data()

# This will calculate solvent accessibility, dihedral angles,
# residue-residue neighbors, secondary structure, and local mainchain
# curvature for a structure in the PDB file.

from modeller import *

log.verbose()

# Get topology library for radii and the model without waters and HETATMs:
env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.io.hetatm = False
env.io.water = False

env.libs.topology.read(file='$(LIB)/top_heav.lib')
mdl = Model(env, file='1fas')

# Calculate residue solvent accessibilities, dihedral angles,
# residue neighbors, and secondary structure, and write to files:
myedat = EnergyData()
myedat.radii_factor = 1.0 # The default is 0.82 (for soft-sphere restraints)
mdl.write_data(file='1fas', edat=myedat, output='PSA DIH NGH SSM')

# Calculate local mainchain curvature
mdl.write_data(output='CRV')

# Use the calculated curvature data (in Biso)
print("The following residues have local mainchain curvature")
print("greater than 90 degrees:")
print([r for r in mdl.residues if r.atoms[0].biso * 10 > 90.0])
```

6.6.33 Model.make_region() — define a random surface patch of atoms

```
make_region(atom_accessibility=1.0, region_size=20)
```

This command defines a contiguous patch of exposed atoms of region_size. First, the exposed atoms in the model are identified by using the atom_accessibility cutoff (in $\AA^2$) (you must first assign accessibilities to every atom's B_{iso} field, either by calling **Model.write_data()** with 'PSA ATOMIC_SOL' output, or by manually assigning to Atom.biso). The seed atom is picked randomly among the exposed atoms. The patch is expanded by iteratively adding the exposed atom that is closest to the gravity center of the currently selected patch atoms. Thus, the patch is defined deterministically once the seed atom is picked. The patch is defined by setting the B_{iso} field of the model to 1 for the patch atoms and to 0 for the remaining atoms. (If you write out the model as a PDB file with **Model.write()**, this appears as the PDB temperature factor. The "temperature" color option of your PDB viewer can be used to display the patch graphically.)

To obtain surface patches that look good in visual inspection, it is necessary to use a non-obvious scaling factor for atomic radii and probe radius for solvent calculation by **Model.write_data()**, as well as the accessibility cutoff for **Model.make_region()**.

Example: examples/commands/make_region.py

```
# Example for: Model.make_region()

# This will define a random contiguous patch of atoms on a surface of the
# protein.

from modeller import *

env = Environ(rand_seed=-18343)
log.level(1, 1, 1, 1, 0)

# Read the PDB file
mdl = Model(env)
mdl.read(file='../atom_files/pdb1fdn.ent')

# Calculate atomic accessibilities (in Biso) with appropriate probe_radius
myedat = EnergyData()
myedat.radii_factor = 1.6
mdl.write_data(edat=myedat, output='PSA ATOMIC_SOL',
               psa_integration_step=0.05, probe_radius=0.1)

# Get the "random" patch of exposed atoms on the surface
mdl.make_region(atom_accessibility=0.5, region_size=35)

# Write out a PDB file with the patch indicated by Biso = 1:
mdl.write(file='1fdn.reg')

# Can also select the patch residues and use selection methods:
s = Selection([a for a in mdl.atoms if a.biso > 0.99])
print("%d atoms in surface patch" % len(s))
```

6.6.34 Model.color() — color MODEL according to alignment

```
color(aln)
```

This command 'colors' the model according to a given alignment, aln, between the model and a sequence. The model has to be the first protein in the alignment. The second protein can be any sequence, with or without known structure.

The coloring is done by setting the B_{iso} (isotropic temperature factor) field in the model as follows:

- 0, for those regions that have residues in both MODEL and the sequence (blue in RASMOL; light green in QUANTA);
- 1, for the two residues that span regions occurring in the sequence but not in MODEL (green in RASMOL; pink in QUANTA);
- 2, regions that occur in MODEL but are deleted from the sequence (red in RASMOL; bright green in QUANTA).

The model can then be written out with **Model.write()** as a PDB file, and colored using your PDB viewer based on the temperature factors. You can then inspect the model for the structural context of deletions and insertions. This is useful in optimizing the alignment for comparative modeling.

Example: examples/commands/color_aln_model.py

```
# Example for: Model.color()

# Two demos:
#
# 1) Use a given alignment to color a structure according to
#    insertions and deletions in a pairwise alignment.
#
# 2) Superpose two 3D structure and do (1).

from modeller import *
env = Environ()
env.io.atom_files_directory = ['../atom_files']

# Demo 1:
mdl = Model(env)
aln = Alignment(env)
mdl.read(file='2nbt', model_segment=('FIRST:A', 'LAST:A'))
aln.append(file='toxin.ali', align_codes=('2nbt', '1fas'), remove_gaps=True)
mdl.color(aln)
mdl.write(file='2nbt-1.clr')

# Demo 2:
aln = Alignment(env)
segs = {'2nbt':('1:A', '66:A'), '1fas':('1:A', '61:A')}
for code in ('2nbt', '1fas'):
    mdl.read(file=code, model_segment=segs[code])
    aln.append_model(mdl, align_codes=code, atom_files=code)
aln.align(gap_penalties_1d=(-600, -400))
aln.malign3d(gap_penalties_3d=(0, 3.0))
aln.write(file='color_aln_model.pap', alignment_format='PAP')

mdl.read(file='2nbt', model_segment=segs['2nbt'])
mdl.color(aln)
mdl.write(file='2nbt-2.clr')
```

6.6.35 Model.make_chains() — Fetch sequences from PDB file

```
make_chains(file, structure_types='structure', minimal_resolution=99.0, minimal_chain_length=30,
max_nonstdres=10, chop_nonstd_termini=True, minimal_stdres=30, alignment_format='PIR')
```

This command is **obsolete**. Please see **Chain.filter()** and **Chain.write()** instead.

This command fetches the sequences of the various chains found in the PDB file that has been read into memory (see **Model.read()**).

The sequence of every chain that matches the input criteria is written out to separate files. See **Chain.filter()** for a description of the input criteria, and **Chain.write()** for a description of the parameters controlling the output of the chain files.

6.6.36 Model.saxs_intens() — Calculate SAXS intensity from model

```
saxs_intens(saxsd, filename, fitflag=False)
```

Calculate SAXS intensity from model. See Section 6.29.

6.6.37 Model.saxs_pr() — Calculate $P(r)$ of model

```
saxs_pr(saxsd, filename)
```

Calculate P(r) from model. See Section 6.29.

6.6.38 Model.saxs_chifun() — Calculate SAXS score chi from model

```
saxs_chifun(transfer_is, edat=None)
```

Calculate SAXS score from model. See Section 6.29.

6.6.39 Model.assess_ga341() — assess a model with the GA341 method

```
assess_ga341()
```

Output: (score, compactness, e_native_pair, e_native_surf, e_native_comb, z_pair, z_surf, z_comb)

This command assesses the quality of the model using the GA341 method method [Melo *et al.*, 2002, John & Šali, 2003]. The method uses the percentage sequence identity between the template and the model as a parameter. MODELLER-produced PDB files contain this information in a `'REMARK'`; in the case of other PDB files, you should supply this by setting Model.seq_id.

If the model contains multiple chains, only the first is evaluated by this method; if you wish to evaluate the model in a different chain, you should write out that single chain into a new model first. (The method was parameterized for use with single-chain proteins, so its use for multi-chain models is not recommended.)

Only standard amino acids are assessed by this command. A `ModellerError`exception will be raised if the model contains no standard amino acids.

When using `AutoModel` or `LoopModel`, automatic GA341 assessment of each model can be requested by adding `assess.GA341` to AutoModel.assess_methods or LoopModel.loop.assess_methods respectively.

Example: examples/assessment/assess_ga341.py

```
# Example for: Model.assess_ga341()

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read a model previously generated by Modeller's AutoModel class
mdl = complete_pdb(env, '../atom_files/1fdx.B99990001.pdb')

# Set template-model sequence identity. (Not needed in this case, since
# this is written by Modeller into the .pdb file.)
mdl.seq_id = 37.037

score = mdl.assess_ga341()
```

6.6.40 Model.assess_normalized_dope() — assess a model with the normalized DOPE method

`assess_normalized_dope()`

Output: Z-score

This command assesses the quality of the model using the normalized DOPE method. This is a Z-score; positive scores are likely to be poor models, while scores lower than -1 or so are likely to be native-like.

The normalized DOPE score is derived from the statistics of raw DOPE scores[4]. See **Selection.assess_dope()** for more information on these raw scores.

When using `AutoModel` or `LoopModel`, automatic normalized DOPE assessment of each model can be requested by adding `assess.normalized_dope` to AutoModel.assess_methods or LoopModel.loop.assess_methods respectively.

Example: examples/assessment/assess_normalized_dope.py

```
# Example for: Model.assess_normalized_dope()

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read a model previously generated by Modeller's AutoModel class
mdl = complete_pdb(env, '../atom_files/1fdx.B99990001.pdb')

zscore = mdl.assess_normalized_dope()
```

[4]The mean and standard deviation of the DOPE score of a given protein is estimated from its sequence. The mean score of a random protein conformation is estimated by a weighted sum of protein composition over the 20 standard amino acid residue types, where each weight corresponds to the expected change in the score by inserting a specific type of amino acid residue. The weights are estimated from a separate training set of 1,686,320 models generated by MODPIPE.

6.6.41 Model.get_normalized_dope_profile() — get per-residue normalized DOPE profile

```
get_normalized_dope_profile()
```

Output: `EnergyProfile`

This returns the individual residue components of the normalized DOPE score, which can be used to detect poor regions of the model. See **Model.assess_normalized_dope()** for more details on normalized DOPE, and Section 6.25 for more details on the returned profile.

6.7 The `Restraints` class: static restraints

The `Restraints` class holds all of the static restraints which act on a model, and methods to manipulate them. It is never created manually, but can be accessed from the `Model` that the restraints act on, as Model.restraints.

6.7.1 Restraints.rigid_bodies — all rigid bodies

This is a list of all rigid bodies in the model. See section 5.3.4 for more information.

6.7.2 Restraints.pseudo_atoms — all pseudo atoms

This is a list of all pseudo atoms in the model. See section 5.3.2 for more information. Note that the list can only be appended to, and not reordered or deleted; this is to prevent restraints defined on the pseudo atoms from becoming invalidated. (You can call **Restraints.clear()** to delete all restraints, including pseudo atoms, and start again if you need to delete pseudo atoms.)

Example: See Section 5.3.2 command.

6.7.3 Restraints.excluded_pairs — all excluded pairs

This is a list of all excluded pairs in the model. See section 5.3.3 for more information.

6.7.4 Restraints.nonbonded_pairs — all nonbonded pairs

This is a list of all nonbonded pairs in the model; right now while the length of the list can be obtained, the actual pairs themselves cannot be. It is read-only; the list is populated automatically when the model's score is calculated, *e.g.* by **Selection.energy()**.

6.7.5 Restraints.symmetry — all symmetry restraints

This is a list of all symmetry restraints on the model. See section 5.3.5 for more information.

6.7.6 Restraints.symmetry.report() — report violated symmetry restraints

```
report(deviation)
```

This writes a comparison of equivalent distances involved in the definition of the symmetry enforcing term to the `log` file. All distances greater than deviation are reported. See Section 5.3.5 for an example.

6.7.7 Restraints.make() — make restraints

```
make(atmsel, restraint_type, spline_on_site, residue_span_range=(0, 99999),
residue_span_sign=True, restraint_sel_atoms=1, basis_pdf_weight='LOCAL',
basis_relative_weight=0.05, intersegment=True, dih_lib_only=False, spline_dx=0.5,
spline_min_points=5, spline_range=4.0, mnch_lib=1, accessibility_type=8, surftyp=1, distngh=6.0,
aln=None, edat=None, io=None)
```

Requirements: topology & parameters

This command calculates and selects new restraints of a specified type. See the original papers for the most detailed definition and description of the restraints [Šali & Blundell, 1993, Šali & Overington, 1994]. The calculation of restraints of all types is now (partly) limited to the atoms in the atom selection atmsel. The new restraints are added to any currently present.

The physical restraint type of the new restraints is specified by restraint_group, and should be an object from the `physical` module (see Table 6.1).

restraint_type selects the types of the generated restraints. (For restraint type `DISTANCE`, do not use this command; instead, use **Restraints.make_distance()**.) Only one restraint type can be selected at a time, except for the stereochemical restraints (`BOND`, `ANGLE`, `DIHEDRAL`, `IMPROPER`) that can all be calculated at the same time. It is useful to distinguish between the stereochemical restraints and homology-derived restraints. The stereochemical restraints are obtained from libraries that depend on atom and/or residue types only (*e.g.*, CHARMM 22 force field [MacKerell *et al.*, 1998] or statistical potentials), and do not require the alignment aln with template structures. In contrast, the homology-derived restraints are calculated from related protein structures, which correspond to all but the last sequence in the alignment aln (the target). These templates are read from coordinate files, which are the only data files required. All restraints are added to the existing restraints, even if they duplicate them (but see the comment for the `'OMEGA'` restraints below).

The atoms for non-bonded restraints also have to be within the residue spanning range specified by residue_span_range = `(r1, r2)`, such that the residue index difference $r1 \leq |ir2 - ir1| \leq r2$ when residue_span_sign = `False` and $r1 \leq (ir2 - ir1) \leq r2$ when residue_span_sign = `True`.

Stereochemical restraints:

- `'BOND'`. This calculates covalent bond restraints (harmonic terms). It relies on the list of the atom–atom bonds for MODEL, prepared previously by the **Model.generate_topology()** command. The mean values and force constants are obtained from the parameter library in memory. Only those bonds are restrained that have all or at least restraint_sel_atoms in the selection atmsel.
- `'ANGLE'`. This calculates covalent angle restraints (harmonic terms). It relies on the list of the atom–atom–atom bonds for MODEL, prepared previously by the **Model.generate_topology()** command. The mean values and force constants are obtained from the parameter library in memory. Only those angles are restrained that have all or at least restraint_sel_atoms in the selection atmsel.
- `'DIHEDRAL'`. This calculates covalent dihedral angle restraints (cosine terms). It relies on the list of the atom–atom–atom–atom dihedral angles for MODEL, prepared previously by the **Model.generate_topology()** command. The minima, phases, and force constants are obtained from the parameter library in memory. Only those dihedral angles are restrained that have all or at least restraint_sel_atoms in the selection atmsel.
- `'IMPROPER'`. This calculates improper dihedral angle restraints (harmonic terms). It relies on the list of the improper dihedral angles for MODEL, prepared previously by the **Model.generate_topology()** command. The mean values and force constants are obtained from the parameter library in memory. Only those impropers are restrained that have all or at least restraint_sel_atoms in the selection atmsel.
- `'STEREO'`. This implies all `'BOND'`, `'ANGLE'`, `'DIHEDRAL'`, and `'IMPROPER'` restraints.
- `'SPHERE14'`. This constructs soft-sphere overlap restraints (lower harmonic bounds) for atom pairs separated by exactly three bonds (1–4 pairs). It relies on atom radii from the `'$RADII14_LIB'` library. Only those non-bonded pairs are restrained that have all or at least EnergyData.nonbonded_sel_atoms in the selection atmsel. They must also satisfy the residue_span_range & residue_span_sign criterion.
- `'LJ14'`. This constructs 1–4 Lennard-Jones restraints using the modified 1–4 Lennard-Jones parameters from the CHARMM parameter library. There is no way to calculate `'LJ14'` as dynamic restraints. Only those non-bonded pairs are restrained that have all or at least EnergyData.nonbonded_sel_atoms in the selection atmsel. They must also satisfy the residue_span_range & residue_span_sign criterion.
- `'COULOMB14'`. This constructs 1–4 Coulomb restraints by relying on the atomic charges from the CHARMM topology library. There is no way to calculate `'COULOMB14'` as dynamic restraints. Only those non-bonded pairs are restrained that have all or at least EnergyData.nonbonded_sel_atoms in the selection atmsel. They must also satisfy the residue_span_range & residue_span_sign criterion.
- `'SPHERE'`. This constructs soft-sphere overlap restraints (lower harmonic bounds) for all atom pairs that are not in bonds, angles, dihedral angles, improper dihedral angles, nor are explicitly excluded by the `'E'` entries read from a restraint file or added by the **Restraints.add()** command. Only those non-bonded pairs are restrained that have all or at least EnergyData.nonbonded_sel_atoms in the selection

atmsel. They must also satisfy the residue_span_range & residue_span_sign criterion. Note that this makes these restraints static (*i.e.*, not dynamic) and that you must set EnergyData.dynamic_sphere to `False` before evaluating the molecular pdf if you want to avoid duplicated restraints. These restraints should usually not be combined with the Lennard-Jones (`'LJ'`) restraints.

When intersegment is `True`, the inter-segment non-bonded restraints are also constructed; otherwise, the segments do not feel each other *via* the non-bonded restraints. This option does not apply to the optimizers (Section 6.11) where information about segments is not used at all (*i.e.*, they behave as if intersegment = `True`).

- `'LJ'`. This constructs Lennard-Jones restraints for all atom pairs that are not in bonds, angles, dihedral angles, improper dihedral angles, nor are explicitly excluded by the `'E'` entries read from a restraint file or added by the **Restraints.add()** command. Only those non-bonded pairs are restrained that have all or at least EnergyData.nonbonded_sel_atoms in the selection atmsel. They must also satisfy the residue_span_range & residue_span_sign criterion. Note that this command makes the non-bonded restraints static (*i.e.*, not dynamic) and that you must set EnergyData.dynamic_lennard to `False` before evaluating the molecular pdf if you want to avoid duplicated restraints. Note that CHARMM uses both `'LJ14'` and `'LJ'`. For large molecules, it is better to calculate `'LJ'` as dynamic restraints because you can use distance cutoff EnergyData.contact_shell in optimization (Section 6.11) to reduce significantly the number of non-bonded atom pairs.
- `'COULOMB'`. This constructs Coulomb restraints for all atom pairs that are not in bonds, angles, dihedral angles, improper dihedral angles, nor are explicitly excluded by the `'E'` entries read from a restraint file or added by the **Restraints.add()** command. Only those non-bonded pairs are restrained that have all or at least EnergyData.nonbonded_sel_atoms in the selection atmsel. They must also satisfy the residue_span_range & residue_span_sign criterion. Note that this command makes the non-bonded restraints static (*i.e.*, not dynamic) and that you must set EnergyData.dynamic_coulomb to `False` before evaluating the molecular pdf if you want to avoid duplicated restraints. Note that CHARMM uses both `'COULOMB14'` and `'COULOMB'`. For large molecules, it is better to calculate `'COULOMB'` as dynamic restraints because you can use distance cutoff EnergyData.contact_shell in optimization (Section 6.11) to reduce significantly the number of non-bonded atom pairs.

Homology-derived restraints:

For these restraints, the input alignment aln must be given.

- `'CHI1_DIHEDRAL'`, `'CHI2_DIHEDRAL'`, `'CHI3_DIHEDRAL'`, `'CHI4_DIHEDRAL'`, `'PHI_DIHEDRAL'`, `'PSI_DIHEDRAL'`, `'OMEGA_DIHEDRAL'`, `'PHI-PSI_BINORMAL'` are the mainchain and sidechain dihedral angle restraints. Only those dihedral angles are restrained that have all or at least EnergyData.nonbonded_sel_atoms in the selection atmsel. The means and standard deviations for the dihedral Gaussian restraints are obtained from the `$RESDIH_LIB` and `$MNCH?_LIB` libraries and their weights from the MDT tables, which are read in as specified by `MDT_LIB` in `$LIB/libs.lib`. The large MDT tables give the conditional weights for each possible dihedral angle class, as a function of all possible combinations of features on which a particular class depends. If dih_lib_only is `True` or there is no equivalent residue in any of the templates, the weights for the dihedral angle classes depend only on the residue type and are obtained from the `'$RESDIH_LIB'` and `'$MNCH?_LIB'` libraries; the dih_lib_only argument allows one to force the calculation of the "homology-derived" mainchain and sidechain dihedral angle restraints that ignore template information. basis_pdf_weight has the same effect as for the distance pdf's.

  When MODELLER's `'OMEGA'` restraints are calculated, the currently existing restraints on atoms `'O C +N +CA'` in all residues are automatically deleted. These deleted restraints correspond to the *improper* dihedral angles involving the ω atoms. They are deleted because they could be "frustrated" by the new `'OMEGA'` restraints. No action is taken with regard to any of the previously existing, possibly duplicated dihedral angle restraints. Thus, to avoid restraint duplication, including that of the `'OMEGA'` restraints, call the **Restraints.unpick_redundant()** command after all the restraints are calculated.

  Dihedral restraints are only calculated for the 20 standard amino acids, plus the two alternate histidine protonation states (HSE and HSP). The statistics for HIS are applied to HSE and HSP.

The weights of basis pdf's depend on local sequence similarity between the target and the templates when basis_pdf_weight = `'LOCAL'` and on global sequence identity when basis_pdf_weight = `'GLOBAL'`.

basis_relative_weight is the cutoff for removing weak basis pdf's from poly-Gaussian feature pdf's: a basis pdf whose weight is less than the basis_relative_weight fraction of the largest weight is deleted.

If spline_on_site is True, then certain dihedral restraints are automatically replaced by splines for efficiency. See **Restraints.spline()** for a description of the spline_dx, spline_min_points, and spline_range parameters.

Several restraint types look up information from pre-calculated MDT tables, and for these the accessibility_type variable defines the type of solvent accessibility.

Example: examples/commands/make_restraints.py

```
# Example for: restraints.make(), restraints.spline(), restraints.write()

# This will compare energies of bond length restraints expressed
# by harmonic potential and by cubic spline.

from modeller import *
from modeller.scripts import complete_pdb

log.verbose()
env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

code = '1fas'
mdl = complete_pdb(env, code)
mdl.write(file=code+'.ini')

sel = Selection(mdl)
mdl.restraints.make(sel, restraint_type='bond', spline_on_site=False)
mdl.restraints.write(file=code+'-1.rsr')
edat = EnergyData(dynamic_sphere=False)
sel.energy(edat=edat)

mdl.restraints.spline(forms.Gaussian, features.Distance, physical.bond,
                      spline_range=5.0, spline_dx=0.005, edat=edat)
mdl.restraints.condense()
mdl.restraints.write(file=code+'-2.rsr')
sel.energy(edat=edat)
```

6.7.8 Restraints.make_distance() — make distance restraints

```
make_distance(atmsel1, atmsel2, aln, spline_on_site, restraint_group, maximal_distance,
residue_span_range=(0, 99999), residue_span_sign=True, distance_rsr_model=1,
basis_pdf_weight='LOCAL', basis_relative_weight=0.05, spline_dx=0.5, spline_min_points=5,
spline_range=4.0, accessibility_type=8, restraint_stdev=(0.1, 1.0), restraint_stdev2=(0.0, 0.0,
0.0), surftyp=1, distngh=6.0, edat=None, io=None, exclude_distance=0.0)
```

Requirements: topology & parameters

This command calculates and selects new distance restraints. See **Restraints.make()** for full details.

Distance restraints are generated for all pairs of atoms i, j where atom i is from selection atmsel1 and atom j is from selection atmsel2. Moreover, for a restraint to be created, at least one distance in the template structures must be less than maximal_distance (in Å). The mean of this basis pdf is equal to the template distance and its standard deviation σ is calculated from an analytic model specified by distance_rsr_model. Use model 5 for C_α–C_α distances and model 6 for N–O distances. For models 1 through 6, this standard deviation is transformed by $\sigma' = a + b * (\sigma + W_g)$ where a and b are given by restraint_stdev and W_g is a gap weighting function of the form $W_g = 0.6 * \max(0, 4 - g_{av})$. g_{av} is the average distance of the two residues involved in the restraint from a gap. For models 3 through 6, this is additionally transformed by $\sigma'' = \sigma' + \sum_i [d + e * \max(0, f - g_i)]$ where the sum is over each of the atoms i involved in the distance, d e and f are given by restraint_stdev2, and g_i is the distance of each residue from a gap. The first six models are polynomials and depend on several structural features of the template and its similarity to the target. The polynomial coefficients are specified in library file `'$PARAMS_LIB'`. When "polynomial model" 7 is selected, the standard deviation of restraints is set to constant a. Each basis pdf in the distance pdf corresponds to one template structure with an equivalent distance.

In addition, the atom pairs restrained by homology-derived restraints must by default not be in a chemical bond, chemical angle, dihedral angle, or on an excluded pairs list. This behavior can be changed by resetting EnergyData.excl_local (see **ConjugateGradients()**).

If the restrained distance for a given atom pair is less than exclude_distance, that pair is also excluded from the nonbonded list. This is useful if you are building short distance restraints to approximate bonds.

An 'equivalent' distance is defined as the distance between a pair of equivalent atoms. See **Atom.get_equivalent_atom()** for the rules for determining equivalency.

6.7.9 Restraints.unpick_all() — unselect all restraints

```
unpick_all()
```

This unselects all of the current restraints.

6.7.10 Restraints.clear() — delete all restraints

```
clear()
```

This deletes all of the current restraints, including pseudo atoms, excluded pairs, rigid bodies, and symmetry restraints.

6.7.11 Restraints.pick() — pick restraints for selected atoms

```
pick(atmsel, residue_span_range=(0, 99999), restraint_sel_atoms=1, restraints_filter=physical.Values(default
```

This command selects some or all of the restraints currently in memory.

The selection is added to any existing selected restraints; if instead you want to select only these restraints, call **Restraints.unpick_all()** first.

This command runs over all restraints in memory, including the currently unselected restraints. Be careful about this: If you have some unselected restraints in memory, **Restraints.pick()** may select them; if you wish to prevent this, do **Restraints.remove_unpicked()** before calling **Restraints.pick()**.

A static restraint is selected if all or at least restraint_sel_atoms of its atoms are within the atom selection atmsel, if it is strong enough based on its standard deviations or force constants (see the next paragraph), and if it does not span fewer residues than residue_span_range[0], or more than residue_span_range[1]. (Restraints which act on only a single atom are not subject to this range check.) Note that here restraint_sel_atoms is used for *all* restraints, while the **Restraints.make()** command and optimizers (Section 6.11) use it for all restraint types *except* non-bonded pairs. (EnergyData.nonbonded_sel_atoms is used for non-bonded pairs by these routines.)

To decide if a restraint is strong enough, the current standard deviations or force constants are compared with the corresponding restraints_filter[physical_restraint_type]. A harmonic restraint, lower and upper bounds, and multi-modal Gaussian restraints are selected if the (smallest) standard deviation is less than the corresponding restraints_filter[i]. The cosine energy term is selected if its force constant is larger than the corresponding restraints_filter[i]. Restraints with other mathematical forms (including user-defined forms) are always selected, as is any restraint of physical type i if restraints_filter[i] $= -999$. The restraints_filter angles have to be specified in radians.

Example: examples/commands/pick_restraints.py

```
# Example for: restraints.pick(), restraints.condense()

# This will pick only restraints that include at least one
# mainchain (CA, N, C, O) atom and write them to a file.

from modeller import *
from modeller.scripts import complete_pdb

log.verbose()
env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

mdl = complete_pdb(env, '1fas')

allsel = Selection(mdl)
mdl.restraints.make(allsel, restraint_type='stereo', spline_on_site=False)
allsel.energy()

atmsel = allsel.only_atom_types('CA N C O')
mdl.restraints.pick(atmsel, restraint_sel_atoms=1)
# Delete the unselected restraints from memory:
mdl.restraints.condense()
atmsel.energy()

mdl.restraints.write(file='1fas.rsr')
```

6.7.12 Restraints.unpick_redundant() — unselect redundant restraints

`unpick_redundant()`

This unselects those cosine dihedral angle restraints (restraint_type = 'DIHEDRAL') that operate on the same atoms as any other restraints on a dihedral angle or a pair of dihedral angles. Such restraints include the MODELLER 'PHI_DIHEDRAL', 'PSI_DIHEDRAL', 'OMEGA_DIHEDRAL', 'CHI1_DIHEDRAL', 'CHI2_DIHEDRAL', 'CHI3_DIHEDRAL', 'CHI4_DIHEDRAL', 'PHI_PSI_CLASS', 'MRFP_DIHEDRAL', and 'PHI_PSI_BINORMAL' dihedral angle restraints, as well as the 2nd, 3rd, *etc.* cosine dihedral angle restraints on the same atoms; the improper dihedral angle restraints are not considered here. For this command to work properly, the cosine dihedral angle restraints must be constructed before any other dihedral angle restraints. This functionality is needed because some of the CHARMM cosine terms are sometimes duplicated by other CHARMM cosine terms as well as by MODELLER homology-derived mainchain and sidechain dihedral and bi-dihedral angle restraints. When using the standard AutoModel class, the redundant CHARMM terms are always removed. See also **Restraints.condense()**.

6.7.13 Restraints.remove_unpicked() — remove unselected restraints

```
remove_unpicked()
```

This command permanently removes all the unselected restraints from memory. See also **Restraints.condense()**.

6.7.14 Restraints.condense() — remove unselected or redundant restraints

```
condense()
```

This command permanently removes all the unselected or redundant restraints from memory. This is exactly the same as calling **Restraints.unpick_redundant()** followed by **Restraints.remove_unpicked()**.

Example: See **Model.read()** command.

6.7.15 Restraints.add() — add restraint

```
add(*args)
```

This command adds one or more restraints to the end of the restraints list and selects them. It should be given one or more arguments, which are the restraints to add. These are mathematical form objects, as described in Section 5.3.1, or secondary structure objects, as described in Section 6.8.

This command is also useful for specifying *cis*-peptide bonds from your own scripts, using the **cispeptide()** command.

Example: examples/commands/add_restraint.py

```
# Example for: restraints.add(), restraints.unpick()

# This will enforce cis conformation for Pro-56.

# Make a model and stereochemical restraints:

from modeller import *
from modeller.scripts import complete_pdb, cispeptide

log.level(output=1, notes=1, warnings=1, errors=1, memory=0)
env = Environ()
env.io.atom_files_directory = ['../atom_files']

env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

code = '1fas'
mdl = complete_pdb(env, code)
rsr = mdl.restraints
atmsel = Selection(mdl)
rsr.make(atmsel, restraint_type='stereo', spline_on_site=False)

# Change the Pro-56 restraint from trans to cis:
a = mdl.chains[0].atoms
cispeptide(rsr, atom_ids1=(a['O:56'], a['C:56'], a['N:57'], a['CA:57']),
                  atom_ids2=(a['CA:56'], a['C:56'], a['N:57'], a['CA:57']))

# Constrain the distance between alpha carbons in residues 5 and 15 to
# be less than 10 angstroms:
rsr.add(forms.UpperBound(group=physical.xy_distance,
                         feature=features.Distance(a['CA:5'], a['CA:15']),
                         mean=10., stdev=0.1))

rsr.write(file='1fas.rsr')
atmsel.energy()
```

6.7.16 Restraints.unpick() — unselect restraints

unpick(*atom_ids)

This command scans the currently selected restraints to find all the restraints that operate on the specified atoms (Section 5.3.1) and then unselects them. The order of the atoms in atom_ids does not matter: all restraints that contain all and only the specified atoms are unselected. This means that it is not possible to distinguish between the dihedral angle and improper dihedral angle restraints on the same four atoms.

The command only unselects the restraints found. To completely remove all the unselected restraints from memory, use **Restraints.condense()**. The **Restraints.unpick()** command is useful in specifying *cis*-peptide bonds in your own scripts; see **cispeptide()**.

Example: See **Restraints.add()** command.

6.7.17 Restraints.reindex() — renumber model restraints using another model

reindex(mdl)

Requirements: restraints

This command renumbers atom indices in all restraints in memory. It is expected that the input restraints refer to atoms in the passed model mdl; the re-indexed restraints will correspond to the current model. Only those restraints that have all atoms in the current model will be selected. You can remove the others by **Restraints.condense()**. This command is useful when the old restraints have to be used while changing from one topology model to another.

Example: examples/commands/reindex_restraints.py

```
# Example for: restraints.reindex()

# This will reindex restraints obtained previously for a simpler topology so
# that they will now apply to a more complicated topology.

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.io.atom_files_directory = ['../atom_files']
tpl = env.libs.topology
par = env.libs.parameters
```

```
# Generate the model for the simpler topology (CA only in this case):
tpl.read(file='$(LIB)/top_ca.lib')
par.read(file='$(LIB)/par_ca.lib')

code = '1fas'
mdl = complete_pdb(env, code)
mdl.write(file=code+'.ca')

# Generate the restraints for the simpler topology:
sel = Selection(mdl)
mdl.restraints.make(sel, restraint_type='stereo', spline_on_site=False)
mdl.restraints.write(file='1fas-ca.rsr')
sel.energy()

# Generate the model for the more complicated topology:
tpl.read(file='$(LIB)/top_heav.lib')
par.read(file='$(LIB)/par.lib')

mdl.read(file=code)
aln = Alignment(env)
aln.append_model(mdl, atom_files=code, align_codes=code)
aln.append_model(mdl, atom_files=code+'.ini', align_codes=code+'-ini')
mdl.clear_topology()
mdl.generate_topology(aln[code+'-ini'])
mdl.transfer_xyz(aln)
mdl.write(file='1fas.ini')

mdl2 = Model(env, file='1fas.ca')
mdl.restraints.reindex(mdl2)
mdl.restraints.write(file='1fas.rsr')
sel = Selection(mdl)
sel.energy()
```

6.7.18 Restraints.spline() — approximate restraints by splines

```
spline(form, feature, group, spline_dx=0.5, spline_range=4.0, spline_min_points=5, output='',
edat=None)
```

This command calculates and selects new restraints that are a spline approximation of the selected restraints of the specified type. The type is specified by form (see Section 5.3.1), feature (Section 5.3.1) and group (Table 6.1). It unselects the approximated restraints.

The restraint is approximated in a certain range only, determined differently for different mathematical forms:

- Any form acting on an angle feature will be splined from $-\pi$ to π.
- `forms.Gaussian` will be splined from $m - \text{spline\_range} \times \sigma$ to $m + \text{spline\_range} \times \sigma$, where m is the mean and σ the standard deviation.
- `forms.MultiGaussian` will be splined from $m - \text{spline\_range} \times \sigma_m$ to $M + \text{spline\_range} \times \sigma_M$, where m and M are the minimal and maximal means of the basis pdfs, and σ_m and σ_M are their corresponding standard deviations.

- `forms.Spline` will be splined using the existing range of the spline.
- For user-defined forms (see Section 7.1.2) the range is defined by their get_range function.
- Forms that act on multiple features, such as `forms.MultiBinormal` or suitable user-defined forms, will result in a `ValueError`.
- All other forms cannot be converted to splines, and will result in a `NotImplementedError`.

The spline points are distributed evenly over this range with an interval of spline_dx. spline_dx should be equal to the scale of the peaks of the restraint that you want to approximate reliably. The value of the restraint beyond the range is determined by linear extrapolation using the first derivatives at the bounds.

If the x-range and spline_dx are such that the number of spline points would be less than spline_min_points, the high end of the range is increased so that there are spline_min_points defining the "splined" restraint.

If output is set to `'SPLINE'`, then tables are also written out comparing each pair of original and splined restraints.

Example: See **Restraints.make()** command.

6.7.19 Restraints.append() — read spatial restraints

```
append(file)
```

This command reads restraints, excluded atom pairs, and pseudo atom definitions from a file. An excluded atom pair specifies two atoms that are not to be tested during generation of the dynamic non-bonded pair list. There is one restraint entry per line. The new restraints are added to those that are already in memory; if you want to replace them, call **Restraints.clear()** first. All the new restraints are automatically selected.

file can be a filename or a readable file handle (see **modfile.File()**).

Example: See **Restraints.make()** command.

6.7.20 Restraints.write() — write spatial restraints

```
write(file)
```

This command writes the currently selected restraints to a file. These can be read with the **Restraints.append()** command.

file can be a filename or a writeable file handle (see **modfile.File()**).

Example: See **Restraints.make()** command.

6.8 The `secondary_structure` module: secondary structure restraints

The `secondary_structure` module provides classes to restrain secondary structure. Note that all of these restraints are simply added to the list of all restraints, and MODELLER will attempt to satisfy them as best it can, but their presence does not guarantee that the requested secondary structure will be adopted.

6.8.1 Alpha() — make an α-helix

`Alpha(residues)`

This makes restraints enforcing an α-helix (mainchain conformation class "A") for the residue segment specified by residues (which can be created using the **Model.residue_range()** function). The helix is restrained by Φ, Ψ binormal restraints, N–O hydrogen bonds, C_α–C_α distances for $i - j \in \{2 - 9\}$, C_α–O distances for $i - j \in \{2 - 9\}$, and O–O distances for $i - j \in \{2 - 6\}$ [5]. Note that this requires all heavy atoms to be present to work properly, so will not work with the C_α-only topology.

In many cases (*e.g.*, most comparative modeling runs) you will already have binormal, C_α–C_α, and N-O restraints active (which will conflict with helix restraints), so it is recommended that you first use **Restraints.unpick()** followed by **Restraints.condense()** to remove these.

To actually add the restraints, pass the new object to **Restraints.add()**.

Example: examples/commands/secondary_structure.py

```
# Example for Model.build_sequence(), secondary_structure.Alpha()

from modeller import *
from modeller.optimizers import ConjugateGradients

# Set up environment
e = Environ()
e.libs.topology.read('${LIB}/top_heav.lib')
e.libs.parameters.read('${LIB}/par.lib')

# Build an extended chain model from primary sequence, and write it out
m = Model(e)
m.build_sequence('GSCASVCGV')
m.write(file='extended-chain.pdb')

# Make stereochemical restraints on all atoms
allatoms = Selection(m)
m.restraints.make(allatoms, restraint_type='STEREO', spline_on_site=False)

# Constrain all residues to be alpha-helical
# (Could also use m.residue_range() rather than m.residues here.)
m.restraints.add(secondary_structure.Alpha(m.residues))

# Get an optimized structure with CG, and write it out
cg = ConjugateGradients()
cg.optimize(allatoms, max_iterations=100)
m.write(file='alpha-helix.pdb')
```

[5] The target distances were all obtained from a regular α-helix in one of the high-resolution myoglobin structures.

6.8.2 Strand() — make a β-strand

```
Strand(residues)
```

This makes restraints enforcing an extended β-strand conformation for the residue segment specified by residues (which can be created using the **Model.residue_range()** function). This is achieved by applying Φ, Ψ binormal restraints only. These binormal restraints force the mainchain conformation into class "B", except for the Pro residues which are restrained to class "P" [Šali & Blundell, 1993].

All of the restraints have the `physical.phi_psi_dihedral` physical restraint type, so can be strengthened or weakened by creating a **physical.Values()** object (see also Section 2.2.2).

In many cases (*e.g.*, most comparative modeling runs) you will already have binormal restraints active (which will conflict with strand restraints), so it is recommended that you first use **Restraints.unpick()** followed by **Restraints.condense()** to remove these.

To actually add the restraints, pass the new object to **Restraints.add()**. See Section 2.2.11 for an example.

6.8.3 Sheet() — make a β-sheet

```
Sheet(atom1, atom2, sheet_h_bonds)
```

This calculates H-bonding restraints for a pair of β-strands. atom1 and atom2 specify the first H-bond in the β-sheet ladder. sheet_h_bonds specifies the number of H-bonds to be added — positive for a parallel sheet, and negative for an anti-parallel sheet. In a parallel sheet, hydrogen bonds start at the first or the second term of the following series (depending on atom1 and atom2): 1N:1O, 1O:3N, 3N:3O, 3O:5N, *etc.* For an anti-parallel sheet, the corresponding series is 1N:3O, 1O:3N, 3N:1O, 3O:1N, *etc.* (note that the residue indices run in decreasing order for the second strand in this case). The extended structure of the individual strands themselves is not enforced; use separate **Strand()** restraints if so desired.

All of the restraints have the `physical.h_bond` physical restraint type, so can be strengthened or weakened by creating a **physical.Values()** object (see also Section 2.2.2).

To actually add the restraints, pass the new object to **Restraints.add()**. See Section 2.2.11 for an example.

6.9 The `Selection` class: handling of sets of atom coordinates

The `Selection` class holds a set of atoms from a model. Such selections can be used to perform actions on only some model atoms.

Selections in MODELLER behave almost identically to standard Python sets - see the section in the Python manual on sets for further information. Selections are 'sets' in the mathematical sense, and as such can be combined with each other in unions (using the Python `union` function, or the `|` operator) or intersections (using the `intersection` function, or the `&` operator), *etc.*

All of the atoms in a selection must belong to the same `Model` object. Note that the selection is not a copy of the model atoms; if model atoms are changed, the atoms will also be moved in any selection that contains those atoms, and vice versa.

To add objects to a selection, you can list them when you create the selection with the **Selection()** constructor, add them to an existing selection with **Selection.add()**, or combine selections with set operations (see above).

See Model.atoms, Sequence.residues, Sequence.chains, **Model.atom_range()**, **Model.residue_range()**, **Model.get_insertions()**, **Model.get_deletions()**, and **Model.loops()** for valid objects (groups of atoms) to add to the selection. You can also add `Model` objects and existing `Selection` objects to a selection. (Adding a residue adds all atoms in that residue to the selection, adding a model adds all atoms in the model, and so on.)

See also **Point.select_sphere()**, which creates a new `Selection` object.

Once you have a selection, it can be manipulated by standard Python set operations (above), by methods to add new atoms (**Selection.by_residue()**, **Selection.select_sphere()**), or by methods to exclude atom or residue types (**Selection.only_sidechain()**, **Selection.only_mainchain()**, **Selection.only_atom_types()**, **Selection.only_residue_types()**, **Selection.only_std_residues()**, **Selection.only_no_topology()**, **Selection.only_het_residues()**, **Selection.only_water_residues()**, **Selection.only_defined()**).

6.9.1 Selection() — create a new selection

```
Selection(*atoms)
```

This creates a new empty `Selection` object. An initial group of atoms or other objects can be added to the selection by listing them here; see Section 6.9 for more information.

Example: examples/python/selection.py

```
from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

mdl = Model(env, file='1fdn')

# New empty selection
s = Selection()

# Add all atoms from residues 4 through 10 (chain A) inclusive (PDB numbering)
s.add(mdl.residue_range('4:A', '10:A'))

# Selection of all atoms currently within 5A of atom CA in residue 1 in chain A
# (this destroys the previous selection):
s = mdl.atoms['CA:1:A'].select_sphere(5)

# Is the CB:1:A atom in the selection?
```

```
print(mdl.atoms['CB:1:A'] in s)

# Alternative ways of selecting the same atom:
print(mdl.chains['A'].residues['1'].atoms['CB'] in s)
print(mdl.residues['1:A'].atoms['CB'] in s)

# All atoms currently within 5A of atom CA:1:A, OR currently within 3A of the
# point (1,10,1):
s = mdl.atoms['CA:1:A'].select_sphere(5) | mdl.point(1,10,1).select_sphere(3)

# All atoms currently within 5A of atom CA:1:A, AND also currently within 3A
# of the point (1,10,1):
s = mdl.atoms['CA:1:A'].select_sphere(5) & mdl.point(1,10,1).select_sphere(3)

# All atoms currently within 5A of atom CA:1:A, OR currently within 3A of the
# point (1,10,1), but not BOTH:
s = mdl.atoms['CA:1:A'].select_sphere(5) ^ mdl.point(1,10,1).select_sphere(3)

# Create a selection containing the CA atom from residue 1, chain A,
# and all of residue 2 (PDB numbering)
s = Selection(mdl.atoms['CA:1:A'], mdl.residues['2:A'])

# All residues EXCEPT 5-10 in chain A (i.e. all atom selection minus the
# selection of residues 5-10, otherwise known as an inverted selection):
s = Selection(mdl) - Selection(mdl.residue_range('5:A', '10:A'))

# All atoms in any residue that contains a CG atom
s = Selection(mdl).only_atom_types('CG').by_residue()

# The same as above, plus all atoms in residues immediately neighboring
# these residues (by sequence)
s = Selection(mdl).only_atom_types('CG').extend_by_residue(1)

# Selection of residues 1, 4, 8 and 10-15 (PDB numbering) from chain A:
s = Selection(mdl.residues['1:A'], mdl.residues['4:A'], mdl.residues['8:A'],
              mdl.residue_range('10:A', '15:A'))

# Print the center of mass (note: not mass weighted)
print(s.mass_center)

# Rotate by 90 degrees about the z axis through the origin (0,0,0)
# (right handed rotation)
s.rotate_origin([0,0,1], 90)

# The same thing, except that the axis passes through the center of mass:
s.rotate_mass_center([0,0,1], 90)

# Translate by 5 angstroms along the x axis
s.translate([5.0, 0, 0])

# Equivalent (but less efficient, as it involves calculating the COM)
s.x += 5.0
```

Example: examples/commands/pick_atoms.py

```
# This will pick various subsets of atoms in the MODEL and compare them
# with MODEL2.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']
log.level(1, 1, 1, 1, 0)

# Read the models and the alignment:
mdl  = Model(env, file='1fas')
mdl2 = Model(env, file='2ctx')
aln = Alignment(env, file='toxin.ali', align_codes=('1fas', '2ctx'))
aln.write(file='toxin.pap', alignment_format='PAP')

# Pick and superpose mainchain atoms:
atmsel = Selection(mdl).only_mainchain()
atmsel.superpose(mdl2, aln)

# Pick and superpose sidechain atoms:
atmsel = Selection(mdl).only_sidechain()
atmsel.superpose(mdl2, aln)

# Pick and superpose CA and CB atoms:
atmsel = Selection(mdl).only_atom_types('CA CB')
atmsel.superpose(mdl2, aln)

# Pick all atoms in residues with rings:
atmsel = Selection(mdl).only_residue_types('TYR PHE TRP HIS')

# Pick and superpose all atoms:
atmsel = Selection(mdl)
atmsel.superpose(mdl2, aln)

# Pick and superpose CA and CB atoms in one segment only:
atmsel = Selection(mdl.residue_range('2:A', '10:A')).only_atom_types('CA CB')
atmsel.superpose(mdl2, aln)

# Pick and superpose all atoms within 6 angstroms of the 'CA' atom in
# residue '10' in chain A:
atmsel = mdl.atoms['CA:10:A'].select_sphere(6.0)
atmsel.superpose(mdl2, aln)

# Pick and superpose all atoms within 6 angstroms of any atom in
# segment 2:A to 10:A
atmsel = Selection(mdl.residue_range('2:A', '10:A')).select_sphere(6.0)
atmsel.superpose(mdl2, aln)

# Pick all atoms in the model
atmsel = Selection(mdl)

# Pick all atoms in all loops (ie residues within 2 positions
# of any gap in the alignment):
loops = mdl2.loops(aln, minlength=5, maxlength=15, insertion_ext=2,
                   deletion_ext=2)
atmsel = Selection(loops)
```

```
# Pick all atoms within 6 angstroms of all loops
atmsel = Selection(loops).select_sphere(6.0)
```

6.9.2 Selection.add() — add objects to selection

`add(obj)`

This adds the given objects (which can be atoms, residues, atom lists, *etc*) to the selection. `obj` can also be a Python list or tuple, in which case every object in the list is added.

6.9.3 Selection.extend_by_residue() — extend selection by residue

`extend_by_residue(extension)`

This returns a new selection, in which any residues in the existing selection that have at least one selected atom are now entirely selected. Additionally, `extension` residues around each selected residue (by sequence) are also selected. The original selection is unchanged.

Example: See **Selection()** command.

6.9.4 Selection.by_residue() — make sure all residues are fully selected

`by_residue()`

This returns a new selection, in which any residues in the existing selection that have at least one selected atom are now entirely selected (all atoms in each residue are selected). The original selection is unchanged. This is equivalent to calling **Selection.extend_by_residue()** with an extension of zero.

Example: See **Selection()** command.

6.9.5 Selection.select_sphere() — select all atoms within radius

`select_sphere(radius)`

This returns a new selection containing all atoms within the given distance from any atom in the current selection (note that this uses only the *current* coordinates — if you move the atoms later, *e.g.* during an optimization, the set of atoms does not change). Compare with **Point.select_sphere()**.

Example: See **Selection()** command.

6.9.6 Selection.only_mainchain() — select only mainchain atoms

`only_mainchain()`

This returns a new selection containing only mainchain atoms (*i.e.*, atom types O, OT1, OT2, OXT, C, CA, N) from the current selection.

6.9.7 Selection.only_sidechain() — select only sidechain atoms

`only_sidechain()`

This returns a new selection containing only sidechain atoms from the current selection. It is the opposite of **Selection.only_mainchain()**.

6.9.8 Selection.only_atom_types() — select only atoms of given types

`only_atom_types(atom_types)`

This returns a new selection containing only atoms from the current selection of the given space-separated type(s).

Example: See **Selection()** command.

6.9.9 Selection.only_residue_types() — select only atoms of given residue type

`only_residue_types(residue_types)`

This returns a new selection containing only atoms from the current selection in residues of the given space-separated type(s). The type is the CHARMM name, as defined in `'modlib/restyp.lib'`.

Example: See **Selection()** command.

6.9.10 Selection.only_std_residues() — select only standard residues

`only_std_residues()`

This returns a new selection containing only atoms from the current selection in standard residue types (*i.e.*, everything but BLK and HETATM).

6.9.11 Selection.only_no_topology() — select only residues without topology

`only_no_topology()`

This returns a new selection containing only atoms from the current selection in residues that have no defined topology. This generally includes BLK and unknown residue types, and is used by the `AutoModel` class to generate rigid body restraints.

6.9.12 Selection.only_het_residues() — select only HETATM residues

`only_het_residues()`

This returns a new selection containing only atoms from the current selection in HETATM residues.

6.9.13 Selection.only_water_residues() — select only water residues

```
only_water_residues()
```

This returns a new selection containing only atoms from the current selection in water residues.

6.9.14 Selection.only_defined() — select only atoms with defined coordinates

```
only_defined()
```

This returns a new selection containing only atoms from the current selection that have defined coordinates (see **Model.build()** and **Model.transfer_xyz()**).

6.9.15 Selection.write() — write selection coordinates to a file

```
write(file, model_format='PDB', no_ter=False, extra_data='')
```

This command writes the coordinates of all atoms in the selection to a file in the selected format.

See **Model.write()** for full details; note that only `'PDB'` and `'MMCIF'` outputs are supported with this command.

6.9.16 Selection.translate() — translate all coordinates

```
translate(vector)
```

This translates the coordinates of all atoms in the selection by the given vector. All distances are in angstroms.

Example: See **Selection()** command.

6.9.17 Selection.rotate_origin() — rotate coordinates about origin

```
rotate_origin(axis, angle)
```

This does a right-handed rotation of the coordinates of all atoms in the selection about the given axis through the origin, by the given angle (in degrees). See also **Selection.transform()** and **Selection.rotate_mass_center()**.

Example: See **Selection()** command.

6.9.18 Selection.rotate_mass_center() — rotate coordinates about mass center

```
rotate_mass_center(axis, angle)
```

This does a right-handed rotation of the coordinates of all atoms in the selection about the given axis through the mass center, by the given angle (in degrees). See also **Selection.transform()** and **Selection.rotate_origin()**.

Example: See **Selection()** command.

6.9.19 Selection.transform() — transform coordinates with a matrix

`transform(matrix)`

This transforms all the selection's coordinates using the given 3x3 matrix. This can be used to perform rotations, translations, shears, *etc.*

Example: examples/commands/rotate_model.py

```
# Example for: Selection.transform(), Selection.translate(),
# Selection.rotate_origin()

# This will orient a model as specified:

from modeller import *

# Read the structure:
env = Environ()
env.io.atom_files_directory = ['../atom_files']
mdl = Model(env, file='1fas')
# Select all atoms
s = Selection(mdl)

# Translate 1 angstrom along the x axis:
s.translate([1, 0, 0])

# Transform with a rotation matrix (no change in this example):
s.transform([[1, 0, 0],
             [0, 1, 0],
             [0, 0, 1]])

# Rotate 90 degrees about the axis, through the origin:
s.rotate_origin([1, 1, 1], 90)

mdl.write(file='1fas.ini')
```

6.9.20 Selection.mutate() — mutate selected residues

`mutate(residue_type)`

This command mutates the selected residues to the type specified by residue_type. CHARMM 4-character residue type names are used (see library file `$RESTYP_LIB`). All of the residues with at least one selected atom are mutated. To produce mutants, employ this command with **Alignment.append_model()** and **Alignment.write()**. It is usually necessary to write the mutated sequence out and read it in before proceeding, because not all sequence related information about the model is changed by this command (*e.g.*, internal coordinates, charges, and atom types and radii are not updated).

Example: examples/commands/mutate_selection.py

```
# Example for: Selection.mutate()

# This will read a PDB file, change its sequence a little, build new
# coordinates for any of the additional atoms using only the internal
# geometry, and write the mutant PDB file.  It can be seen as primitive
# but rapid comparative modeling for substitution mutants. For more
# sophisticated modeling, see https://salilab.org/modeller/wiki/Mutate%20model
#
# For insertion and deletion mutants, follow the standard comparative
# modeling procedure.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']

# Read the topology library with non-hydrogen atoms only:
env.libs.topology.read(file='$(LIB)/top_heav.lib')
# To produce a mutant with all hydrogens, uncomment this line:
#env.libs.topology.read(file='$(LIB)/top_allh.lib')

# Read the CHARMM parameter library:
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read the original PDB file and copy its sequence to the alignment array:
code = '1fas'
aln = Alignment(env)
mdl = Model(env, file=code)
aln.append_model(mdl, atom_files=code, align_codes=code)

# Select the residues to be mutated: in this case all ASP residues:
sel = Selection(mdl).only_residue_types('ASP')

# The second example is commented out; it selects residues '1' and '10'.
#sel = Selection(mdl.residues['1'], mdl.residues['10'])

# Mutate the selected residues into HIS residues (neutral HIS):
sel.mutate(residue_type='HIS')

# Add the mutated sequence to the alignment arrays (it is now the second
# sequence in the alignment):
aln.append_model(mdl, align_codes='1fas-1')

# Generate molecular topology for the mutant:
mdl.clear_topology()
mdl.generate_topology(aln['1fas-1'])

# Transfer all the coordinates you can from the template native structure
# to the mutant (this works even if the order of atoms in the native PDB
# file is not standard):
mdl.transfer_xyz(aln)

# Build the remaining unknown coordinates for the mutant:
mdl.build(initialize_xyz=False, build_method='INTERNAL_COORDINATES')

# Write the mutant to a file:
mdl.write(file='1fas-1.atm')
```

6.9.21 Selection.randomize_xyz() — randomize selected coordinates

`randomize_xyz(deviation)`

This command randomizes the Cartesian coordinates of the selected atoms.

If deviation is positive, the coordinates are randomized by the *addition* of a random number uniformly distributed in the interval from −deviation to +deviation angstroms. Atoms in residues containing rings (TYR, PHE, TRP, HIS) are capped to move no more than 0.5 angstroms regardless of the setting of deviation, so that the ring structure is roughly maintained and the system does not later get stuck in local minima.

If deviation is negative, the coordinates are *assigned* a random value uniformly distributed in the interval from −deviation to +deviation angstroms.

Any defined rigid bodies (see Section 5.3.4) remain rigid; only their mass centers are modified by this command (no rotation is done).

Example: examples/commands/randomize_xyz.py

```
# Example for: Selection.randomize_xyz()

# This will randomize the X,Y,Z of the model:

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']

mdl = Model(env, file='1fas')

# Act on all atoms in the model
sel = Selection(mdl)

# Change all existing X,Y,Z for +- 4 angstroms:
sel.randomize_xyz(deviation=4.0)
mdl.write(file='1fas.ini1')

# Assign X,Y,Z in the range from -100 to 100 angstroms:
sel.randomize_xyz(deviation=-100.0)
mdl.write(file='1fas.ini2')
```

6.9.22 Selection.superpose() — superpose model on selection given alignment

`superpose(mdl2, aln, fit=True, superpose_refine=False, rms_cutoff=3.5, reference_atom='', reference_distance=3.5, refine_local=True, swap_atoms_in_res='')`

Output: SuperposeData object

This command superposes mdl2 on the selection, without changing the alignment, aln.

The selection model must be the first sequence in the alignment; mdl2 must be the second sequence in the alignment. The equivalent atoms are the selected atoms that have equivalently named atoms in mdl2; the atom equivalences are defined in library `$ATMEQV_LIB`.

No fitting is done if fit = `False`.

rms_cutoff is the cutoff used in calculating the cutoff Rms deviations; *i.e.*, those position and distance Rms deviations that are defined on the equivalent atoms which are less than rms_cutoff angstroms away from each other (as superposed using all aligned positions) and those equivalent distances which are less than rms_cutoff angstroms different from each other, respectively.

If refine_local is `True` the superposition is then refined by considering local similarity. The DRMS profile of the two structures is calculated over a moving window of 11 residues, and a simple heuristic is then used to detect boundaries between local structural fragments. Then each of these fragments is used as the basis for least-squares fitting. The final returned orientation is that which results in the maximum number of equivalent positions, if any is better than the original superposition. (Note that this may result in a higher RMS.)

If superpose_refine is `True` the refinement of the superposition is done by repeating the fitting with only those aligned pairs of atoms that are within rms_cutoff of each other until there is no change in the number of equivalent positions. This refinement can only remove compared positions, not add them like **Alignment.align3d()** can do. This is useful for comparing equivalent parts of two structures with a fixed alignment but omitting divergent parts from the superposition and Rms deviation calculation; *e.g.*, comparing a model with the X-ray structure.

If superpose_refine is `False` and reference_atom is non-blank, only those pairs of equivalently named selected atoms from aligned residues are superposed that come from residues whose reference_atom atoms are closer than reference_distance Å to each other.

When the selection model and mdl2 have exactly the same atoms in the same order, one can set swap_atoms_in_res to any combination of single character amino acid residue codes in `DEFHLNQRVY`. Certain atoms (see below) in the specified sidechains of mdl2 are then swapped to minimize their Rms deviation relative to the selection model. The labeling resulting in the lowest Rms deviation is retained. The following swaps are attempted:

Residue	Swap(s)
D	OD1, OD2
E	OE1, OE2
F	CD1, CD2
	CE1, CE2
H	ND1, CD2
	NE2, CE1
N	OD1, ND2
Q	OE1, NE2
R	NH1, NH2
V	CG1, CG2
Y	CD1, CD2
	CE1, CE2

On successful completion, a SuperposeData object is returned, which contains all of the calculated data. For instance, if you save this in a variable 'r', the following data are available:

- `r.initial_rms`; the RMS before superposition
- `r.rms`; the RMS after superposition
- `r.drms`; the distance RMS after superposition
- `r.cutoff_rms`; the RMS after superposition of atoms within rms_cutoff
- `r.cutoff_drms`; the DRMS after superposition of distances within rms_cutoff
- `r.rotation`; the rotation matrix that was used to transform mdl2 (applied first), if fit is `True`
- `r.translation`; the translation that was used to transform mdl2 (applied after rotation), if fit is `True`
- `r.num_equiv_pos`; the number of equivalent positions
- `r.num_equiv_dist`; the number of equivalent distances
- `r.num_equiv_cutoff_pos`; the number of equivalent positions within rms_cutoff

- r.num_equiv_cutoff_dist; the number of equivalent distances within rms_cutoff

Example: examples/commands/superpose.py

```
# Example for: Selection.superpose()

# This will use a given alignment to superpose Calpha atoms of
# one structure (2ctx) on the other (1fas).

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']

mdl  = Model(env, file='1fas')
mdl2 = Model(env, file='2ctx')
aln = Alignment(env, file='toxin.ali', align_codes=('1fas', '2ctx'))

atmsel = Selection(mdl).only_atom_types('CA')
r = atmsel.superpose(mdl2, aln)

# We can now use the calculated RMS, DRMS, etc. from the returned 'r' object:
rms = r.rms
drms = r.drms
print("%d equivalent positions" % r.num_equiv_pos)

mdl2.write(file='2ctx.fit')
```

Example: examples/commands/align3d.py

```
# Example for: Alignment.align3d(), Selection.superpose()

# This will align 3D structures of two proteins:

from modeller import *
log.verbose()
env = Environ()
env.io.atom_files_directory = ['../atom_files']

# First example: read sequences from a sequence file:
aln = Alignment(env)
aln.append(file='toxin.ali', align_codes=['1fas', '2ctx'])
aln.align(gap_penalties_1d=[-600, -400])
aln.align3d(gap_penalties_3d=[0, 4.0])
aln.write(file='toxin-str.ali')

# Second example: read sequences from PDB files to eliminate the
# need for the toxin.ali sequence file:
mdl = Model(env)
aln = Alignment(env)
for code in ['1fas', '2ctx']:
    mdl.read(file=code)
    aln.append_model(mdl, align_codes=code, atom_files=code)
aln.align(gap_penalties_1d=(-600, -400))
aln.align3d(gap_penalties_3d=(0, 2.0))
aln.write(file='toxin-str.ali')

# And now superpose the two structures using current alignment to get
# various RMS's:
mdl = Model(env, file='1fas')
atmsel = Selection(mdl).only_atom_types('CA')
mdl2 = Model(env, file='2ctx')
atmsel.superpose(mdl2, aln)
```

Example: examples/commands/swap_atoms_in_res.py

```
# This script illustrates the use of the swap_atoms_in_res
# argument to the Selection.superpose() command:

# Need to make sure that the topologies of the two molecules
# superposed are exactly the same:

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

atfil = '../atom_files/pdb1fdn.ent'
mdl = complete_pdb(env, atfil)
aln = Alignment(env)
aln.append_model(mdl, align_codes='orig')

mdl2 = Model(env, file='1fdn.swap.atm')
aln.append_model(mdl2, align_codes='swap')
atmsel = Selection(mdl)
atmsel.superpose(mdl2, aln, swap_atoms_in_res='')
atmsel.superpose(mdl2, aln, swap_atoms_in_res='DEFHLNQRVY', fit=False)
atmsel.superpose(mdl2, aln, swap_atoms_in_res='', fit=True)
```

6.9.23 Selection.rotate_dihedrals() — change dihedral angles

`rotate_dihedrals(deviation, change, dihedrals=('PHI', 'PSI', 'CHI1', 'CHI2', 'CHI3', 'CHI4'))`

Requirements:
for change='OPTIMIZE': topology & restraints
for change='RANDOMIZE': topology

This command changes the dihedral angles of the selected residues. A residue is selected if any of its atoms is in the atom selection.

change selects an optimization (when equal to 'OPTIMIZE') or randomization (when equal to 'RANDOMIZE'):

1. When optimizing, this command finds the first selected restraint that restrains the specified dihedral angle of each selected residue. It then sets the value of that dihedral to the most likely value.

2. When randomizing, the command changes the specified dihedral angle of each selected residue by adding a random value distributed uniformly from −deviation to +deviation degrees.

dihedrals can be either a vector of dihedral angle names or a single string containing all the dihedral angle names separated by blanks. The dihedral angles involved in cyclic structures are not changed (*e.g.*, sidechain dihedral angles in disulfide bonds and prolines). The dihedral angles that can be changed are listed at the top of the `$RESDIH_LIB` library: `alpha, phi, psi, omega, chi1, chi2, chi3, chi4, chi5`. Dihedral angle `'alpha'` is the virtual C_α dihedral angle defined by four consecutive C_α atoms.

The bond connectivity of the MODEL has to exist before this command is executed. If you read in the model by **Model.read()**, the bond connectivity is defined by subsequent calls to **Topology.append()** and **Model.generate_topology()** (also make sure that sequence entry does not exist in the alignment or that no alignment is in memory).

Example: examples/commands/rotate_dihedrals.py

```
# Example for: Selection.rotate_dihedrals()

from modeller import *
from modeller.scripts import complete_pdb

# This will optimize and randomize dihedrals in a MODEL
env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Select dihedral angle types for optimization and randomization:
dih = 'phi psi omega chi1 chi2 chi3 chi4 chi5'

# Read the sequence, get its topology and coordinates:
mdl = complete_pdb(env, '1fas')

# Select all atoms
atmsel = Selection(mdl)

atmsel.rotate_dihedrals(change='RANDOMIZE', deviation=90.0, dihedrals=dih)
mdl.write(file='1fas.ini1')

# Get restraints from somewhere and optimize dihedrals:
mdl.restraints.make(atmsel, restraint_type='stereo', spline_on_site=False)
atmsel.rotate_dihedrals(change='OPTIMIZE', deviation=90.0, dihedrals=dih)
mdl.write(file='1fas.ini2')
```

6.9.24 Selection.unbuild() — undefine coordinates

```
unbuild()
```

This command undefines all of the Cartesian coordinates of the selected atoms.

6.9.25 Selection.hot_atoms() — atoms violating restraints

```
hot_atoms(pick_hot_cutoff, residue_span_range=(0, 99999), viol_report_cut=physical.Values(default=4.500000,
chi1_dihedral=999.000000, chi2_dihedral=999.000000, chi3_dihedral=999.000000,
```

`chi4_dihedral=999.000000, chi5_dihedral=999.000000, phi_psi_dihedral=6.500000, nonbond_spline=999.000000, accessibility=999.000000, density=999.000000, gbsa=999.000000, em_density=999.000000), schedule_scale=None, edat=None)`

Output: Selection

This command evaluates the energy for all atoms in the selection, and returns a new selection containing atoms that should be optimized to remove hot spots in the model; only selected restraints are considered. The scaling factors for the physical restraint types are given by schedule_scale.

More precisely, the command first flags violated selected atoms. An atom is violated if it is part of a violated restraint. A restraint of physical group x (Table 6.1) is violated when its relative heavy violation (see Section 5.3.1) is larger than specified in viol_report_cut[x].

The command then flags those selected atoms that are within the pick_hot_cutoff angstroms of any of the already flagged atoms.

It is often sensible to follow this command with **Selection.extend_by_residue()**, to select sidechains and neighboring residues.

This command is usually followed by the **Restraints.pick()** command, to select all the restraints that operate on selected (hot) atoms, and then an optimization (see Section 6.11).

Example: examples/commands/pick_hot_atoms.py

```
# Example for: Selection.hot_atoms()

# This will pick atoms violated by some restraints (bond length restraints
# here), select restraints operating on violated atoms, and calculate the
# energy for the selected restraints only (note that a list of violated
# restraints can be obtained by the ENERGY command alone).

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.edat.dynamic_sphere = False
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read the sequence, calculate its topology and coordinates:
mdl = complete_pdb(env, "1fas")

# Just to get some violations:
atmsel = Selection(mdl)
atmsel.randomize_xyz(deviation=0.06)
# Create the bond length restraints and ignore the hard sphere overlap:
mdl.restraints.make(atmsel, restraint_type='bond', spline_on_site=False)
# Pick hot residues and the corresponding violated and neighboring restraints:
atmsel = atmsel.hot_atoms(pick_hot_cutoff=4.0).by_residue()
mdl.restraints.unpick_all()
mdl.restraints.pick(atmsel)
# Calculate the energy of the selected restraints and write them out in detail:
atmsel.energy(output='VERY_LONG')
```

6.9.26 Selection.energy() — evaluate atom selection given restraints

```
energy(asgl_output=False, normalize_profile=False, residue_span_range=(0, 99999), output='LONG',
file='default', viol_report_cut=physical.Values(default=4.500000, chi1_dihedral=999.000000,
chi2_dihedral=999.000000, chi3_dihedral=999.000000, chi4_dihedral=999.000000,
chi5_dihedral=999.000000, phi_psi_dihedral=6.500000, nonbond_spline=999.000000,
accessibility=999.000000, density=999.000000, gbsa=999.000000, em_density=999.000000),
viol_report_cut2=physical.Values(default=2.000000), smoothing_window=3, schedule_scale=None,
edat=None)
```

Output: (molpdf, terms)

Requirements: restraints

The main purpose of this command is to compare spatial features of the atom selection with the selected restraints in order to determine the violations of the molecular pdf. It lists variable amounts of information about the values of the basis, feature, and molecular pdf's for the current MODEL. All arguments that affect the value of the molecular pdf are also relevant for the **Selection.energy()** command.

The scaling factors for the physical restraint types are given by schedule_scale. This allows easy reporting of only a selected subset of all restraints.

Most of the output goes to the `log` file. The output of the **Selection.energy()** command has to be examined carefully, at least at the end of the optimization, when the final model is produced. Additional output files, for the ASGL plotting program are created if asgl_output = `True` (undocumented).

output selects various kinds of output information:

- `'LONG'` writes restraint violations one per line to the `log` file.
- `'VERY_LONG'` writes the most detailed examination of the selected basis and feature pdf's to the `log` file, using several lines of output for each restraint.
- `'NO_REPORT'` suppresses the output of the violated restraints summary (unless profiles are also requested) and also that of nonbond clashes.
- `'GRADIENT'` writes the magnitudes of the 'force' gradients for the currently selected restraints to the isotropic temperature factors (B_{iso}) for each atom of the current MODEL.
- `'ENERGY_PROFILE'` or `'VIOLATIONS_PROFILE'` write out residue energies or heavy relative violations to a file and to the B_{iso} column (see below).

viol_report_cut is a vector with one real number for each physical restraint type. A restraint is reported when its 'heavy relative violation' (see Section 5.3.1) is larger than the corresponding cutoff.

viol_report_cut2 is similar to viol_report_cut, except that it contains cutoffs for restraint 'energies', not heavy relative violations.

The meaning of various other reported properties of the violated restraints is briefly described in the `log` file. For interpreting the seriousness of violations, use the following rule of thumb: There should be at most a few small violations (*e.g.*, 4 standard deviations) for all monomodal restraints. In comparative modeling, the monomodal restraints include the stereochemical restraints and distance restraints when only one homologous structure is used. For the multimodal restraints, there are usually many violations reported because the heaviest violations are used in deciding whether or not to report a violation. In comparative modeling, the multimodal restraints include the χ_i restraints, (Φ, Ψ) binormal restraints and distance restraints when more than one template is used. See also Section 3.1, Question 13.

For profiles:

This command calculates residue energies or heavy relative violations, depending on output, for all physical restraint types (see Table 6.1). Relative heavy violations (see Section 5.3.1) are used because only *relative* violations of different features are comparable. In both cases, the residue sum is the sum over all restraints that have at least one atom in a given residue. The contribution of each restraint is counted exactly once for each residue, without any weighting. Restraints spanning more than one residue contribute equally to all of them. Thus, the sum of residue energies is generally larger than molecular pdf. The command also calculates the sum over all physical restraint types of the contributions for each residue and then writes all the contributions, plus this sum, as columns in a a file suitable for plotting by a plotting program such as ASGL or GNUPLOT.

If normalize_profile is `True` the profile for each residue is normalized by the number of terms applying to each residue.

All the curves are smoothed by the running window averaging method if smoothing_window is larger than 0: The window is centered on residue i and extends for (smoothing_window/2) - 1 residues on each side. Thus, smoothing_window has to be an even number (or it is made such by the program automatically). The only exceptions are the two termini, where a smaller number of residues are available for smoothing. The relative weight of residue j when calculating the smoothed value at residue i is (smoothing_window$/2 - |j - i|$).

The energy or the violations profile (sum over all restraint types) is also written to the B_{iso} field of the model (the temperature factor for PDB X-ray structures). Note that all the atoms in one residue get the same number. This output is useful for exploring the violations on a graphics terminal.

This function returns the total value of the objective function, molpdf, and the contributions from each physical restraint type, terms.

Example: examples/scoring/energy.py

```
# Example for: Selection.energy()

# This will calculate the stereochemical energy (bonds,
# angles, dihedrals, impropers) for a given model.

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

def patch_disulfides(mdl):
    # Must patch disulfides here to calculate the non-bonded
    # energy properly. Also, when you use hydrogens, disulfides
    # must always be patched so that sulfhydril hydrogens are
    # removed from the model.
    for ids in [ ('17:A', '39:A'),
                 ( '3:A', '22:A'),
                 ('53:A', '59:A'),
                 ('41:A', '52:A') ]:
        mdl.patch(residue_type='DISU', residues=[mdl.residues[r] for r in ids])

mdl = complete_pdb(env, "1fas", special_patches=patch_disulfides)

# Select all atoms
atmsel = Selection(mdl)

mdl.restraints.make(atmsel, restraint_type='stereo', spline_on_site=False)

# Actually calculate the energy
(molpdf, terms) = atmsel.energy(edat=EnergyData(dynamic_sphere=True))

# molpdf is the total 'energy', and terms contains the contributions from
# each physical type. Here we print out the bond length contribution:
print("Bond energy is %.3f" % terms[physical.bond])
```

6.9.27 Selection.debug_function() — test code self-consistency

```
debug_function(residue_span_range=(0, 99999), debug_function_cutoff=(0.01, 0.001, 0.1),
detailed_debugging=False, schedule_scale=None, edat=None)
```

Output: n_exceed

This command checks the self-consistency of the code for the objective function and its derivatives by calculating and comparing numeric and analytical derivatives. All the parameters influencing the evaluation of the molecular pdf are also relevant (see **Selection.energy()**). The derivative is reported if both the absolute difference and the fractional difference between the two kinds of evaluations exceed debug_function_cutoff[0] and debug_function_cutoff[1], respectively. This command returns n_exceed, the number of such reported differences.

The scaling factors for the physical restraint types are given by schedule_scale. This allows some restraints to be turned off (scaled to zero) for the purpose of this test, if required.

When detailed_debugging is `True`, the analytic and numeric derivatives of each restraint with respect to atomic positions are also compared for the atoms 'violated' by the whole molecular pdf. The absolute cutoff for writing out the discrepancies is scaled by debug_function_cutoff[2]; the relative cutoff remains the same as before.

Example: examples/scoring/debug_function.py

```
# Example for: Selection.debug_function()

# This will use the MODELLER AutoModel class to construct homology
# restraints for 1fas. It will then use Model.debug_function() to test
# the source code for the function and derivatives calculation
# by comparing analytical and numerical first derivatives (note that
# AutoModel is a derived class of model, so all 'model' methods will work
# on 'AutoModel'). Some discrepancies may be reported but ignore them here.

from modeller import *
from modeller.automodel import AutoModel    # Load the AutoModel class

log.verbose()
env = Environ()
env.io.atom_files_directory = ['../atom_files']

a = AutoModel(env, alnfile = 'debug_function.ali',
              knowns  = ('2ctx', '2nbt'), sequence = '1fas')
a.spline_on_site = False
a.make(exit_stage=1)

# Test on all atoms
atmsel = Selection(a)

# To assign 0 weights to restraints whose numerical derivatives
# code does not work (i.e., splines for angles and dihedrals):
scal = physical.Values(default=1.0, lennard_jones=0, coulomb=0, h_bond=0,
                       phi_dihedral=0, psi_dihedral=0, omega_dihedral=0,
                       chi1_dihedral=0, chi2_dihedral=0, chi3_dihedral=0,
                       chi4_dihedral=0, disulfide_angle=0,
                       disulfide_dihedral=0, chi5_dihedral=0)
atmsel.energy(output='SHORT', schedule_scale=scal)
atmsel.debug_function(debug_function_cutoff=(15.00, 0.10, 0.1),
                      detailed_debugging=True, schedule_scale=scal)
```

6.9.28 Selection.assess_dope() — assess a model selection with the DOPE method

`assess_dope(**vars)`

Output: molpdf

This command assesses the quality of the selected atoms in the model using the DOPE (Discrete Optimized Protein Energy) method [Shen & Šali, 2006]. (See **Selection.assess()** for assessment with SOAP and other potentials.) This is a statistical potential optimized for model assessment. As with **Model.assess_ga341()**, the benchmark set used to develop this method contained only single-chain proteins, and thus no guarantees can be made about the applicability of the method to multiple-chain systems.

DOPE uses the standard MODELLER energy function, so any of the arguments accepted by **Selection.energy()** can also be used here. (Note also that the model's topology must be set up in order to calculate the energy, which can be done for you by the **complete_pdb()** script.)

Only the DOPE energy itself is returned by this command (all other components of the MODELLER energy function, such as stereochemical restraints, Lennard-Jones interactions, homology-derived restraints, *etc*, are ignored) unless you manually set schedule_scale. See **Selection.energy()** for more details. Note that the assessment uses a custom `EnergyData` object, so any changes you make to the selection's `EnergyData` (*e.g.*, changing the EnergyData.contact_shell cutoff distance or setting EnergyData.nonbonded_sel_atoms) will not be honored. Note also that any intra-rigid body distances are *not* considered as part of the DOPE assessment (see Section 5.3.4), so if you have any defined rigid bodies, you may want to consider turning them off before requesting this assessment.

The DOPE model score is designed for selecting the best structure from a collection of models built by MODELLER. (For example, you could build multiple `AutoModel` models by setting AutoModel.ending_model, and select the model that returns the lowest DOPE score.) The score is unnormalized with respect to the protein size and has an arbitrary scale, therefore scores from different proteins cannot be compared directly. If you wish to do this, use **Model.assess_normalized_dope()** instead, which returns a Z-score.

When using `AutoModel` or `LoopModel`, automatic DOPE assessment of each model can be requested by adding `assess.DOPE` to AutoModel.assess_methods or LoopModel.loop.assess_methods respectively.

Example: examples/assessment/assess_dope.py

```
# Example for: Selection.assess_dope()

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Read a model previously generated by Modeller's AutoModel class
mdl = complete_pdb(env, '../atom_files/1fdx.B99990001.pdb')
```

```
# Select all atoms in the first chain
atmsel = Selection(mdl.chains[0])

score = atmsel.assess_dope()
```

6.9.29 Selection.assess_dopehr() — assess a model with the DOPE-HR method

`assess_dopehr(**vars)`

Output: molpdf

This command assesses the quality of the model using the DOPE-HR method. This is very similar to the original DOPE method (see **Selection.assess_dope()**) but is obtained at higher resolution (using a bin size of 0.125Å rather than 0.5Å).

When using `AutoModel` or `LoopModel`, automatic DOPE-HR assessment of each model can be requested by adding `assess.DOPEHR` to AutoModel.assess_methods or LoopModel.loop.assess_methods respectively.

Example: See **Selection.assess_dope()** command.

6.9.30 Selection.get_dope_profile() — get per-residue DOPE profile

`get_dope_profile()`

Output: `EnergyProfile`

This returns the individual residue components of the DOPE score, which can be used to detect poor regions of the model. See **Selection.assess_dope()** for more details on DOPE, and Section 6.25 for more details on the returned profile.

6.9.31 Selection.get_dopehr_profile() — get per-residue DOPE-HR profile

`get_dopehr_profile()`

Output: `EnergyProfile`

This returns the individual residue components of the DOPE-HR score, which can be used to detect poor regions of the model. See **Selection.assess_dopehr()** for more details on DOPE-HR, and Section 6.25 for more details on the returned profile.

6.9.32 Selection.assess() — assess a model selection

`assess(assessor, output='SHORT NO_REPORT', **vars)`

Output: score

This command assesses the quality of the selected atoms in the model using the provided assessor object. Typically this is used for SOAP scoring, with assessor being **soap_loop.Scorer()** or **soap_protein_od.Scorer()**.

Assessment uses the standard MODELLER energy function, so any of the arguments accepted by **Selection.energy()** can also be used here. See **Selection.assess_dope()** for more details on this and schedule_scale.

Any of the assessor objects accepted by this function can also be used for automatic assessment of each `AutoModel` or `LoopModel` model; see Section 2.2.3 or Section 2.3.3 for examples.

Example: examples/assessment/assess_soap_protein.py

```
# Example for: Selection.assess(), soap_protein_od.Scorer()

from modeller import *
from modeller.scripts import complete_pdb
from modeller import soap_protein_od

env = Environ()
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Set up SOAP-Protein-OD scoring (note: if assessing multiple models, it is
# best to create 'sp' just once and keep it around, since reading in the
# potential from disk can take a long time).
sp = soap_protein_od.Scorer()

# Read a model previously generated by Modeller's AutoModel class
mdl = complete_pdb(env, '../atom_files/1fdx.B99990001.pdb')

# Select all atoms in the first chain
atmsel = Selection(mdl.chains[0])

# Assess with the above Scorer
try:
    score = atmsel.assess(sp)
except ModellerError:
    print("The SOAP-Protein-OD library file is not included with MODELLER.")
    print("Please get it from https://salilab.org/SOAP/.")
```

6.10 The `physical` module: contributions to the objective function

The `physical` module defines all of the physical restraint types (see Table 6.1). It also defines a `physical.Values` class, which allows values for some or all of these types to be specified, for use as energy scaling parameters, cutoffs, *etc*.

Please note that the physical restraint types are currently hard-coded into the MODELLER program; you cannot add new types.

6.10.1 physical.Values() — create a new set of physical values

`Values(default=1.0, **keys)`

This creates a new empty `Values` object. This is very similar to a Python dictionary; valid keys are `'default'` or any of the objects from Table 6.1. For example, if `'v'` is a new object, you can set the value for the bond angle contribution to `'0.5'` with `'v[physical.angle] = 0.5'`. If you try to read a physical type from `'v'` which is not set, you'll get `v['default']`. As a convenience, you can set initial values for the default and/or physical types by passing them as parameters to the `'physical.Values()'` constructor, without the `'physical.'` prefix. For example `'physical.Values(default=1.0, h_bond=0.1, coulomb=0.1)'` would scale all types by 1.0 except for the H bond and Coulomb terms.

Example: See **Selection.debug_function()** command.

Python object	Index	Group
physical.bond	1	Bond length potential
physical.angle	2	Bond angle potential
physical.dihedral	3	Stereochemical cosine dihedral potential
physical.improper	4	Stereochemical improper dihedral potential
physical.soft_sphere	5	soft-sphere overlap restraints
physical.lennard_jones	6	Lennard-Jones 6–12 potential
physical.coulomb	7	Coulomb point-point electrostatic potential
physical.h_bond	8	H-bonding potential
physical.ca_distance	9	Distance restraints 1 (C_α–C_α)
physical.n_o_distance	10	Distance restraints 2 (N–O)
physical.phi_dihedral	11	Mainchain Φ dihedral restraints
physical.psi_dihedral	12	Mainchain Ψ dihedral restraints
physical.omega_dihedral	13	Mainchain ω dihedral restraints
physical.chi1_dihedral	14	Sidechain χ_1 dihedral restraints
physical.chi2_dihedral	15	Sidechain χ_2 dihedral restraints
physical.chi3_dihedral	16	Sidechain χ_3 dihedral restraints
physical.chi4_dihedral	17	Sidechain χ_4 dihedral restraints
physical.disulfide_distance	18	Disulfide distance restraints
physical.disulfide_angle	19	Disulfide angle restraints
physical.disulfide_dihedral	20	Disulfide dihedral angle restraints
physical.lower_distance	21	X lower bound distance restraints
physical.upper_distance	22	X upper bound distance restraints
physical.sd_mn_distance	23	Distance restraints 3 (SDCH–MNCH)
physical.chi5_dihedral	24	Sidechain χ_5 dihedral restraints
physical.phi_psi_dihedral	25	(Φ, Ψ) binomial dihedral restraints
physical.sd_sd_distance	26	Distance restraints 4 (SDCH–SDCH)
physical.xy_distance	27	Distance restraints 5 (X–Y)
physical.nmr_distance	28	NMR distance restraints 6 (X–Y)
physical.nmr_distance2	29	NMR distance restraints 7 (X–Y)
physical.min_distance	30	Minimal distance restraints
physical.nonbond_spline	31	Non-bonded spline restraints
physical.accessibility	32	Atomic accessibility restraints
physical.density	33	Atom density restraints
physical.absposition	34	Absolute position restraints
physical.dihedral_diff	35	Dihedral angle difference restraints
physical.gbsa	36	GBSA implicit solvent potential
physical.em_density	37	EM density fitting potential
physical.saxs	38	SAXS restraints
physical.symmetry	39	Symmetry restraints

Table 6.1: *List of "physical" restraint types.*

6.11 The optimizers module: optimization of the model

The `optimizers` module provides a number of methods to optimize a model. The molecular pdf is optimized with respect to the selected coordinates, and the optimized coordinates are returned.

These optimizers are often used to implement the variable target function method, for example in the `AutoModel` and `LoopModel` classes. See Section 6.12 for an example.

6.11.1 ConjugateGradients() — optimize atoms given restraints, with CG

```
ConjugateGradients(output='NO_REPORT', min_atom_shift=0.01, residue_span_range=(0, 99999),
**vars)
```

Output: molpdf

Requirements: restraints

This command creates a new Python optimizer object. Calling the object's `optimize` method with an atom selection then performs a number of optimizing iterations using a modified version of the Beale restart conjugate gradients method [Shanno & Phua, 1980,Shanno & Phua, 1982]. A brief description of the algorithm is given in Section A.2.

The optimization can be controlled with a number of keyword arguments, which can be specified either when the object is created, or when the `optimize` method is called (if the same keyword is specified in both, that for the `optimize` method takes precedence). Valid keywords are:

- min_atom_shift is a convergence criterion for the optimization. When the maximal atomic shift is less than the specified value, the optimization is finished regardless of the number of optimization cycles or function value and its change.
- max_iterations is used to prevent a waste of CPU time in the optimization. When that many calls of the objective function are done, the optimization is finished regardless of the maximal atomic shift. (Note that each optimization step usually requires more than one call of the objective function.)
- output, if `'REPORT'`, writes a summary of the optimization results to the `log` file after optimization. If it is `'NO_REPORT'`, no such report is written.
- edat is an `EnergyData` object containing objective function parameters, if you do not want to use the defaults. See Section 6.3 for more information.
- schedule_scale specifies scaling factors for the physical restraint types, if you do not want to use the defaults.
- residue_span_range determines what atom pairs can possibly occur in the non-bonded atom pairs list used for dynamic restraints (see Section 5.3).
- actions, if set, should be a list of periodic actions. Each is a Python object containing an action which is carried out periodically during the optimization, after every step. For example, **actions.WriteStructure()** can be used to write out a PDB file with structure snapshots during the run, while **actions.Trace()** writes basic information about the optimization to a trace file. If multiple actions are given, they are run in the order they are given.

It is useful in some simulations to be able to set EnergyData.contact_shell to something large (*e.g.*, 8Å) and EnergyData.update_dynamic to `999999.9`, so that the pairs list is prepared only at the beginning of the optimization. However, you have to make sure that the potential energy is not invisibly pumped into the system by making contacts that are not on the list of non-bonded pairs (see below).

The `optimize` method, when called, returns molpdf, the value of the objective function at the end of optimization. An exception is raised if optimization is aborted because dynamic restraints could not be calculated as a result of a system being too large. It is up to the calling script to ensure that sensible action is taken; *e.g.*, skipping the rest of modeling for the model that resulted in an impossible function evaluation. This option is useful when calculating several independent models and you do not want one bad model to abort the whole calculation. A probable reason for an interrupted optimization is that it was far from convergence by the time the calculation of dynamic restraints was first requested. Two possible solutions are: (1) optimize more thoroughly (*i.e.* slowly) and (2) use a different contact pairs routine (set EnergyData.nlogn_use = 9999).

Example: examples/scoring/optimize.py

```
# Example for: ConjugateGradients(), MolecularDynamics(), Model.switch_trace()

# This will optimize stereochemistry of a given model, including
# non-bonded contacts.

from modeller import *
from modeller.scripts import complete_pdb
from modeller.optimizers import ConjugateGradients, MolecularDynamics, actions

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.edat.dynamic_sphere = True

env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

code = '1fas'
mdl = complete_pdb(env, code)
mdl.write(file=code+'.ini')

# Select all atoms:
atmsel = Selection(mdl)

# Generate the restraints:
mdl.restraints.make(atmsel, restraint_type='stereo', spline_on_site=False)
mdl.restraints.write(file=code+'.rsr')

mpdf = atmsel.energy()

# Create optimizer objects and set defaults for all further optimizations
cg = ConjugateGradients(output='REPORT')
md = MolecularDynamics(output='REPORT')

# Open a file to get basic stats on each optimization
trcfil = open(code+'.D00000001', 'w')

# Run CG on the all-atom selection; write stats every 5 steps
cg.optimize(atmsel, max_iterations=20, actions=actions.Trace(5, trcfil))
# Run MD; write out a PDB structure (called '1fas.D9999xxxx.pdb') every
# 10 steps during the run, and write stats every 10 steps
md.optimize(atmsel, temperature=300, max_iterations=50,
            actions=[actions.WriteStructure(10, code+'.D9999%04d.pdb'),
                     actions.Trace(10, trcfil)])
# Finish off with some more CG, and write stats every 5 steps
cg.optimize(atmsel, max_iterations=20,
            actions=[actions.Trace(5, trcfil)])

mpdf = atmsel.energy()

mdl.write(file=code+'.B')
```

6.11.2 QuasiNewton() — optimize atoms with quasi-Newton minimization

```
QuasiNewton(output='NO_REPORT', min_atom_shift=0.01, max_atom_shift=100.0,
residue_span_range=(0, 99999), **vars)
```

Output: molpdf

Requirements: restraints

This functions in a very similar way to **ConjugateGradients()**, but uses a variable metric (quasi-Newton) method instead to find the minimum. The algorithm implemented in MODELLER is the BFGS or Broyden-Fletcher-Goldfarb-Shanno method [Press *et al.*, 1992]. It takes the same keyword arguments as **ConjugateGradients()**, plus one additional max_atom_shift argument. This is used to limit the maximum size of an optimization move.

6.11.3 MolecularDynamics() — optimize atoms given restraints, with MD

```
MolecularDynamics(output='NO_REPORT', cap_atom_shift=0.2, md_time_step=4.0,
init_velocities=True, temperature=293.0, md_return='FINAL', equilibrate=999999,
guide_factor=0.0, guide_time=0.0, friction=0.0, residue_span_range=(0, 99999), **vars)
```

Output: molpdf

Requirements: restraints

This command creates a new Python optimizer object. Calling the object's `optimize` method with an atom selection then performs a molecular dynamics optimization at a fixed temperature. This is the most basic version of the iterative solver of the Newton's equations of motion. The integrator uses the Verlet algorithm [Verlet, 1967]. All atomic masses are set to that of carbon 12. A brief description of the algorithm is given in Section A.2.

The molecular dynamics optimizer pretends that the natural logarithm of the molecular pdf is energy in kcal/mole. md_time_step is the time step in femtoseconds. temperature is the temperature of the system in Kelvin. max_iterations determines the number of MD steps. If md_return is `'FINAL'` the last structure is returned as the MODEL. If md_return is `'MINIMAL'` then the structure with the lowest value of the objective function on the whole trajectory is returned as the MODEL. Rescaling of velocities is done every equilibrate steps to match the specified temperature. Atomic shifts along one axis are limited by cap_atom_shift (in angstroms). This value should be smaller than EnergyData.update_dynamic. If init_velocities = `True`, the velocity arrays are initialized, otherwise they are not. In that case, the final velocities from the previous run are used as the initial velocities for the current run.

If both guide_factor and guide_time are non-zero, self-guided molecular dynamics [Wu & Wang, 1999] is carried out.

See **ConjugateGradients()** for a description of the other parameters and the edat and actions optional keyword arguments.

Example: See **ConjugateGradients()** command.

6.11.4 actions.WriteStructure() — write out the model coordinates

```
WriteStructure(skip, filepattern, write_all_atoms=True, first=False, last=False, start=0)
```

This action writes out a file containing the current optimizer structure, every skip steps during the optimization. It should be specified in the actions argument to an optimizer object (*e.g.*, **ConjugateGradients()** or **MolecularDynamics()**).

filepattern is a C-style format string, used to construct filenames. It should contain a `%d` format, which is substituted with the model number (*e.g.*, specifying `'file%d.pdb'` would generate files called `'file0.pdb'`, `'file1.pdb'`, `'file2.pdb'`, *etc*). The model number will start with start (or 0, if not given).

If write_all_atoms is `True` (the default) then all atoms in the model are written out to the structure file, whether or not they are selected. If `False`, only selected atoms are written out.

If first is `True`, then the structure at step 0 (before the optimization) is also written out. If last is `True`, then the structure of the last step is written, regardless of whether it is a multiple of skip. By default, both are `False`.

Example: See **ConjugateGradients()** command.

6.11.5 actions.Trace() — write out optimization energies, *etc*

```
Trace(skip, output=None)
```

This action writes out information about the optimization to a trace file every skip steps, starting with the state just before the optimization (step 0). The type of information depends on the type of optimization being carried out, but generally includes the iteration number, energy values, and atomic shifts.

output can be a standard Python `file` object, to which the trace is written, or a file name. In the latter case, a file with that name is created (overwriting any existing file). If output is not specified, the trace is written to the logfile instead.

Example: See **ConjugateGradients()** command.

6.11.6 actions.CHARMMTrajectory() — write out a CHARMM trajectory

```
CHARMMTrajectory(skip, filename, first=False, last=False)
```

This action writes out a trajectory file in CHARMM or X-PLOR format. This is more efficient than **actions.WriteStructure()**, as binary files are smaller than multiple PDB files, and only the moving (selected) atom coordinates are written at each step after the first. Binary trajectory files can be read in by visualization software such as CHIMERA or VMD. [6] You will typically also need a CHARMM-format PSF file to accompany the trajectory, which you can obtain with **Model.write_psf()**.

To use, create a `charmm_trajectory` object, and pass it in the actions argument to an optimizer object (*e.g.*, **ConjugateGradients()** or **MolecularDynamics()**).

If first is `True`, then the structure at step 0 (before the optimization) is also written out. If last is `True`, then the structure of the last step is written, regardless of whether it is a multiple of skip. By default, both are `False`.

Example: examples/python/trajectory.py

```
# Example for PSF and binary trajectory output

from modeller import *
from modeller.scripts import complete_pdb
from modeller.optimizers import MolecularDynamics, actions

env = Environ()
```

[6]Note that binary trajectory files are machine dependent; it is up to the visualization software to do any necessary byte-swapping.

```
env.io.atom_files_directory = ['../atom_files']
env.edat.dynamic_sphere = True
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

code = '1fas'
mdl = complete_pdb(env, code)

# Stereochemical restraints on all atoms:
atmsel = Selection(mdl)
mdl.restraints.make(atmsel, restraint_type='stereo', spline_on_site=False)

# Write a PSF
mdl.write_psf(code+'.psf')

# Run 100 steps of MD, writing a CHARMM binary trajectory every 5 steps
md = MolecularDynamics(output='REPORT')
md.optimize(atmsel, temperature=300, max_iterations=100,
            actions=actions.CHARMMTrajectory(5, filename=code+'.dcd'))
```

6.11.7 User-defined optimizers

The `optimizers` module also provides a `StateOptimizer` class. This class cannot be directly used to optimize the system, but instead it can be used as a base for you to write your own optimization algorithms in Python. To do this, create a subclass and override the `optimize` method to do your optimization. Your optimizer does not act directly on the atom coordinates, but instead gets a 'state' vector with the same number of elements as there are degrees of freedom in the system. (This allows you to also optimize rigid bodies, for example, without having to worry about the specifics of their representation.)

Several utility functions are provided:

- `'self.get_state()'`: returns a state vector representing the current state of the system (x,y,z coordinates of all non-rigid atoms in the selection, and center of mass and rotation angles of all selected rigid bodies).
- `'self.energy(state)'`: given a state vector, returns the system energy and a similar vector of state gradients. Also updates the atom shifts `self.shiftavr` and `self.shiftmax` from the previous state.
- `'self.next_step()'`: updates the step counter `self.step`, and does any periodic actions, if defined.
- `'self.finish()'`: does any cleanup at the end of the optimization.

If you want to define parameters for your optimization in the same way as the other optimizers, set `'_ok_keys'` appropriately and then call `self.get_parameter()` to get their values.

Example: examples/python/steepest_descent.py

```
from modeller.optimizers import StateOptimizer

class SteepestDescent(StateOptimizer):
    """Very simple steepest descent optimizer, in Python"""

    # Add options for our optimizer
    _ok_keys = StateOptimizer._ok_keys + ('min_atom_shift', 'min_e_diff',
                                          'step_size', 'max_iterations')

    def __init__(self, step_size=0.0001, min_atom_shift=0.01, min_e_diff=1.0,
                 max_iterations=None, **vars):
        StateOptimizer.__init__(self, step_size=step_size,
                                min_atom_shift=min_atom_shift,
                                min_e_diff=min_e_diff,
                                max_iterations=max_iterations, **vars)

    def optimize(self, atmsel, **vars):
        # Do normal optimization startup
        StateOptimizer.optimize(self, atmsel, **vars)

        # Get all parameters
        alpha = self.get_parameter('step_size')
        minshift = self.get_parameter('min_atom_shift')
        min_ediff = self.get_parameter('min_e_diff')
        maxit = self.get_parameter('max_iterations')

        # Main optimization loop
        state = self.get_state()
        (olde, dstate) = self.energy(state)
        while True:
            for i in range(len(state)):
                state[i] -= alpha * dstate[i]
            (newe, dstate) = self.energy(state)
            if abs(newe - olde) < min_ediff:
                print("Finished at step %d due to energy criterion" % self.step)
                break
            elif self.shiftmax < minshift:
                print("Finished at step %d due to shift criterion" % self.step)
                break
            elif maxit is not None and self.step >= maxit:
                print("Finished at step %d due to step criterion" % self.step)
                break
            if newe < olde:
                alpha *= 2
            else:
                alpha /= 2
            olde = newe
            self.next_step()
        self.finish()
```

Example: examples/python/steepest_descent_test.py

```
from modeller import *
from modeller.optimizers import actions
from modeller.scripts import complete_pdb

# Load our custom steepest descent optimizer
from steepest_descent import SteepestDescent

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')
```

```
# Read in the initial structure:
code = '1fdn'
mdl = complete_pdb(env, code)
atmsel = Selection(mdl)

# Generate the restraints:
mdl.restraints.make(atmsel, restraint_type='stereo', spline_on_site=False)

# Optimize with our custom optimizer:
opt = SteepestDescent(max_iterations=80)
opt.optimize(atmsel, actions=actions.Trace(5))
```

6.12 The Schedule class: variable target function optimization

The Schedule class is used for variable target function optimization (the initial optimization used by the AutoModel class).

6.12.1 Schedule() — create a new schedule

```
Schedule(last_scales, steps)
```

This creates a new Schedule object, which can contain multiple schedule steps, given by the list steps. Each step then defines some of the optimization parameters: (1) the optimization method; (2) maximal number of residues that the restraints are allowed to span (Section 6.7.7); (3) the individual scaling factors for all the physical restraint types. last_scales is used by **Schedule.make_for_model()**.

The usual schedule for the variable target function part of optimization in comparative modeling is as follows. The residue range (**Restraints.pick()** and Section 6.7.7) is increased with increasingly larger steps until the protein length is reached. The scaling of homology-derived and bonded stereochemical restraints increases from a small value to 1 in the initial few steps to allow for imperfect starting geometries, especially those that result from **Selection.randomize_xyz()** and long insertions or deletions. (For AutoModel, the restraints are additionally scaled by Environ.schedule_scale. This is useful when template-derived fold restraints have to be weakened relative to some external restraints, so that the fold can actually reflect these external restraints, even when they are quite different from the template-derived restraints.) The soft-sphereoverlap restraints are slowly introduced only in the last four steps of the variable target function method to save CPU time and increase the radius of convergence.

In comparative modeling by the AutoModel class in the default mode, the variable target function method is usually followed by simulated annealing with molecular dynamics. In this last stage, all homology-derived and stereochemical restraints are generally used scaled only by Environ.schedule_scale. Thus, it is recommended that if you define your own schedule, the scaling factors for the last step are all 1, so that the energy surface followed in optimization is continuous.

There are a number of variables defined in the AutoModel class that can be used to influence the thoroughness of both the variable target function and molecular dynamics parts of the optimization; see Section 2.2.2.

Example: examples/commands/make_schedule.py

```
# This will create a VTFM optimization schedule and then
# use it to optimize the model

from modeller import *
from modeller.scripts import complete_pdb

# Load in optimizer and schedule support
from modeller import schedule, optimizers

log.verbose()

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.edat.dynamic_sphere = True

env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')
code = '1fas'
mdl = complete_pdb(env, code)

# Generate the restraints:
```

```
atmsel = Selection(mdl)
mdl.restraints.make(atmsel, restraint_type='stereo', spline_on_site=False)

# Create our own library schedule:
# 5 steps of conjugate gradients (CG), each step using a larger
# residue range (2 up to 9999) and energy scaling factor (0.01 up to 1.0),
# followed by 3 steps of molecular dynamics (MD) at successively lower
# temperature. The scaling factors for the last 5 steps are always retained.
CG = optimizers.ConjugateGradients
MD = optimizers.MolecularDynamics
libsched = schedule.Schedule(5,
          [ schedule.Step(CG, 2, physical.Values(default=0.01)),
            schedule.Step(CG, 5, physical.Values(default=0.1)),
            schedule.Step(CG, 10, physical.Values(default=0.2)),
            schedule.Step(CG, 50, physical.Values(default=0.5)),
            schedule.Step(CG, 9999, physical.Values(default=1.0)),
            schedule.Step(MD(temperature=300.), 9999, \
                          physical.Values(default=1.0)),
            schedule.Step(MD(temperature=200.), 9999, \
                          physical.Values(default=1.0)),
            schedule.Step(MD(temperature=100.), 9999, \
                          physical.Values(default=1.0)) ])

# Make a trimmed schedule suitable for our model, and scale it by schedule_scale
mysched = libsched.make_for_model(mdl) * env.schedule_scale

# Write the trimmed schedule to a file
fh = open(code+'.sch', 'w')
mysched.write(fh)
fh.close()

# Optimize for all steps in the schedule
for step in mysched:
    step.optimize(atmsel, output='REPORT', max_iterations=200)

mdl.write(file=code+'.B')
```

6.12.2 Schedule.make_for_model() — trim a schedule for a model

`make_for_model(mdl)`

This takes the input schedule, and returns a new schedule, trimmed to the right length for mdl. Schedule steps are taken from the input schedule in order, finishing when the first step with a residue range greater than or equal to the number of residues in mdl is reached, unless the range is 9999. The value of last_scales for the input schedule is also considered; the last last_scales entries in the new schedule will always have the same scaling factors as the last last_scales entries in the input schedule, even if trimming occurred.

Example: See **Schedule()** command.

6.12.3 Schedule.write() — write optimization schedule

`write(fh)`

This command writes out the schedule for the variable target function method to the given file or file handle (see **modfile.File()**), `fh`.

Example: See **Schedule()** command.

6.13 The `GroupRestraints` class: restraints on atom groups

The `GroupRestraints` class holds classifications of atoms into classes/groups, and restraints which act on certain atom groups. Such restraints are used, for example, for the statistical potentials (such as DOPE) used for loop modeling. These restraints are only calculated if EnergyData.dynamic_modeller is set to `True`.

6.13.1 GroupRestraints() — create a new set of group restraints

```
GroupRestraints(env, classes, parameters=None)
```

This creates a new set of group restraints. The set is initialized by reading in classes, a file containing a classification of residue:atom pairs into groups. If the parameters argument is also given, this is used to read in a file of restraint parameters, using **GroupRestraints.append()**.

classes can be either a file name or a readable file handle (see **modfile.File()**).

Example: examples/commands/group_restraints.py

```
# Example for: GroupRestraints()
from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Allow calculation of statistical (dynamic_modeller) potential
env.edat.dynamic_modeller = True

mdl = complete_pdb(env, "1fas")

# Read Fiser/Melo loop modeling potential
gprsr = GroupRestraints(env, classes='$(LIB)/atmcls-melo.lib',
                        parameters='$(LIB)/melo1-dist.lib')
# Read DOPE loop modeling potential
#gprsr = GroupRestraints(env, classes='$(LIB)/atmcls-mf.lib',
#                        parameters='$(LIB)/dist-mf.lib')
# Read DOPE-HR loop modeling potential
#gprsr = GroupRestraints(env, classes='$(LIB)/atmcls-mf.lib',
#                        parameters='$(LIB)/dist-mfhr.lib')


# Use this potential for the 1fas model
mdl.group_restraints = gprsr

# Evaluate the loop score of PDB residues 1 through 10 in chain A
atmsel = Selection(mdl.residue_range('1:A', '10:A'))
atmsel.energy()
```

6.13.2 GroupRestraints.append() — read group restraint parameters

```
append(file)
```

This reads a set of parameters from file, which should act on the atom classes previously defined. Any parameters read are added to any already in this object (to clear them all, simply create a new object). The format of the group restraints file is the same as the MODELLER restraints format (see Section 5.3.1) except that rather than numeric atom indices, atom group names (as defined in the classes file) are used. These restraints are further limited, in that they can act only on 1 or 2 atoms.

file can be either a file name or a readable file handle (see **modfile.File()**).

6.14 The gbsa module: implicit solvation

The gbsa module provides methods for scoring models with GB/SA implicit solvation. This is primarily used by the **DOPELoopModel()** class.

Born radii are calculated using the mAGB method [Gallicchio & Levy, 2004].

6.14.1 gbsa.Scorer() — create a new scorer to evaluate GB/SA energies

```
Scorer(library='$LIB/solv.lib', solvation_model=1, cutoff=8.0)
```

This creates a new class to be used for scoring models with the GB/SA implicit solvation model. To activate scoring, you must add an instance of this class to the relevant EnergyData.energy_terms list, in the same way as for user-defined energy terms (see Section 7.1.3).

library is the name of a library file containing radii and solvation parameters for all atom types. solvation_model selects which column of solvation parameters to use from this file. cutoff sets the distance in angstroms used to calculate the Born radii; the calculation can be made faster at the expense of accuracy by using a smaller cutoff. Note that GB/SA uses the same nonbonded list as the other dynamic terms, so cutoff should be chosen to be no greater than EnergyData.contact_shell. The electrostatic component of GB/SA is also switched using the value of EnergyData.coulomb_switch.

Example: examples/scoring/gbsa.py

```
# Example for: gbsa.scorer()
# This will calculate the GB/SA implicit solvation energy for a model.

from modeller import *
from modeller import gbsa
from modeller.scripts import complete_pdb

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

# Calculate just the GB/SA score; turn off soft-sphere
env.edat.dynamic_sphere = False
env.edat.energy_terms.append(gbsa.Scorer())
# GB/SA falls off slowly with distance, so a larger cutoff than the
# default (4.0) is recommended
env.edat.contact_shell = 8.0

mdl = complete_pdb(env, "1fas")

# Select all atoms
atmsel = Selection(mdl)

# Calculate the energy
atmsel.energy()
```

6.15 SOAP potentials

MODELLER includes a number of statistically optimized atomic potentials (SOAP) [Dong *et al.*, 2013]. Individual potentials are optimized for scoring and assessing protein-protein interfaces (`soap_pp`), loops (`soap_loop`), protein-peptide interactions (`soap_peptide`), and protein structures (`soap_protein_od`).

6.15.1 soap_loop.Scorer() — create a new scorer to evaluate SOAP-Loop energies

```
Scorer(library='$LIB/soap_loop.hdf5', group=physical.xy_distance)
```

This creates a new class to be used for assessing or scoring models with SOAP-Loop.

For assessing models (*i.e.*, a one-time score for each model after optimization is complete), simply assign an instance of this class to AutoModel.assess_methods or LoopModel.loop.assess_methods (see Section 2.3.3 for an example). For manual assessment, see **Selection.assess()**.

To add to the energy function (*e.g.*, to use in optimization, which is not usually what you want), you must add an instance of this class to the relevant EnergyData.energy_terms list, in the same way as for user-defined energy terms (see Section 7.1.3). (Note that the default value of 4.0 for EnergyData.contact_shell is too small to be used with SOAP scores. Set it to the undefined value (-999) to be sure not to discard SOAP statistics for longer distances.)

library is the name of the SOAP-Loop library. This library is not included with the MODELLER distribution due to size (it is roughly 500MB) but can be downloaded from the SOAP web site.

SOAP-Loop is an orientation-dependent potential. It can only reliably be used for scoring (not optimization) as its first derivatives are zero.

6.15.2 soap_peptide.Scorer() — create a new scorer to evaluate SOAP-Peptide energies

```
Scorer(library='$LIB/soap_peptide.hdf5', group=physical.xy_distance)
```

See **soap_loop.Scorer()** for more details.

SOAP-Peptide is an orientation-dependent potential. It can only be used for scoring (not optimization) as its first derivatives are zero.

6.15.3 soap_pp.PairScorer() — create a new scorer to evaluate SOAP-PP pairwise energies

```
PairScorer(library='$LIB/soap_pp_pair.hdf5', group=physical.xy_distance)
```

The SOAP-PP potential includes two terms — a pairwise term that scores pairs of atoms that span the protein-protein interface, and an atomic term that scores all atoms for solvent accessibility. Both are typically used for SOAP-PP scoring. The libraries for both are included with MODELLER.

This pairwise term depends only on distance (it is not orientation-dependent) and supports first derivatives.

6.15.4 soap_pp.AtomScorer() — create a new scorer to evaluate SOAP-PP atomistic energies

```
AtomScorer(library='$LIB/soap_pp_atom.hdf5', group=physical.accessibility)
```

See **soap_pp.PairScorer()** for more details.

Note that this term can only be used for scoring (not optimization) as its first derivatives are zero.

6.15.5 soap_pp.Assessor() — assess with all components of the SOAP-PP score

```
Assessor(pair_library='$LIB/soap_pp_pair.hdf5', pair_group=physical.xy_distance,
atom_library='$LIB/soap_pp_atom.hdf5', atom_group=physical.accessibility)
```

This is a convenience class that can be used to assess models (see **soap_loop.Scorer()** for more details) using both components of the SOAP-PP score, **soap_pp.AtomScorer()** and **soap_pp.PairScorer()**.

6.15.6 soap_protein_od.Scorer() — create a new scorer to evaluate SOAP-Protein-OD energies

```
Scorer(library='$LIB/soap_protein_od.hdf5', group=physical.xy_distance)
```

See **soap_loop.Scorer()** for more details.

SOAP-Protein-OD is an orientation-dependent potential. It can only be used for scoring (not optimization) as its first derivatives are zero.

This can also be used for manual assessment (see **Selection.assess()**) or automatic assessment of `AutoModel` models (see Section 2.2.3).

6.16 The `Alignment` class: comparison of sequences and structures

This section describes the commands for reading, writing, making, analyzing and using the alignments of sequences and structures (pairwise and multiple). For the underlying dynamic programming methods see Section A.1.

An `Alignment` object acts like a regular Python list, where each element in the list represents a sequence in the alignment, as a `Sequence` object (see Section 6.17). The list can be indexed in standard Python fashion or by alignment code. Thus, given an `Alignment` object `aln`, `len(aln)` gives the number of sequences in the alignment, `aln[0]` the first sequence, `aln[-1]` the last sequence, and `aln['1abcA']` the sequence with alignment code '1abcA'.

6.16.1 Alignment() — create a new alignment

```
Alignment(env, **vars)
```

This creates a new `Alignment` object; by default, this contains no sequences. If any keyword arguments are given, they are passed to the **Alignment.append()** function to create the initial alignment.

6.16.2 Alignment.comments — alignment file comments

This is a list of all alignment file comments. If adding or changing comments, make sure to keep the required prefix (`'C;'` for PIR files).

6.16.3 Alignment.positions — list of residue-residue alignment positions (including gaps)

This is a list of all 'positions' in the alignment. Unlike individual sequences in the alignment (see Section 6.17), which are lists of the residues in those sequences (not including gaps) the 'positions' correspond to columns in the alignment file, including gaps. Each element in the list contains a method `get_residue(seq)` which, given a sequence in the same alignment, returns the residue in that sequence which is aligned at that position, or `None` if a gap is present.

Note that chain breaks are considered to have zero width, and thus fall 'between' alignment positions; they do not count as alignment positions.

6.16.4 Alignment.append() — read sequences and/or their alignment

```
append(file, align_codes='all', atom_files=None, remove_gaps=True, alignment_format='PIR',
io=None, allow_alternates=False)
```

Output: end_of_file

This command reads the sequence(s) and/or their alignment from a text file. Only sequences with the specified codes are read in; align_codes = `'all'` can be used to read all sequences. The sequences are added to any currently in the alignment.

file can be either a file name or a readable file handle (see **modfile.File()**).

There are several alignment formats:

1. The `'PIR'` format resembles that of the PIR sequence database. It is described in Section B.1 and is used for comparative modeling because it allows for additional data about the proteins that are useful for automated access to the atomic coordinates.
2. The `'FASTA'` format resembles the `'PIR'` format but has a missing second 'comment' line and a missing star at the end of each sequence.

3. The 'PAP' format is nicer to look at but contains less information and is not used by other programs. When used in conjunction with PDB files, the PDB files must contain exactly the residues in the sequences in the 'PAP' file; *i.e.*, it is not possible to use only a segment of a PDB file. In addition, the 'PAP' protein codes must be expandable into proper PDB atom filenames, as described in Section 5.1.3. Alternatively, a list of PDB file names can be specified with the atom_files parameter, in the same order as the sequences read from the alignment file. (atom_files is not used for other alignment formats.) The protein sequence can now start in any column (this was limited to column 11 before release 5).
4. The 'QUANTA' format can be used to communicate with the QUANTA program. You are not supposed to mix 'QUANTA' format with any other format because the 'QUANTA' format contains residue numbers which do not occur in the other formats and are difficult to guess correctly. MODELLER can write out alignments in the 'QUANTA' format but cannot read them in.
5. The 'INSIGHT' format is very similar to the 'PAP' format and can sometimes be used to communicate with the INSIGHTII program. When used in conjunction with PDB files, the same rules as for the 'PAP' format apply.
6. The 'PSS' format is in the .horiz format used by PSI-PRED to report secondary structure predictions of sequences. A confidence of the prediction is also reported as an integer value between 0 and 9 (high).

If remove_gaps = True, positions with gaps (or whitespace) in all selected sequences are removed from the alignment.

The io argument is required since PIR files can contain empty sequences or ranges; in this case, the sequence or range is read from the corresponding PDB file.

If allow_alternates = True, and reading a 'PIR' file where '.' is used to force MODELLER to read the sequence range from the corresponding PDB file (see Section B.1), then the search for matches between the alignment sequence and PDB is made a little more flexible. Not only will an exact equivalence of one-letter codes be considered a match, but each residue's alternate (as defined by the STD column in 'modlib/restyp.lib') will also count as a match; for example, B (ASX) in the alignment will be considered a match for N (ASN) in the PDB, while G (GLY) in the alignment will match any non-standard residue in the PDB for which an explicit equivalence has not been defined (the DEFATM behavior in 'modlib/restyp.lib'). The alignment sequence will be modified to match the exact sequence from the PDB. This is useful if the alignment sequence is extracted from a database containing 'cleaned' sequences, *e.g.* that created by **SequenceDB.read()**.

For 'PIR' and 'FASTA' files, the end_of_file variable is set to 1 if MODELLER reached the end of the file during the read, or 0 otherwise.

This command can raise a FileFormatError if the alignment file format is invalid, or a SequenceMismatchError if a 'PIR' sequence does not match that read from PDB (when an empty range is given).

Example: examples/commands/read_alignment.py

```
# Example for: Alignment.append(), Alignment.write(),
#              Alignment.check()

# Read an alignment, write it out in the 'PAP' format, and
# check the alignment of the N-1 structures as well as the
# alignment of the N-th sequence with each of the N-1 structures.

from modeller import *

log.level(output=1, notes=1, warnings=1, errors=1, memory=0)
env = Environ()
env.io.atom_files_directory = ['../atom_files']

aln = Alignment(env)
aln.append(file='toxin.ali', align_codes='all')
aln.write(file='toxin.pap', alignment_format='PAP')
aln.write(file='toxin.fasta', alignment_format='FASTA')
aln.check()
```

6.16.5 Alignment.clear() — delete all sequences from the alignment

```
clear()
```

This deletes all of the sequences from the alignment. It is not exactly the same as deleting the alignment object and creating a new one, since any structural data already read in remains in the alignment object. This is useful if the sequences are reread and the structural information needs to be reused.

6.16.6 Alignment.read_one() — read sequences one by one from a file

```
read_one(file, remove_gaps=False, alignment_format='PIR', io=None, allow_alternates=False)
```

Output: `True` only if a sequence was read

This reads a single sequence from an open alignment file into the current alignment. This is useful, for example, when dealing with a very large database of sequences, which you do not want to read into memory in its entirety. The sequences can then be processed individually.

On exit, `True` is returned if a sequence was read. The read sequence is the only sequence in the final alignment (anything in the alignment before calling this method is erased). If the end of the file was reached without reading a sequence, `False` is returned.

Arguments are as for **Alignment.append()**. Note that only `'PIR'` or `'FASTA'` format files can be read with this command. file should be an open file handle (see **modfile.File()**). Since only a single sequence is read, if remove_gaps is `True`, **all** gaps in the sequence are removed, regardless of whether they are aligned with other sequences in the alignment file.

This command can raise a `FileFormatError` if the alignment file format is invalid, or a `SequenceMismatchError` if a `'PIR'` sequence does not match that read from PDB (when an empty range is given).

Example: examples/commands/alignment_read_one.py

```
# Example for: Alignment.read_one()

from modeller import *
env = Environ()

# Create an empty alignment
aln = Alignment(env)

# Open the input alignment file, and get a handle to it:
input = modfile.File('toxin.ali', 'r')
# Same for the output file:
output = modfile.File('toxin-filter.ali', 'w')

# Read sequences one by one from the file handle in PIR format:
while aln.read_one(input, alignment_format='PIR'):
    print("Read code %s" % aln[0].code)
    # Write only X-ray structures to the output file:
    if aln[0].prottyp == 'structureX':
        aln.write(output, alignment_format='FASTA')
```

```
# Explicitly close the files (not strictly necessary in this simple
# example, because they'll be closed at the end of the script anyway):
input.close()
output.close()
```

6.16.7 Alignment.check_structure_structure() — check template structure superpositions

`check_structure_structure(eqvdst=6.0, io=None)`

Output: n_exceed

This command checks the alignment of the template structures (all but the last entry in the alignment): For each pairwise superposition of the templates, it reports those equivalent pairs of C_α atoms that are more than eqvdst Å away from each other. Such pairs are almost certainly misaligned. The pairwise superpositions are done using the C_α atoms and the given alignment. The number of such pairs is returned.

Note that the target structures are actually changed by the superpositions carried out by this command. If you want to use these superpositions as a crude initial model for `AutoModel` model building (rather than setting AutoModel.initial_malign3d) please bear in mind that the later templates in your alignment are always fitted on the earlier templates. Thus, a more reliable initial model will be obtained if you list the higher coverage templates earlier in the knowns variable in your `AutoModel` script (*e.g.*, always list multimeric templates before monomers).

If you want to use the original non-superposed structures, either avoid calling this command, or delete and recreate the alignment object afterwards to force it to reread the structure files.

6.16.8 Alignment.check_sequence_structure() — check sequence/structure alignment for sanity

`check_sequence_structure(gapdist=8.0, io=None)`

Output: n_exceed

This command checks the alignment of the target sequence (the last entry in the alignment) with each of the templates: For all consecutive pairs of C_α atoms within each chain in the target, it calculates the distance between the two equivalent C_α atoms in each of the templates. If the distance is longer than gapdist Å, it is reported. In such a case, the alignment between the template and the target is almost certainly incorrect. The total number of exceeded pair distances is returned.

6.16.9 Alignment.check() — check alignment for modeling

`check(io=None)`

This command evaluates an alignment to be used for comparative modeling, by calling **Alignment.check_structure_structure()** and **Alignment.check_sequence_structure()**.

Example: examples/commands/read_alignment.py

```
# Example for: Alignment.append(), Alignment.write(),
#              Alignment.check()

# Read an alignment, write it out in the 'PAP' format, and
# check the alignment of the N-1 structures as well as the
# alignment of the N-th sequence with each of the N-1 structures.

from modeller import *

log.level(output=1, notes=1, warnings=1, errors=1, memory=0)
env = Environ()
env.io.atom_files_directory = ['../atom_files']

aln = Alignment(env)
aln.append(file='toxin.ali', align_codes='all')
aln.write(file='toxin.pap', alignment_format='PAP')
aln.write(file='toxin.fasta', alignment_format='FASTA')
aln.check()
```

6.16.10 Alignment.compare_with() — compare two alignments

`compare_with(aln)`

Output: Percent residue-residue equivalence

This command compares two pairwise alignments read by the **Alignment.append()** commands. The alignment of the first sequence with the second sequence in `aln` is evaluated with respect to the current alignment. The numbers are not symmetric; *i.e.*, they will change if the sequences or alignments are swapped. The output in the `log` file is self-explanatory. The percentage of equivalent residue-residue pairs in the two alignments is returned.

Example: examples/commands/compare_alignments.py

```
# Example for: Alignment.compare_with(), Alignment.append_model()

# Compare two alignments of two proteins each. In this case, the first
# alignment is a sequence-sequence alignment and the second alignment
# is a structure-structure alignment.

from modeller import *
log.level(1, 1, 1, 1, 0)
env = Environ()
env.io.atom_files_directory = ['../atom_files']

# Generate and save sequence-sequence alignment:
aln = Alignment(env)
for code in ('1fas', '2ctx'):
    mdl = Model(env, file=code)
    aln.append_model(mdl=mdl, align_codes=code, atom_files=code)
aln.align(gap_penalties_1d=(-600, -400))
aln.write(file='toxin-seq.ali')

# Generate and save structure-structure alignment:
```

```
aln.align3d(gap_penalties_3d=(0, 2.0))
aln.write(file='toxin-str.ali')

# Compare the two pairwise alignments:
aln  = Alignment(env, file='toxin-seq.ali', align_codes='all')
aln2 = Alignment(env, file='toxin-str.ali', align_codes='all')
aln.compare_with(aln2)
```

6.16.11 Alignment.append_model() — copy model sequence and coordinates to alignment

`append_model(mdl, align_codes, atom_files='')`

This command adds the sequence and coordinates of the given model, mdl, to the end of the current alignment.

You should additionally set align_codes and atom_files to the PDB ID and file name, respectively. This information is added to the alignment with the new sequence. (Alternatively, you can set this information later by setting Sequence.code and Sequence.atom_file.)

This command can raise a `ValueError` if the align code is too long (see Sequence.code.)

Example: examples/commands/aln_append_model.py

```
# This demonstrates one way to generate an initial alignment between two
# PDB sequences. It can later be edited by hand.

# Set Modeller environment (including search patch for Model.read())
from modeller import *
env = Environ()
env.io.atom_files_directory = [".", "../atom_files/"]

# Create a new empty alignment and model:
aln = Alignment(env)
mdl = Model(env)

# Read the whole 1fdn atom file
code='1fdn'
mdl.read(file=code, model_segment=('FIRST:@', 'END:'))

# Add the model sequence to the alignment
aln.append_model(mdl, align_codes=code, atom_files=code)

# Read 5fd1 atom file chain A from 1-63, and add to alignment
code='5fd1'
mdl.read(file=code, model_segment=('1:A', '63:A'))
aln.append_model(mdl, align_codes=code, atom_files=code)

# Align them by sequence
aln.malign(gap_penalties_1d=(-500, -300))
aln.write(file='fer1-seq.ali')

# Align them by structure
aln.malign3d(gap_penalties_3d=(0.0, 2.0))
```

```
# check the alignment for its suitability for modeling
aln.check()

aln.write(file='fer1.ali')
```

6.16.12 Alignment.append_sequence() — add a sequence from one-letter codes

`append_sequence(sequence, blank_single_chain=False, zero_width_break=False)`

This builds a new sequence from the provided one-letter codes, and adds it to the end of the alignment. You can also use '-' and '/' characters in this sequence to add gaps and chain breaks.

By default, the newly created chains are labeled `'A'`, `'B'`, `'C'` and so on. However, if only one chain is generated, and blank_single_chain is set `True`, it is given a blank chain ID.

Internally, a chain break is not a residue (it is essentially a signal to not construct a peptide bond between adjacent residues) and so has "zero width". However, the '/' character has a width of one just like a residue or a gap. To simplify alignment construction, if zero_width_break is `False` (the default), a gap is also added to the alignment when a chain break is encountered, so that other sequences will align. For example, after appending the two sequences `AC/G` and `ETPW`, `G` will align with `W` by default, or with `P` if zero_width_break is set to `True`.

Example: See **Model.build_sequence()** command.

6.16.13 Alignment.append_profile() — add profile sequences to the alignment

`append_profile(prf)`

This adds all the sequences from the given profile, prf, to the alignment. It is similar in operation to **Profile.to_alignment()**.

6.16.14 Alignment.write() — write sequences and/or their alignment

`write(file, alignment_format='PIR', alignment_features='INDICES CONSERVATION', align_block=0, align_alignment=False)`

This command writes the whole alignment to a text file.

file can be either a file name or a writeable file handle (see **modfile.File()**).

alignment_format selects the format to write the alignment in; see **Alignment.append()**.

The `'PAP'` format, which corresponds to a relatively nice looking alignment, has several additional formatting options that can be selected by the alignment_features variable. This scalar variable can contain any combination of the following keywords:

- `'INDICES'`, the alignment position indices;
- `'CONSERVATION'`, a star for each absolutely conserved position;
- `'ACCURACY'`, the alignment accuracy indices, scaled between 0–9, as calculated by **Alignment.consensus()**;
- `'HELIX'`, average content of helical residues for structures 1 – align_block at each position, 0 for 0% and 9 for 100%, as calculated by **Alignment.align2d()**.

- `'BETA'`, average content of β-strand residues for structures 1 – align_block at each position, 0 for 0% and 9 for 100%, as calculated ‘by **Alignment.align2d()**.
- `'ACCESSIBILITY'`, average relative sidechain buriedness for structures 1 – align_block, 0 for 0% (100% accessibility) and 9 for 100% (0% accessibility), as calculated by **Alignment.align2d()**;
- `'STRAIGHTNESS'`, average mainchain straightness structures 1 – align_block at each position 0 for 0% and 9 for 100%, as calculated by **Alignment.align2d()**.
- `'PRED_SS'`, predicted secondary structure (H,E,C)
- `'CONF_SS'`, confidence of predicted secondary structure (0(low) - 9(high))

Options `'HELIX'`, `'BETA'`, `'ACCESSIBILITY'`, and `'STRAIGHTNESS'` are valid only after executing command **Alignment.align2d()**, where the corresponding quantities are defined. They refer to the 3D profile defined for the first align_block structures (run **Alignment.align2d()** with fit = `False` to prepare these structural data without changing the input alignment). Similarly, the `'ACCURACY'` option is valid only after the **Alignment.consensus()** command. Options `'PRED_SS'` and `'CONF_SS'` are best exercised after reading in a ".PSS" file of secondary structure predictions. In the case of multiple sequences, it may be necessary to use the command **Sequence.transfer_res_prop()** first.

align_alignment and align_block are used to ensure correct indication of identical alignment positions, depending on whether sequences or two blocks of sequences were aligned: For sequences (align_alignment = `False` and align_block is ignored), a `'*'` indicating a conserved position is printed where all sequences have the same residue type. For blocks (align_alignment = `True` and align_block indicates the last sequence of the first block), a `'*'` is printed only where the two blocks have the same order of residue types (there has to be the same number of sequences in both blocks). The blocks option is useful when comparing two alignments, possibly aligned by the **Alignment.align()** command.

Example: See **Alignment.append()** command.

6.16.15 Alignment.edit() — edit overhangs in alignment

```
edit(overhang, edit_align_codes, base_align_codes, min_base_entries, by_chain=False, io=None)
```

This command edits the overhangs in the alignment.

edit_align_codes specifies the alignment codes for the alignment entries whose overhangs are to be cut; in addition, `all` or `last` can be used.

base_align_codes specifies the alignment codes for the alignment entries that are used to determine the extent of the overhangs to be cut from the edited entries; in addition, `all` or `rest` (relative to edit_align_codes) can be used.

The same entries can be cut and used for determining the base.

The base of the alignment is determined by the first and last alignment positions that have at least min_base_entries entries that started by that position, beginning from the first and last alignment positions, respectively.

The cuts are shortened by overhang residues respectively, so that reasonably short termini can be easily modeled *ab initio* if desired.

The io argument is used because the beginning and ending residue numbers for the `'structure'` entries in the alignment are renumbered automatically by reading the appropriate atom files.

The number of residues (not alignment positions) removed from the start and end of the first sequence in edit_align_codes is returned.

Normally, this procedure ignores chain breaks, removing overhangs only from the very start and end of the entire sequence. However, if by_chain is set to `True`, overhangs for every chain in the edit_align_codes sequence are removed. This mode only works for a single edited sequence, and only for a sequence that does not have structural information. In this case, a list of pairs is returned, one for each chain; each pair contains the number of residues removed from the start and end of the chain.

Example: examples/commands/edit_alignment.py

```
# Example for: Alignment.edit()

# Read an alignment, write it out in the 'PAP' format, with overhangs cut.

from modeller import *

log.level(1, 1, 1, 1, 0)
env = Environ()
env.io.atom_files_directory = ['.', '../atom_files']

aln = Alignment(env, file='overhang.ali', align_codes='all',
                alignment_format='PIR')

# Cut overhangs in the 1is4 sequence that are longer than 3 residues
# relative to the longest remaining entry in the alignment:
aln.edit(edit_align_codes='1is4', base_align_codes='rest',
         min_base_entries=1, overhang=3)
aln.write(file='overhang-1.pir', alignment_format='PIR')
aln.write(file='overhang-1.pap', alignment_format='PAP')
```

6.16.16 Alignment.describe() — describe proteins

`describe(io=None)`

This command outputs basic data about the proteins in the current alignment (*e.g.*as read in by **Alignment.append()**). This command is useful for preparation before comparative modeling because it summarizes disulfides, *cis*-prolines, charges, chain breaks, *etc.* Results which depend only on the amino acid sequences are still written out even if some atom files do not exist.

Example: examples/commands/describe.py

```
# Example for: Alignment.describe()

# Describe the sequences and structures in the alignment.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']
aln = Alignment(env, file='toxin.ali', align_codes=('2ctx', '2abx'))
aln.describe()
```

6.16.17 Alignment.id_table() — calculate percentage sequence identities

`id_table(matrix_file)`

This command calculates percentage residue identities for all pairs of sequences in the current alignment. The percentage residue identity is defined as the number of identical residues divided by the length of the shorter sequence.

In addition to the output in the `log` file, this routine creates file matrix_file, or writes to an already open file handle (see **modfile.File()**), with pairwise sequence distances that can be used directly as the input to the tree making programs of the PHYLIP package, such as KITSCH [Felsenstein, 1985], and also for the **Environ.dendrogram()** and **Environ.principal_components()** commands. A more general version of this command, which allows a user specified measure for residue–residue differences is **Alignment.compare_sequences()**.

Example: examples/commands/id_table.py

```
# Example for: Alignment.id_table(), Alignment.compare_sequences(),
#              misc.principal_components(), misc.dendrogram()

# Pairwise sequence identity between sequences in the alignment.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']
# Read all entries in this alignment:
aln = Alignment(env, file='toxin.ali')

# Access pairwise properties:
s1 = aln[0]
s2 = aln[1]
print("%s and %s have %d equivalences, and are %.2f%% identical" % \
      (s1, s2, s1.get_num_equiv(s2), s1.get_sequence_identity(s2)))

# Calculate pairwise sequence identities:
aln.id_table(matrix_file='toxin_id.mat')

# Calculate pairwise sequence similarities:
mdl = Model(env, file='2ctx', model_segment=('1:A', '71:A'))
aln.compare_sequences(mdl, rr_file='$(LIB)/as1.sim.mat', max_gaps_match=1,
                      matrix_file='toxin.mat', variability_file='toxin.var')
mdl.write(file='2ctx.var')

# Do principal components clustering using sequence similarities:
env.principal_components(matrix_file='toxin.mat', file='toxin.princ')

# Dendrogram in the log file:
env.dendrogram(matrix_file='toxin.mat', cluster_cut=-1.0)
```

6.16.18 Alignment.compare_sequences() — compare sequences in alignment

`compare_sequences(mdl, matrix_file, variability_file, max_gaps_match, rr_file='$LIB/as1.sim.mat')`

The pairwise similarity of sequences in the current alignment is evaluated using a user specified residue–residue scores file.

The residue–residue scores, including gap–residue, and gap–gap scores, are read from file rr_file. The sequence pair score is equal to the average pairwise residue–residue score for all alignment positions that have at most max_gaps_match gaps (1 by default). If the gap–residue and gap–gap scores are not defined in matrix_file, they are set to the worst and best residue–residue score, respectively. If matrix_file is a similarity matrix, it is converted into a distance matrix ($x' = -x + x_{max}$).

The comparison matrix is written in the PHYLIP format to file matrix_file.

The family variability as a function of alignment position is calculated as the RMS deviation of all residue – residue scores at a given position, but only for those pairs of residues that have at most max_gaps_match gaps (0, 1, or 2). The variability is written to file variability_file, as is the number of pairwise comparisons contributing to each positional variability. The variability, scaled by 0.1, is also written into the B_{iso} field of the model mdl, which must correspond to the first sequence in the alignment.

Example: See **Alignment.id_table()** command.

6.16.19 Alignment.align() — align two (blocks of) sequences

```
align(off_diagonal=100, local_alignment=False, matrix_offset=0.0, gap_penalties_1d=(-900.0,
-50.0), n_subopt=0, subopt_offset=0.0, weigh_sequences=False, smooth_prof_weight=10,
align_what='BLOCK', weights_type='SIMILAR', input_weights_file=None, output_weights_file=None,
rr_file='$(LIB)/as1.sim.mat', overhang=0, align_block=0, break_break_bonus=10000.0)
```

IMPORTANT NOTE: This command is **obsolete**, and is no longer maintained. It is strongly recommended that you use **Alignment.salign()** instead.

This command aligns two blocks of sequences.

The two blocks of sequences to be aligned are sequences 1 to align_block and align_block+1 to the last sequence. The sequences within the two blocks should already be aligned; their alignment does not change.

The command can do either the global (similar to [Needleman & Wunsch, 1970]; local_alignment = `False`) or local dynamic programming alignment (similar to [Smith & Waterman, 1981]; local_alignment = `True`).

For the global alignment, set overhang length overhang to more than 0 so that the corresponding number of residues at either of the four termini won't be penalized by any gap penalties (this makes it a pseudo local alignment).

To speed up the calculation, set off_diagonal to a number smaller than the shortest sequence length. The alignments matching residues i and j with $|i - j| >$ off_diagonal are not considered at all in the search for the best alignment.

The gap initiation and extension penalties are specified by gap_penalties_1d. The default values of -900 -50 for the `'as1.sim.mat'` similarity matrix were found to be optimal for pairwise alignments of sequences that share from 30% to 45% sequence identity (RS and AŠ, in preparation).

The residue type – residue type scores are read from file rr_file. The routine automatically determines whether it has to maximize similarity or minimize distance.

matrix_offset applies to local alignment only and influences its length. matrix_offset should be somewhere between the lowest and highest residue–residue scores. A smaller value of this parameter will make the local alignments shorter when distance is minimized, and longer when similarity is maximized. This works as follows: The recursively constructed dynamic programming comparison matrix is reset to 0 at position i, j when the current alignment score becomes larger (distance) or smaller (similarity) than matrix_offset. Note that this is equivalent to the usual shifting of the residue–residue scoring matrix in the sense that there are two combinations of gap_penalties_1d and matrix_offset values that will give exactly the same alignments irrespective of whether the matrix is actually offset (with 0 used to restart local alignments in dynamic programming) or the matrix is not offset but matrix_offset is used as the cutoff for restarting local alignments in dynamic programming. For the same reason, the matrix offset does not have any effect on the global alignments if the gap extension penalty is also shifted for half of the matrix offset.

The position–position score is an average residue–residue score for all possible pairwise comparisons between the two blocks ($n \times m$ comparisons are done, where n and m are the number of sequences in the two blocks, respectively). The first exception to this is when align_what is set to `'ALIGNMENT'`, in which case the two alignments defined by align_block are aligned; *i.e.*, the score is obtained by comparing only equivalent positions between the two alignment blocks (only n comparisons are done, where n is the number of sequences in each of the two blocks). This option is useful in combination with **Alignment.compare_with()** and

Alignment.write() for evaluation of various alignment parameters and methods. The second exception is when align_what is set to 'LAST', in which case only the last sequences in the two blocks are used to get the scores. In 'BLOCK', 'ALIGNMENT', and 'LAST' comparisons, penalty for a comparison of a gap with a residue during the calculation of the scoring matrix is obtained from the score file (gap–gap match should have a score of 0.0).

Only the 20 standard residue types, plus Asx (changes to Asn) and Glx (changes to Gln) are recognized. Every other unrecognized residue, except for a gap and a chain break, changes to Gly for comparison purposes.

When aligning two sequences containing multiple chains (*i.e.*, with align_what set to 'BLOCK' and align_block set to 1), this command will attempt to ensure that the chain breaks are aligned with each other (so that residues from one chain will not align with residues from another). This is done by adding a bonus score to positions in the dynamic programming matrix that correspond to aligning two chain breaks. This score can be adjusted by setting the break_break_bonus parameter, or the behavior can be disabled by setting it to zero. For other kinds of alignments, chain breaks are ignored.

Example: examples/commands/align.py

```
# Example for: Alignment.align()

# This will read two sequences, align them, and write the alignment
# to a file:

from modeller import *
log.verbose()
env = Environ()

aln = Alignment(env)
aln.append(file='toxin.ali', align_codes=('1fas', '2ctx'))

# The as1.sim.mat similarity matrix is used by default:
aln.align(gap_penalties_1d=(-600, -400))
aln.write(file='toxin-seq.ali')
```

6.16.20 Alignment.align2d() — align sequences with structures

```
align2d(overhang=0, align_block=0, rr_file='$(LIB)/as1.sim.mat', align_what='BLOCK',
off_diagonal=100, max_gap_length=999999, local_alignment=False, matrix_offset=0.0,
gap_penalties_1d=(-100.0, 0.0), gap_penalties_2d=(3.5, 3.5, 3.5, 0.2, 4.0, 6.5, 2.0, 0.0,
0.0), surftyp=1, fit=True, fix_offsets=(0.0, -1.0, -2.0, -3.0, -4.0), input_weights_file=None,
output_weights_file=None, n_subopt=0, subopt_offset=0.0, input_profile_file=None,
output_profile_file=None, weigh_sequences=False, smooth_prof_weight=10, weights_type='SIMILAR',
break_break_bonus=10000.0, io=None)
```

IMPORTANT NOTE: This command is **obsolete**, and is no longer maintained. It is strongly recommended that you use **Alignment.salign()** instead.

This command aligns a block of sequences (second block) with a block of structures (first block). It is the same as the **Alignment.align()** command except that a variable gap **opening** penalty is used. This gap penalty depends on the 3D structure of all sequences in block 1. The variable gap penalty can favor gaps in exposed regions, avoid gaps within secondary structure elements, favor gaps in curved parts of the mainchain, and minimize the distance between the two C_α positions spanning a gap. The **Alignment.align2d()** command is preferred for aligning a sequence with structure(s) in comparative modeling because it tends to place gaps in a better structural context. See Section A.1.2 for the dynamic programming algorithm that implements the variable gap penalty. gap_penalties_2d specifies parameters ω_H, ω_S, ω_B, ω_C, ω_D, d_o, γ, t and

$\omega_S C$. (Section A.1.2). The default gap penalties gap_penalties_1d and gap_penalties_2d as well as the rr_file substitution matrix were found to be optimal in pairwise alignments of structures and sequences sharing from 30% to 45% sequence identity [Madhusudhan *et al.*, 2006].

The linear gap penalty function for inserting a gap in block 1 of structures is: $g = f_1(H, S, B, C, SC)u + lv$ where u and v are the usual gap opening and extension penalties, l is gap length, and f_1 is a function that is at least 1, but can be larger to make gap opening more difficult in the following circumstances: between two consecutive (*i.e.*, $i, i+1$) helical positions, two consecutive β-strand positions, two consecutive buried positions, or two consecutive positions where the mainchain is locally straight. This function is $f_1 = 1 + [\omega_H H_i H_{i+1} + \omega_S S_i S_{i+1} + \omega_B B_i B_{i+1} + \omega_C C_i C_{i+1} + \omega_S CSC_i SC_{i+1}]$, H_i is the fraction of helical residues at position i in block 1, S_i is the fraction of β-strand residues at position i in block 1, B_i is the average relative sidechain buriedness of residues at position i in block 1, C_i is the average straightness of residues at position i in block 1, and SC_i is the structural conservedness at position i in block 1. See Section 6.6.32 for the definition of these features. The original straightness is modified here by assigning maximal straightness of 1 to all residues in a helix or a β-strand. The structural conservedness of the residues in block 1 are imported from an external source input_profile_file. The structural conservedness at a particular position gives the likelihood of the occurrence of a gap when structurally similar regions from all known protein structures are aligned structurally.

The linear gap penalty function for opening a gap in block 2 of sequences is: $g = f_2(H, S, B, C, D, SC)u + lv$ where f_2 is a function that is at least 1, but can be larger to make the gap opening in block 2 more difficult in the following circumstances: when the first gap position is aligned with a helical residue, a β-strand residue, a buried residue, extended mainchain, or when the whole gap in block 2 is spanned by two residues in block 1 that are far apart in space. This function is $f_2 = 1 + [\omega_H H_i + \omega_S S_i + \omega_B B_i + \omega_C C_i + \omega_D \sqrt{d - d_o} + \omega_S CSC_i]$. d is the distance between the two C_α atoms spanning the gap, averaged over all structures in block 1 and d_o is the distance that is small enough to correspond to no increase in the opening gap penalty (*e.g.*, 8.6Å).

To find the best alignment, this method backtracks through the dynamic programming matrix, effectively adding gaps up to max_gap_length. Thus, for optimum performance you may want to reduce this parameter from its default value.

Other parameters are described in **Alignment.align()**.

When fit is False, no alignment is done and the routine returns only the average structural information, which can be written out by the **Alignment.write()** command.

Example: examples/commands/align2d.py

```
# Demonstrating ALIGN2D, aligning with variable gap penalty

from modeller import *
log.verbose()
env = Environ()

env.libs.topology.read('$(LIB)/top_heav.lib')
env.io.atom_files_directory = ['../atom_files']

# Read aligned structure(s):
aln = Alignment(env)
aln.append(file='toxin.ali', align_codes='2ctx')
aln_block = len(aln)

# Read aligned sequence(s):
aln.append(file='toxin.ali', align_codes='2nbt')

# Structure sensitive variable gap penalty sequence-sequence alignment:
aln.align2d(overhang=0, gap_penalties_1d=(-100, 0),
            gap_penalties_2d=(3.5, 3.5, 3.5, 0.2, 4.0, 6.5, 2.0, 0., 0.),
            align_block=aln_block)
```

```
aln.write(file='align2d.ali', alignment_format='PIR')
aln.write(file='align2d.pap', alignment_format='PAP',
          alignment_features='INDICES HELIX BETA STRAIGHTNESS ' + \
                             'ACCESSIBILITY CONSERVATION')
aln.check()

# Color the first template structure according to gaps in alignment:
aln = Alignment(env)
aln.append(file='align2d.ali', align_codes=('2ctx', '2nbt'),
           alignment_format='PIR', remove_gaps=True)
mdl = Model(env)
mdl.read(file=aln['2ctx'].atom_file,
         model_segment=aln['2ctx'].range)
mdl.color(aln=aln)
mdl.write(file='2ctx.aln.pdb')

# Color the first template structure according to secondary structure:
mdl.write_data(file='2ctx', output='SSM')
mdl.write(file='2ctx.ssm.pdb')

# Superpose the target structure onto the first template:
mdl2 = Model(env)
mdl2.read(file=aln['2nbt'].atom_file,
          model_segment=aln['2nbt'].range)
sel = Selection(mdl).only_atom_types('CA')
sel.superpose(mdl2, aln)
mdl2.write(file='2nbt.fit.pdb')
```

6.16.21 Alignment.malign() — align two or more sequences

`malign(rr_file='$(LIB)/as1.sim.mat', off_diagonal=100, local_alignment=False, matrix_offset=0.0, overhang=0, align_block=0, gap_penalties_1d=(-900.0, -50.0))`

IMPORTANT NOTE: This command is **obsolete**, and is no longer maintained. It is strongly recommended that you use **Alignment.salign()** instead.

This command performs a multiple sequence alignment. The sequences to be aligned are the sequences in the current alignment arrays. The command uses the dynamic programming method for the best sequence alignment, given the gap initiation and extension penalties specified by gap_penalties_1d, and residue type weights read from file rr_file. See command **Alignment.align()** for more information.

The algorithm for the multiple alignment is as follows. First, sequence 2 is aligned with sequence 1 (*i.e.*, block of sequences from 1–align_block). Next, sequence 3 is aligned with an average of the aligned sequences 1 and 2; *i.e.*, the weight matrix is an average of the weights 1–3 and 2–3. For this averaging, the gap–residue and gap–gap weights are obtained from the residue–residue weight matrix file, not from gap penalties. If the corresponding weights are not in the file, they are set to the worst and best residue–residue score, respectively.

See instructions for **Alignment.align()** for more details.

Example: examples/commands/malign.py

```
# Example for: Alignment.malign()

# This will read all sequences from a file, align them, and write
# the alignment to a new file:

from modeller import *

env = Environ()

aln = Alignment(env, file='toxin.ali', align_codes='all')
aln.malign(gap_penalties_1d=(-600, -400))
aln.write(file='toxin-seq.pap', alignment_format='PAP')
```

6.16.22 Alignment.consensus() — consensus sequence alignment

```
consensus(align_block=0, gap_penalties_1d=(-900.0, -50.0), weigh_sequences=False,
input_weights_file=None, output_weights_file=None, weights_type='SIMILAR',
smooth_prof_weight=10)
```

This command is similar to **Alignment.align()** except that a consensus alignment of two blocks of sequences is produced. A consensus alignment is obtained from a consensus similarity matrix using the specified gap penalties and the global dynamic programming method. The consensus similarity matrix is obtained by aligning the two blocks of sequences many times with different parameters and methods and counting how many times each pair was aligned. This command is still experimental and no detailed description is given at this time.

This command also produces the alignment accuracy that can be printed out by the **Alignment.write()** command in the `'PAP'` format (0 inaccurate, 9 accurate). If the gap initiation penalty is 0, the gap extension penalty of say 0.4 means that only those positions will be equivalenced that were aligned in at least 80% of the individual alignments (*i.e.*, 2 times 0.40).

Example: examples/commands/align_consensus.py

```
# Example for: Alignment.consensus()

# This will read 2 sequences and prepare a consensus alignment
# from many different pairwise alignments.

from modeller import *
env = Environ()

aln = Alignment(env)
aln.append(file='toxin.ali', align_codes=('2ctx', '2abx'))
aln.consensus(gap_penalties_1d=(0, 0.4), align_block=1)
aln.write(file='toxin-seq.pap', alignment_format='PAP')
```

6.16.23 Alignment.compare_structures() — compare 3D structures given alignment

```
compare_structures(compare_mode=3, fit=True, fit_atoms='CA', matrix_file='family.mat',
output='LONG', asgl_output=False, refine_local=True, rms_cutoffs=(3.5, 3.5, 60.0, 60.0, 15.0,
60.0, 60.0, 60.0, 60.0, 60.0, 60.0), varatom='CA', edat=None, io=None)
```

This command compares the structures in the given alignment. It does not make an alignment, but it calculates the RMS and DRMS deviations between atomic positions and distances, and class differences between the mainchain and sidechain dihedral angles. In contrast to the **Selection.superpose()** command, **Alignment.compare_structures()** works with a multiple alignment and it writes more information about the pairwise comparisons.

output selects short (`'SHORT'`) or long (`'LONG'`) form of output to the `log` file. If it contains word `'RMS'` or `'DRMS'` it also outputs the RMS or DRMS deviation matrix to file matrix_file. This file can be used with the PHYLIP program or with the **Environ.dendrogram()** or **Environ.principal_components()** commands of MODELLER to calculate a clustering of the structures.

compare_mode selects the form of the positional variability calculated for each position along the sequence:

1, for true RMS deviation over all proteins that have a residue at the current position. This does not make any sense for periodic quantities like dihedral angles.

2, for the average absolute distance over all pairs of residues that have a residue at the current position.

3, the same as 2 except that average distance, not its absolute value is used (convenient for comparison of 2 structures to get the $\pm$ sign of the changes for dihedral angles and distances).

rms_cutoffs specifies cutoff values for calculation of the position, distance, and dihedral angle RMS deviations for pairwise overall comparisons. If difference between two equivalent points is larger than cutoff it is not included in the RMS sum. The order of cutoffs in this vector is: atomic position, intra-molecular distance, α, Φ, Ψ, ω, χ_1, χ_2, χ_3, χ_4, and χ_5 (there are 5 dihedrals in a disulfide bridge), where α is the virtual C_α dihedral angle between four consecutive C_α atoms. These cutoffs do not affect positional variability calculations.

fit_atoms string specifies all the atom types (including possibly a generic `'ALL'`) to be fitted in the least-squares superposition. These atom types are used in the least-squares superposition, and in calculation of the position and distance RMS deviations.

varatom specifies the atom type that is used for getting the average structure and RMS deviation at each alignment position in the ASGL output file `'posdif.asgl'`. This ASGL file contains the positional variability of the selected atom type in the family of compared proteins. The ASGL output files can then be used with ASGL scripts `'posdif'` and `'dih'` to produce POSTSCRIPT plots of the corresponding variabilities at each alignment position. asgl_output has to be `True` to obtain the ASGL output files.

If fit = `True`, a least-squares superposition is done before the comparisons; otherwise, the orientation of the molecules in the input atom files is used.

Example: See **Alignment.malign3d()** command.

6.16.24 Alignment.align3d() — align two structures

```
align3d(off_diagonal=100, overhang=0, local_alignment=False, matrix_offset=0.0,
gap_penalties_3d=(0.0, 1.75), fit=True, fit_atoms='CA', align3d_trf=False, output='LONG',
align3d_repeat=False, io=None)
```

IMPORTANT NOTE: This command is **obsolete**, and is no longer maintained. It is strongly recommended that you use **Alignment.salign()** instead.

This command uses the current alignment as the starting point for an iterative least-squares superposition of two 3D structures. This results in a new pairwise structural alignment. A good initial alignment may be obtained by sequence alignment (**Alignment.align()**). For superpositions, only one atom per residue is used, as specified by fit_atoms.

The alignment algorithm is as follows. First, structure 2 is least-squares fit on structure 1 using all the equivalent residue positions in the initial alignment that have the specified atom type. Next, the residue–residue distance matrix is obtained by calculating Euclidean distances between all pairs of selected atoms from the two structures. The alignment of the two structures is then obtained by the standard dynamic programming optimization based on the residue–residue distance matrix.

gap_penalties_3d[0] is a gap creation penalty (usually 0), and gap_penalties_3d[1] is a gap extension penalty, say 1.75. This procedure identifies pairs of residues as equivalent when they have their selected atoms at most 2 times gap_penalties_3d[1] angstroms apart in the current orientation (this is so when the gap initiation penalty is 0). The reason is that an equivalence costs the distance between the two residues while an alternative, the gap–residue and residue-gap matches, costs twice the gap extension penalty.

From the dynamic programming run, a new alignment is obtained. Thus, structure 2 can be fitted onto structure 1 again, using this new alignment, and the whole cycle is repeated until there is no change in the number of equivalent positions and until the difference in the rotation matrices for the last two superpositions is very small. At the end, the framework, that is the alignment positions without gaps, is written to the `log` file.

If fit is `False`, no alignment is done.

If output contains `'SHORT'`, only the best alignment and its summary are displayed. If output contains `'LONG'`, summaries are displayed for all initial alignments in each framework cycle. If output contains `'VERY_LONG'`, all alignments are displayed.

If align3d_trf is `True`, the weights in the weight matrix are modified distances [Subbiah *et al.*, 1993].

If align3d_repeat is `True`, three additional initial alignments are tried and the one resulting in the largest number of equivalent positions is selected.

Example: examples/commands/align3d.py

```
# Example for: Alignment.align3d(), Selection.superpose()

# This will align 3D structures of two proteins:

from modeller import *
log.verbose()
env = Environ()
env.io.atom_files_directory = ['../atom_files']

# First example: read sequences from a sequence file:
aln = Alignment(env)
aln.append(file='toxin.ali', align_codes=['1fas', '2ctx'])
aln.align(gap_penalties_1d=[-600, -400])
aln.align3d(gap_penalties_3d=[0, 4.0])
aln.write(file='toxin-str.ali')

# Second example: read sequences from PDB files to eliminate the
# need for the toxin.ali sequence file:
mdl = Model(env)
aln = Alignment(env)
for code in ['1fas', '2ctx']:
    mdl.read(file=code)
    aln.append_model(mdl, align_codes=code, atom_files=code)
aln.align(gap_penalties_1d=(-600, -400))
aln.align3d(gap_penalties_3d=(0, 2.0))
aln.write(file='toxin-str.ali')

# And now superpose the two structures using current alignment to get
# various RMS's:
mdl = Model(env, file='1fas')
atmsel = Selection(mdl).only_atom_types('CA')
mdl2 = Model(env, file='2ctx')
atmsel.superpose(mdl2, aln)
```

6.16.25 Alignment.malign3d() — align two or more structures

`malign3d(off_diagonal=100, overhang=0, local_alignment=False, matrix_offset=0.0, gap_penalties_3d=(0.0, 1.75), fit=True, fit_atoms='CA', output='LONG', write_whole_pdb=True, current_directory=True, write_fit=False, edit_file_ext=('.pdb', '_fit.pdb'), io=None)`

IMPORTANT NOTE: This command is **obsolete**, and is no longer maintained. It is strongly recommended that you use **Alignment.salign()** instead.

This command uses the current alignment as the starting point for an iterative least-squares superposition of two or more 3D structures. This results in a new multiple structural alignment. A good initial alignment may be obtained by sequence alignment (**Alignment.malign()**). For superpositions, only one atom per residue is used, as specified by fit_atoms. The resulting alignment can be written to a file with the **Alignment.write()** command. The multiply superposed coordinates remain in memory and can be used with such commands as **Model.transfer_xyz()** if Sequence.atom_file is not changed in the meantime. It is best to use the structure that overlaps most with all the other structures as the first protein in the alignment. This may prevent an error exit due to too few equivalent positions during framework construction.

The alignment algorithm is as follows. There are several cycles, each of which consists of an update of a framework and a calculation of a new alignment; the new alignment is based on the superposition of the structures onto the latest framework. The framework in each cycle is obtained as follows. The initial framework consists of the atoms in structure 1 that correspond to fit_atoms. If there is no specified atom types in any of the residues at a given position, the coordinates for this framework position are approximated by the neighboring coordinates. Next, all other structures are fit to this framework. The final framework for the current cycle is then obtained as an average of all the structures, in their fitted orientations, but only for residue positions that are common to all of them, given the current alignment. Another result is that all the structures are now superposed on this framework. Note that the alignment has not been changed yet. Next, the multiple alignment itself is re-derived in $N-1$ dynamic programming runs, where N is the number of structures. This is done as follows. First, structure 2 is aligned with structure 1, using the inter-molecular atom–atom distance matrix, for all atoms of the selected type, as the weight matrix for the dynamic programming run. Next, structure 3 is aligned with an average of structures 1 and 2 using the same dynamic programming technique. Structure 4 is then aligned with an average of structures 1–3, and so on. Averages of structures *i*–*j* are calculated for all alignment positions where there is at least one residue in any of the structures *i*–*j* (this is different from a framework which requires that residues from all structures be present). Note that in this step, residues out of the current framework may get aligned and the current framework residues may get unaligned. Thus, after the series of $N-1$ dynamic programming runs, a new multiple alignment is obtained. This is then used in the next cycle to obtain the next framework and the next alignment. The cycles are repeated until there is no change in the number of equivalent positions. This procedure is best viewed as a way to determine the framework regions, not the whole alignment. The results from this command are expected to be similar to the output of program MNYFIT [Sutcliffe *et al.*, 1987].

gap_penalties_3d[0] is a gap creation penalty (usually 0), and gap_penalties_3d[1] is a gap extension penalty, say 1.75. This procedure identifies pairs of positions as equivalent when they have their selected atoms at most 2 times gap_penalties_3d[1] angstroms apart in the current superposition (this is so when the gap initiation penalty is 0), as described for the **Alignment.align3d()** command.

Argument output can contain the following values:

- `'SHORT'`, only the final framework is written to the `log` file.
- `'LONG'`, the framework after the alignment stage in each cycle is written to the `log` file.
- `'VERY_LONG'`, the framework from the framework stage in each cycle is also written to the log.

If write_fit is `True`, the fitted atom files are written out in their final fitted orientations. To construct the filenames, first the file extension in edit_file_ext[0] is removed (if present), and then the extension in edit_file_ext[1] is added (if not already present). By default this creates files with a `_fit` extension.

If current_directory is `True`, the fitted atom files will go to the current directory. Otherwise, the output will be in the directory with the original files[7].

[7]This won't work in combination with write_whole_pdb = `False` for structures that were added to the alignment with

If write_whole_pdb is True, the whole PDB files are written out[8]; otherwise only the parts corresponding to the aligned sequences are output.

If fit is False, the initial alignment is not changed. This is useful when all the structures have to be superimposed with the initial alignment (fit = False and write_fit = True).

Example: examples/commands/malign3d.py

```
# Example for: Alignment.malign3d(), Alignment.compare_structures()

# This will read all sequences from a sequence file, multiply align
# their 3D structures, and then also compare them using this alignment.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']

aln = Alignment(env, file='toxin.ali', align_codes='all')
aln.malign(gap_penalties_1d=(-600, -400))
aln.malign3d(gap_penalties_3d=(0, 2.0), write_fit=True, write_whole_pdb=False)
aln.write(file='toxin-str.pap', alignment_format='PAP')

# Make two comparisons: no cutoffs, and 3.5A/60 degree cutoffs for RMS, DRMS,
# and dihedral angle comparisons:
aln.compare_structures(rms_cutoffs=[999]*11)
aln.compare_structures(rms_cutoffs=(3.5, 3.5, 60, 60, 60, 60, 60, 60, 60,
                                    60, 60))
```

6.16.26 Alignment.salign() — align two or more sequences/structures of proteins

```
salign(residue_type2='REGULAR', no_ter=False, overhang=0, off_diagonal=100, matrix_offset=0.0,
gap_penalties_1d=(-900.0, -50.0), gap_penalties_2d=(3.5, 3.5, 3.5, 0.2, 4.0, 6.5, 2.0,
0.0, 0.0), gap_penalties_3d=(0.0, 1.75), feature_weights=(1.0, 0.0, 0.0, 0.0, 0.0,
0.0), rms_cutoff=3.5, fit=True, surftyp=1, fit_on_first=False, gap_function=False,
align_block=0, max_gap_length=999999, align_what='BLOCK', input_weights_file=None,
output_weights_file=None, weigh_sequences=False, smooth_prof_weight=10, fix_offsets=(0.0,
-1.0, -2.0, -3.0, -4.0), substitution=False, comparison_type='MAT', matrix_comparison='CC',
alignment_type='PROGRESSIVE', edit_file_ext=('.pdb', '_fit.pdb'), weights_type='SIMILAR',
similarity_flag=False, bkgrnd_prblty_file='$(LIB)/blosum62_bkgrnd.prob', ext_tree_file=None,
dendrogram_file='', matrix_scaling_factor=0.0069, auto_overhang=False, overhang_factor=0.4,
overhang_auto_limit=60, local_alignment=False, improve_alignment=True, fit_atoms='CA',
output='', write_whole_pdb=True, current_directory=True, write_fit=False, fit_pdbnam=True,
rr_file='$(LIB)/as1.sim.mat', n_subopt=0, subopt_offset=0.0, align3d_trf=False,
normalize_pp_scores=False, gap_gap_score=0.0, gap_residue_score=0.0, nsegm=2,
matrix_offset_3d=-0.1, break_break_bonus=10000.0, io=None)
```

Output: SalignData object

Alignment.append_model(), since such inputs may not have an atom file to extract the directory from. In this case the outputs will end up in the current directory.

[8]Any structures that were added with **Alignment.append_model()** will need to have their corresponding atom files available, so that the originals can be reread at this point.

This command is a general dynamic programming based alignment procedure for aligning sequences, structures or a combination of the two. It is loosely based on the program COMPARER [Šali & Blundell, 1990]. SALIGN can be used to generate multiple protein structures/sequences alignments or to align two blocks of sequences/structures that are in memory.

See also Section 6.34 for utility scripts to simplify the high-level usage of SALIGN.

Please note that the method is still in development, and has not yet been fully benchmarked. As with any other alignment method, generated alignments should be assessed for quality.

Broadly classifying, three different types of protein alignment categories are tackled by this command:

1. Multiple structure alignments
2. Aligning a structure block to a sequence block
3. Multiple and pair-wise protein sequence alignment

The command incorporates the functionality of several old MODELLER commands (**Alignment.align()**, **Alignment.align2d()**, **Alignment.malign()**, **Alignment.align3d()**, and **Alignment.malign3d()**). Some of the examples below illustrate the equivalent script files to replace the old alignment commands with **Alignment.salign()**.

In addition to these, this command has several new alignment features including profile-profile sequence alignments and a dendrogram based multiple sequence/structure alignment among others.

All pair-wise alignments make use of local or global dynamic programming. A switch from one to another can be effected by setting local_alignment to `True` or `False`. The dynamic programming can be carried out using affine gap penalties (as previously used in **Alignment.align()**, by setting gap_function to `False`) or an environment dependent gap penalty function (as used in **Alignment.align2d()**, by setting gap_function to `True`). (Please note that the default gap_penalties_1d parameters are optimal for the affine gap penalty; see the align2d examples for reasonable parameters if you wish to use the environment dependent gap penalty.) All arguments that associated to the **Alignment.align()** and **Alignment.align2d()** commands apply.

If at least one of the blocks in a pairwise alignment consists of structures, dynamic programming can be performed using structure dependent gap penalties.

On successful completion, an SalignData object is returned, from which some of the calculated data can be queried. For example, if you save this in a variable 'r', the following data are available:

- `r.aln_score`; the alignment score
- `r.qscorepct`; the quality score (percentage) if output contains `'QUALITY'`

Features of proteins used for alignment

Central to the dynamic programming algorithm is the weight matrix. In SALIGN, this matrix is constructed by weighting the contribution from six features of protein structure and sequence:

Feature 1 is the residue type. $W^1_{i,j}$ is obtained from the residue type – residue type dissimilarity matrix, specified in the file rr_file. $W^1_{i,j}$ dissimilarity score for positions i and j in the two compared sub-alignments is the average dissimilarity score for a comparison of all residues in one sub-alignment with all residues in the other sub-alignment (note that gaps are ignored here). Should only feature weight 1 be non-zero, the user has an option of considering residue-residue similarity scores instead of distance scores by setting similarity_flag to `True`. (The other features are distance features, and so if their weights are non-zero, similarity_flag must be turned off, which is the default.)

Feature 2 is the inter-molecular distance for a pair of residues (unless align3d_trf is `True`: see **Alignment.align3d()**). Only one atom per residue is of course selected, as specified by fit_atoms (*e.g.*, C_α, although we should also allow for C_β in the future, which requires an intervention for Gly). This 'position' feature is complicated because it depends on the relative orientation of the structures corresponding to the two compared alignments. $W^2_{i,j}$ is the Euclidean distance between the compared positions i and j in the two compared sub-alignments that are already optimally aligned and superposed based on their coordinates alone. This optimal alignment is obtained by an iterative procedure as follows (the same as in **Alignment.align3d()**). The average structures for both sub-alignments are calculated for all sub-alignment positions with at least one defined selected atom. This calculation is straightforward because the structures within the two sub-alignments are already superposed with each other (see below). Then, the distance matrix for dynamic programming with affine gap penalties is calculated as the matrix of Euclidean distances between the two averages. The dynamic programming results into a new alignment, dependent also on the gap initiation and extension penalties gap_penalties_3d (a reasonable setting is (0, 3)). gap_penalties_3d[0] is a gap creation penalty (usually 0), and gap_penalties_3d[1] is a gap extension penalty, say 3. When the gap initiation penalty is 0, pairs of positions are identified as equivalent when they have their selected atoms at most 2 times gap_penalties_3d[1] angstroms apart in the current superposition, as described for the **Alignment.align3d()** command. The new alignment is then used to generate the new superposition of the two averages, and the iteration of the distance matrix calculation, alignment and superposition is repeated until there are no changes in the number of equivalent positions and in the rotation matrix relating the two averages.

The values of both improve_alignment and fit are used in the calculation of the position feature. That is, the initial alignment and the orientation of the coordinates can be selected not to change at all during the calculation of the inter-molecular distance matrix.

When the calculation of the inter-molecular distance matrix is finished, all the structures in the second sub-alignment are rotated and translated following the optimal rotation and translation of the second average on the first average. These superpositions prepare the individual structures for the next of the $n - 1$ stages of the progressive multiple alignment, and also orient all the structures for writing out to atom files with a `'_fit.pdb'` extension if write_fit = `True`. If fit_pdbnam = `False`, the PDB filenames in the output alignment file will not have the `'_fit.pdb'` extensions. Thus, feature 2 needs to be selected by feature_weight[2] > 0 if you wish to write out the structures superposed according to the tree-following procedure; also, fit_on_first must be `False`, otherwise the structures are written out superposed on the first structure according to the final alignment (see also below).

The alignment produced within the routine that calculates W^2 does not generally correspond to the alignment calculated based on W. Therefore, the multiply superposed structures are not necessarily superposed based on the final multiple alignment produced by **Alignment.salign()**. If you wish such a superposition, you can use **Alignment.malign3d()** with fit = `False` and write_fit = `True` (the meaning of fit is different between **Alignment.salign()** and **Alignment.malign3d()**).

Unless the position feature is selected, the initial alignment does not matter. If the position feature is selected, a good starting alignment is a multiple sequence alignment, obtained either by **Alignment.malign()** or by **Alignment.salign()** used without the position feature (the initial alignment can also be prepared using the position feature). If the position feature is used, each pair of structures needs to have at least 3 aligned residues at all points during the alignment.

There are several possibilities as to the final orientation of the input coordinates. If fit_on_first is `True`, all the coordinate sets are superposed on the first structure, using the final multi-feature multiple alignment. If fit_on_first is `False`, and position feature was used, and fit was `True`, the coordinates will be superposed in the progressive manner guided by the tree, by the routine that calculates the inter-molecular distance matrices; this superposition is based only on the positions of the selected atoms (feature 2), not on other features such as residue type, secondary, structure, *etc.* If improve_alignment is `False`, it does not make much sense to have fit = `True` (use fit_on_first = `True`).

For local alignments, the matrix offset variable is matrix_offset_3d.

Feature 3 is the fractional sidechain accessibility. The pair-wise residue–residue dissimilarity is calculated by classifying residues into the buried ($< 15\%$), semi-exposed, and exposed classes ($> 30\%$). The dissimilarity is 0 for equal classes or if the absolute difference in the accessibility is less than 5%, 1 for neighboring classes and 2 for the buried–exposed match. The position–position dissimilarity is the average residue–residue dissimilarity for comparing all residues from one group to all residues in the other group (gaps are ignored).

Feature 4 is the secondary structure type, distinguishing between helix, strand, and other. The pair-wise residue–residue dissimilarity is 0 for equal classes, 1 for 'helix' or 'strand' matched to 'other', and 2 for 'helix' matched to 'strand'. Position–position dissimilarity is calculated in the same way as for feature 3.

Feature 5 is the local conformation. A pair-wise residue–residue score is DRMSD between the selected atoms (fit_atoms) from the segments of (2*nsegm + 1) residues centered on the two matched residues. Position–position dissimilarity is calculated in the same way as for feature 3.

Feature 6 is a user specified feature for which a external matrix (in MODELLER matrix format; see the substitution matrices in the `modlib` directory for examples) has to be specified using input_weights_file. The user can input either a similarity matrix (weights_type = `SIMILAR`) or a distance matrix (weights_type = `DISTANCE`).

Alignment of protein sequences

- **Multiple protein sequence alignment**
  Aligning multiple sequences is similar to aligning multiple structures, the difference being that for sequence alignments only the first feature weight can be non-zero (the other features require coordinates). This is the default for feature_weights.

  Example: examples/salign/salign_multiple_seq.py
```
# Illustrates the SALIGN multiple sequence alignment
from modeller import *

log.verbose()
env = Environ()
env.io.atom_files_directory = ['.', '../atom_files']

aln = Alignment(env, file='malign_in.ali')

aln.salign(overhang=30, gap_penalties_1d=(-450, -50),
           alignment_type='tree', output='ALIGNMENT')

aln.write(file='malign.ali', alignment_format='PIR')
```

- **Alignment of two sequence blocks**
  Two blocks of sequences can be aligned using the information contained within each of the multiple sequence blocks [Martí-Renom *et al.*, 2004]. Pairs of sequence blocks are aligned using SALIGN the same way in which **Alignment.align()** aligned sequence blocks; to align two blocks, simply proceed as normal, but set align_block to the number of sequences in the first block (the rest of the sequences are placed in the second block) and set alignment_what to `BLOCK`. Also, since this kind of alignment is effected only between two blocks, alignment_type is set to `PAIRWISE`.
- **Alignment of protein sequences by their profiles**
  As for pairwise alignment of sequence blocks, above, two blocks can be aligned by their profiles. align_block demarcates the end of the first block and align_what is set to `PROFILE` indicating that the blocks will be aligned using their profiles, and alignment_type is set to `PAIRWISE`. The weight matrix for dynamic programming is created by comparing the sequence information in the two blocks. Two kinds of comparisons can be performed:
  1. A correlation coefficient of the variation of the 20 amino acids at each position (comparison_type is set to `PSSM`).
  2. Comparing the residue substitution matrices implied at each position of the two blocks (comparison_type is set to `MAT`).

  Matrix comparisons are of three types: taking the maximum, average or correlation coefficient of residue-residue substitution at the aligned positions (matrix_comparison set to `MAX`, `AVE` or `CC`).

  Profile comparisons are done in similarity space rather than distance space, so similarity_flag should be set to `True`. They will also currently only work with feature 1 (since feature 1 is the only feature which works both in similarity and in distance space) - the weights for all other features must be set to zero.

  Example: examples/salign/salign_profile_profile.py
```
# profile-profile alignment using salign
from modeller import *
```

```
log.level(1, 0, 1, 1, 1)
env = Environ()

aln = Alignment(env, file='mega_prune.faa', alignment_format='FASTA')

aln.salign(rr_file='${LIB}/blosum62.sim.mat',
           gap_penalties_1d=(-500, 0), output='',
           align_block=15,   # no. of seqs. in first MSA
           align_what='PROFILE',
           alignment_type='PAIRWISE',
           comparison_type='PSSM',  # or 'MAT' (Caution: Method NOT benchmarked
                                    # for 'MAT')
           similarity_flag=True,    # The score matrix is not rescaled
           substitution=True,       # The BLOSUM62 substitution values are
                                    # multiplied to the corr. coef.
           #output_weights_file='test.mtx', # optional, to write weight matrix
           smooth_prof_weight=10.0) # For mixing data with priors

#write out aligned profiles (MSA)
aln.write(file='salign.ali', alignment_format='PIR')

# Make a pairwise alignment of two sequences
aln = Alignment(env, file='salign.ali', alignment_format='PIR',
                align_codes=('12asA', '1b8aA'))
aln.write(file='salign_pair.ali', alignment_format='PIR')
aln.write(file='salign_pair.pap', alignment_format='PAP')
```

Alignment of protein structures with sequences

As stated earlier, all **Alignment.align()** and **Alignment.align2d()** related commands apply to **Alignment.salign()** too. The example below is a **Alignment.salign()** equivalent of **Alignment.align2d()** (and **Alignment.align()**). For a description of the gap_penalties_2d see the section on **Alignment.align2d()**.

Example: examples/salign/salign_align2d.py

```
# align2d/align using salign

# parameters to be input by the user
# 1.  gap_penalties_1d
# 2.  gap_penalties_2d
# 3.  input alignment file

from modeller import *
log.verbose()
env = Environ()
env.io.atom_files_directory = ['../atom_files']

aln = Alignment(env, file='align2d_in.ali', align_codes='all')
aln.salign(rr_file='$(LIB)/as1.sim.mat',  # Substitution matrix used
           output='',
           max_gap_length=20,
           gap_function=True,              # If False then align2d not done
           feature_weights=(1., 0., 0., 0., 0., 0.),
           gap_penalties_1d=(-100, 0),
           gap_penalties_2d=(3.5, 3.5, 3.5, 0.2, 4.0, 6.5, 2.0, 0.0, 0.0),
           # d.p. score matrix
           #output_weights_file='salign.mtx'
           similarity_flag=True)   # Ensuring that the dynamic programming
                                   # matrix is not scaled to a difference matrix
aln.write(file='align2d.ali', alignment_format='PIR')
aln.write(file='align2d.pap', alignment_format='PAP')
```

Caution: The values of gap_penalties_2d have been optimized for similarity matrices. If using a distance matrix, you will need to derive new optimized values.

Alignment of protein structures

Structure alignments can make use of all the 5 structure/sequence features as well as the 6th user provided feature matrix. Pairwise alignments of structures can make use of the constant gap penalties or the environment dependent gap penalties. Multiple structure alignments are constructed from pairwise structure alignments.

This section describes the use of SALIGN to produce a single alignment of multiple structures. If the best output alignment is desired, it is recommended to run SALIGN in an iterative fashion, to determine the best parameter values. A utility script is provided for this purpose - see **iterative_structural_align()**.

- **Pairwise protein structure alignment**
  For optimal pairwise alignments it is suggested to call SALIGN multiple times, typically 2-3 times. The first SALIGN call will give an initial alignment which is refined in the subsequent calls. Usually, feature 2 is made non-zero only during the 'refinement' stage as rigid body refinement is done at the position implied by the alignment in memory.
- **Tree Multiple Structure Alignments**
  When alignment_type is set to `tree`, a dendrogram of the n proteins in memory is calculated using the selected features. The multiple alignment is then a progression of $n-1$ pairwise alignments of the growing sub-alignments. A sub-alignment is an alignment of $< n$ proteins. The pairwise alignment of two sub-alignments is achieved using affine or environment dependent gap penalties, depending on whether gap_function is set to `False` or `True` (arguments to the **Alignment.align()** and **Alignment.align2d()** commands apply).
- **Progressive Multiple Structure Alignments**
  If alignment_type is set to `progressive`, the multiple alignment follows only the last part of the 'tree' alignment where, in $n-1$ alignments, all the structures/sequences are successively aligned to the first one on the list.

The alignment of proteins within a sub-alignment does not change when the sub-alignment is aligned with another protein or sub-alignment. The pairwise alignment of sub-alignments is guided by the dendrogram. First, the most similar pair of proteins are aligned. Second, the next most similar pair of proteins are aligned, or the third protein is aligned with the sub-alignment of the first two, as indicated by the dendrogram. This greedy, progressive procedure requires $n-1$ steps to align all n proteins, and each step requires a pairwise alignment of two sub-alignments.

If in a multiple alignment, overhangs are to be penalized differently for the pairs of alignments that create the multiple, auto_overhang can be set to `True`. This will ensure that the value of overhang changes as overhang_factor times the numerical difference in the residues of the pair. Further, this is only effected if the difference is greater than overhang_auto_limit.

The dendrogram can be written out in a separate file by specifying the file name to dendrogram_file.

Example: examples/salign/salign_multiple_struc.py

```
# Illustrates the SALIGN multiple structure/sequence alignment
from modeller import *

log.verbose()
env = Environ()
env.io.atom_files_directory = ['.', '../atom_files']

aln = Alignment(env)
for (code, chain) in (('1is4', 'A'), ('1uld', 'D'), ('1ulf', 'B'),
                      ('1ulg', 'B'), ('1is5', 'A')):
    mdl = Model(env, file=code, model_segment=('FIRST:'+chain, 'LAST:'+chain))
    aln.append_model(mdl, atom_files=code, align_codes=code+chain)

for (weights, write_fit, whole) in (((1., 0., 0., 0., 1., 0.), False, True),
                                    ((1., 0.5, 1., 1., 1., 0.), False, True),
                                    ((1., 1., 1., 1., 1., 0.), True, False)):
    aln.salign(rms_cutoff=3.5, normalize_pp_scores=False,
               rr_file='$(LIB)/as1.sim.mat', overhang=30,
               gap_penalties_1d=(-450, -50),
               gap_penalties_3d=(0, 3), gap_gap_score=0, gap_residue_score=0,
               dendrogram_file='1is3A.tree',
               alignment_type='tree', # If 'progresive', the tree is not
                                      # computed and all structures will be
                                      # aligned sequentially to the first
               #ext_tree_file='1is3A_exmat.mtx', # Tree building can be avoided
                                                 # if the tree is input
               feature_weights=weights, # For a multiple sequence alignment only
                                        # the first feature needs to be non-zero
               improve_alignment=True, fit=True, write_fit=write_fit,
               write_whole_pdb=whole, output='ALIGNMENT QUALITY')

aln.write(file='1is3A.pap', alignment_format='PAP')
aln.write(file='1is3A.ali', alignment_format='PIR')

# The number of equivalent positions at different RMS_CUTOFF values can be
# computed by changing the RMS value and keeping all feature weights = 0
aln.salign(rms_cutoff=1.0,
           normalize_pp_scores=False, rr_file='$(LIB)/as1.sim.mat', overhang=30,
           gap_penalties_1d=(-450, -50), gap_penalties_3d=(0, 3),
           gap_gap_score=0, gap_residue_score=0, dendrogram_file='1is3A.tree',
           alignment_type='progressive', feature_weights=[0]*6,
           improve_alignment=False, fit=False, write_fit=True,
           write_whole_pdb=False, output='QUALITY')
```

Sub-optimal alignments

The weight matrix can be offset at random, many times over, to generate several 'sub-optimal' alignments. The number of sub-optimal alignments to be output can be specified with n_subopt. Though the matrix positions at which these offsets are applied cannot be controlled, the user can choose by how much the matrix will be offset (subopt_offset). The alignments are written into the file 'suboptimal_alignments.out' (or 'suboptimal_alignments2d.out' if gap_function is `True`) in a simple format. For each such alignment, an `'ALIGNMENT:'` line is written, containing in order

- the number of the alignment (the first alignment is numbered 0, and is the true or 'optimal' alignment, for reference)
- the start, end and length of the alignment
- the number of aligned positions
- the score of the alignment
- three final fields for internal use

After this the two sequences are written as `'SEQ1'` and `'SEQ2'`, as a simple mapping from alignment positions to sequences.

Note that subopt_offset should be positive for alignments using distance matrices (similarity_flag = `False`) and negative when using similarity matrices (similarity_flag = `True`).

Note that because the suboptimal alignments are generated sequentially, the alignment in memory at the end of the command will be the last (or worst) suboptimal alignment, *not* the optimal alignment.

The suboptimal alignment file can be converted into a set of real alignments using the **Alignment.get_suboptimals()** method.

Example: examples/salign/salign_subopt.py

```
from modeller import *

log.verbose()
env = Environ()
aln = Alignment(env, file='fm07254_test.ali', alignment_format='PIR')
aln.salign(feature_weights=(1., 0, 0, 0, 0, 0), gap_penalties_1d=(-450, -50),
           n_subopt = 5, subopt_offset = 15)

# Convert suboptimal alignment output file into actual alignments
f = open('suboptimal_alignments.out')
for (n, aln) in enumerate(aln.get_suboptimals(f)):
    aln.write(file='fm07254_out%d.ali' % n)
```

Alignments using external restraints

Fix positions: The user can choose to have certain alignment positions "fixed" by offsetting the appropriate matrix entries. This is done by adding a new pseudo sequence to the alignment with the align code `._fix_pos`. The residues of this pseudo sequence are integer values from 0 through 4 (alternatively, a blank is equivalent to 0). Any alignment position at which this pseudo sequence contains a '0' is treated normally; if, however, a non-zero integer is used, the alignment matrix is offset, generally making the alignment in that position more favorable (and more so for higher integers). The actual offset values themselves can be specified by the user by setting the fix_offsets variable. Note that since SALIGN converts all DP scoring matrices to distance matrices (unless otherwise specified using similarity_flag), the values of fix_offsets used in anchoring alignment positions should be numerically smaller or negative in comparison to the values in the DP matrix.

Example: examples/salign/salign_fix_positions.py

```
# Demonstrating the use of alignment restraints, only available in
# align2d and salign:
from modeller import *

log.verbose()
env = Environ()

# The special alignment entry '_fix_pos' has to be the last entry in the
# alignment array. Its sequence contains characters blank (or 0), 1, 2, 3,
# and 4 at the restrained alignment positions. The residue-residue score from
# the substitution matrix for these positions will be offset by the scalar
# value FIX_OFFSETS[0..4].
aln = Alignment(env, file='fix_positions.ali', align_codes=('1leh', '3btoA',
                                                             '_fix_pos'))

# fix_offsets specifies the offset corresponding to character ' 1234' in the
# _fix_pos entry in the alignment
# (this offsets unlabeled positions for 0, the ones indicated by 1 by
#  1000, those indicated by 2 by 2000, etc.)
aln.salign(fix_offsets=(0, -10, -20, -30, -40),
           gap_penalties_2d=(0, 0, 0, 0, 0, 0, 0, 0, 0), # Any values are
                                                         # possible here
           local_alignment=False, # Local alignment works, too
           gap_penalties_1d=(-600, -400)) # This is best with the default value
                                          # of gap_penalties_2d

# Write it out, the _fix_pos is erased automatically in salign:
aln.write(file='fix_positions_salign.pap', alignment_format='PAP')
```

External weight matrix: An example of using feature 6.

Example: examples/salign/salign_external_matrix.py

```
# Reads an external matrix
from modeller import *

log.verbose()
env = Environ()

aln = Alignment(env, file='1dubA-1nzyA.ali', align_codes='all')

aln.salign(alignment_type='pairwise', output='',
           rr_file='$(LIB)/blosum62.sim.mat',
           #rr_file='$(LIB)/as1.sim.mat',
           #max_gap_length=20,
           gap_function=False,
           input_weights_file='external.mtx',   # External weight matrix
           #weights_type='DISTANCE',            # type of ext. wgt. mtx
           # ensure appropriate gap penalites for the ext. matrix
           #feature_weights=(1., 0., 0., 0., 0., 0.), gap_penalties_1d=(30, 26),
           #output_weights_file='score.mtx',
           feature_weights=(1., 0., 0., 0., 0., 1.),
           gap_penalties_1d=(-500, -300))
aln.write(file='output.ali', alignment_format='PIR')
aln.write(file='output.pap', alignment_format='PAP')
```

Multiple structure alignment according to a user specified dendrogram The user has the option of inputting an n X n matrix from which a dendrogram can be inferred. The multiple tree alignment is then confined to follow this externally input dendrogram. To effect this, specify the name of the external matrix file with the ext_tree_file variable. The format of this file is the same as the external weight matrix, above; the first line of the file contains the dimensions of the matrix (in this case it should be a square matrix, with both dimensions equal to the number of sequences in the alignment), and subsequent lines contain sequence-sequence distances, one row of the matrix per line.

Gap penalties and correcting for gaps

SALIGN makes use of three sets of gap penalties. gap_penalties_1d are for dynamic programming making use of constant gap penalties. gap_penalties_2d are when a variable function for gap penalty is used. gap_penalties_3d is used along with feature 2 only, when structures are aligned by a least squares fit of their atomic positions. All SALIGN features produce some measure of residue equivalence (similarity or distance scores). The scales of these scores differ depending on the feature used. For optimal usage, gap_penalties_1d should be set appropriately considering the features used. Note: If feature 1 is non zero and a similarity substitution matrix is employed, no matter what other features are also used in conjunction, gap_penalties_1d should always take on values appropriate to the substitution matrix used. For example, if feature 1 is non zero (other features may or may not be non-zero), and the residue substitution matrix used is the BLOSUM62 similarity matrix, gap_penalties_1d is set to (-450, -50) and when feature 1 is zero gap_penalties_1d is set to values appropriate for a distance matrix, *e.g.*, (2, 3). A word of caution: gap penalties have not yet been optimized for aligning sequences by their profiles and for structure alignments.

The gap correction function is $g_{i,j} = \frac{n_{rg}}{(n_1 n_2)} r + \frac{n_{gg}}{(n_1 n_2)} g$, where n_1 and n_2 are the number of proteins in the two sub-alignments, n_{rg} is the number of gap–residue pairs, and n_{gg} is the number of gap–gap pairs when comparing protein positions from one sub-alignment with protein position from the other sub-alignment, r is gap_residue_score and g is gap_gap_score. The smaller (even negative) is gap_gap_score, and the larger is gap_residue_score, the more will the gaps be aligned with gaps.

Whenever an alignment of two sequences is done and feature 1 is non-zero, SALIGN will attempt to keep any chain breaks aligned. This behavior can be tuned by setting the break_break_bonus parameter. See **Alignment.align()** for more information.

Useful SALIGN information and commands

The **Alignment.salign()** command uses position-position dissimilarity scores (except when similarity_flag is switched on), as opposed to similarity scores. This convention applies to all the features, including the residue-residue similarities read from the rr_file; however, if a residue type – residue type similarity matrix is read in, it is automatically converted into the distance matrix by $D = \max_{i,j} S_{i,j} - S$. In addition, it is also scaled linearly such that the residue–residue dissimilarity scores range from 0 to 1 (to facilitate weighting this feature with other features).

For each pairwise alignment, the weight matrix W has dimensions N and M that correspond to the lengths of the sub-alignments to be aligned based on the weight matrix W. The dissimilarity score for aligning position i with position j is calculated as $W_{i,j} = \sum_f [\frac{\omega_f}{\sum_f \omega_f} W^f_{i,j}] + g_{i,j}$, where the sum runs over all selected features f, and g is a function that may be used to correct the $W_{i,j}$ score for the presence of gaps within the sub-alignments (see below). A feature f is selected when its weight ω_f (specified in feature_weights) is non-zero. The matrices W^f are normalized to have the mean of 0 and standard deviation of 1 when normalize_pp_scores is `True`, but it is recommended not to use this option for now (*i.e.*, use feature_weights to scale the contributions of the different features to the final W). The weights of 1 will weigh the different features approximately evenly (the residue-residue dissimilarities of feature 1 are scaled to a range from 0 to 1, the position differences of feature 2 are in angstroms, the fractional solvent accessibility scores of feature 3 and the secondary structure scores of feature 4 range from 0 to 2, and the DRMS difference of feature 5 is expressed in angstroms).

If you enable verbose logging with **log.verbose()**, there will be more output in the `'log'` file, such as the dendrogram. The dendrogram can also be written out in a separate file by specifying the file name to dendrogram_file.

Argument output can contain the following values:

- `'ALIGNMENT'`: the alignments in the first $n-2$ stages of the pairwise alignment of sub-alignments are written out.
- `'QUALITY'`: the final alignment is used to obtain pairwise least-squares superpositions and the corresponding average and minimal numbers of pairs of aligned residues that are within rms_cutoff Å in all pairs of aligned structures. These numbers can be used as absolute quality measures for the final multiple alignment. This option requires the coordinate files for the aligned proteins.

If write_fit is True, the fitted atom files are written out in their fitted orientations. For this and other options below, also read the text above.

If output_weights_file is specified, the dynamic programming weight matrix is written out into the file. (If it is None, no file is written out.)

If current_directory is True, the output _pdb.fit files will be written to the current directory. Otherwise, the output will be in the directory with the original files[9].

If write_whole_pdb is True, the whole PDB files are written out[10]; otherwise only the parts corresponding to the aligned sequences are output.

If fit is False, the initial superposition is not changed. This is useful when all the structures have to be compared with a given alignment as is, without changing their relative orientation.

If fit_on_first is True, the structures are fit to the first structure according to the final alignment before they are written out.

If improve_alignment is False, the initial alignment is not changed, though the structures may still be superimposed if fit = True. This is useful when all the structures have to be superimposed with the initial alignment.

6.16.27 Alignment.get_suboptimals() — parse suboptimal alignments file

```
get_suboptimals(f, align_block=0)
```

This command, given an open Python file, will read the alignments (from **Alignment.salign()**) in that file into the alignment object, one by one. (The first such alignment is the optimal alignment for reference; the remainder are the suboptimal alignments.) The alignment object itself is returned. The alignment must contain only the two sequences which correspond to the alignment from which the suboptimal alignment file was originally generated, in the case of a simple pairwise alignment. If the alignment was originally generated as an alignment of two previously-aligned blocks, then the alignment must contain the same set of sequences, and align_block must be set to the same value as when the alignment was generated.

Example: See **Alignment.salign()** command.

6.16.28 Alignment.to_profile() — convert alignment to profile format

```
to_profile()
```

This command will convert the alignment, currently in memory, into the profile format. For more details on the profile format, see **Profile.read()**.

Example: examples/commands/aln_to_prof.py

```
from modeller import *
env = Environ()

# Read in the alignment file
aln = Alignment(env)
aln.append(file='toxin.ali', alignment_format='PIR', align_codes='ALL')

# Convert the alignment to profile format
```

[9]This won't work in combination with write_whole_pdb = False for structures that were added to the alignment with **Alignment.append_model()**, since such inputs may not have an atom file to extract the directory from. In this case the outputs will end up in the current directory.

[10]Any structures that were added with **Alignment.append_model()** will need to have their corresponding atom files available, so that the originals can be reread at this point.

```
prf = aln.to_profile()

# Write out the profile

# in text file
prf.write(file='alntoprof.prf', profile_format='TEXT')

# in binary format
prf.write(file='alntoprof.bin', profile_format='BINARY')
```

6.16.29 Alignment.segment_matching() — align segments

`segment_matching(file, root_name, file_ext, file_id, align_block, segment_report, segment_cutoff, segment_shifts, segment_growth_n, segment_growth_c, min_loop_length, rr_file='$LIB/as1.sim.mat')`

This command enumerates alignments between two blocks of sequences. More precisely, it enumerates the alignments between the segments in the first block and the sequences in the second block. The segments can be moved to the left and right as well as lengthened and shortened, relative to the initial alignment. The regions not in segments or not aligned with segments are left un-aligned, possibly to be modeled as insertions. Typically, the first block of sequences corresponds to structures, the segments to secondary structure elements in the first block, and the second block to the sequences one of which is to be modeled later on. The command is useful for generating many alignments which can then be used by another MODELLER script to generate and evaluate the corresponding 3D models.

All the sequences and segments are defined in the alignment array. The first block of sequences, the ones with segments, are the first align_block sequences. The regions corresponding to the segments are defined by the last entry in the alignment as contiguous blocks of non-gap residues. Any standard single character residue code may be used. The segments must be separated by gap residues, '-'. The remaining sequences from align_block $+ 1$ to NSEQ $- 1$ are the second block of sequences. The alignment of the sequences within the two blocks does not change. A sample alignment file is

```
 _aln.pos         10        20        30        40        50        60
7rsa      KETAAAKFERQHMDSSTSAASSSNYCNQMMKSRNLTKDRCKPVNTFVHESLADVQAVCSQKNVAC-KN
edn       ---KPPQFTWAQWFETQHINMTSQQCTNAMQVINNYQRRCKNQNTFLLTTFANVVNVCGNPNMTCPSN
templ     --HHHHHHHHHH-----------GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
 _consrvd        *              *  *   *   *    ***  ***     * *  **   *  *  *


 _aln.p   70        80        90       100       110       120       130
7rsa      -GQTNCYQSYSTMSITDCRETGSS--KYPNCAYKTTQANKHIIVACEGN---------PYVPVHFDAS
edn       KTRKNCHHSGSQVPLIHCNLTTPSPQNISNCRYAQTPANMFYIVACDNRDQRRDPPQYPVVPVHLDRI
templ     GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
 _consrvd     **  * *      *  *  *     ** *  * **   ****            * **** *


 _aln.pos
7rsa      V
edn       I
templ     G
 _consrvd
```

The enumeration of alignments explores all possible combinations of alignments between each segment and the 2nd block of sequences: The starting position of each segment i is varied relative to the input alignment in the interval from segment_shift[$2i-1$] to segment_shift[$2i$]. There has to be at least min_loop_length[i] and min_loop_length[$i+1$] residues that are not in any segment before and after the i-th segment, respectively. The location of the N-terminus of segment i is varied relative to the location in the input alignment in the interval from segment_growth_n[$2i-1$] to segment_growth_n[$2i$]. Similarly, the location of the C-terminus of segment i is varied relative to the location in the input alignment in the interval from segment_growth_c[$2i-1$] to segment_growth_c[$2i$]. The shortening and lengthening of the segments may be useful in determining the best anchor regions for modeling of a loop.

Each alignment is scored according to the similarity scoring matrix specified by filename rr_file. This matrix may contain residue-gap scores, the gap being residue type 21; otherwise the value is set to the smallest value in the matrix. The score for an alignment is obtained by summing scores only over all alignment positions corresponding to the segments (no gap penalty is added for loops). When there is more than one sequence in any of the two blocks, the position score is an average of all pairwise comparisons between the two blocks of sequences. In the case where the number of positions in the alignment changes (*i.e.*, the segments grow or shorten), the scores are not comparable to each other. It is feasible to enumerate on the order of 10^{10} different alignments in less than one hour of CPU time.

In general, two runs are required. In the first run, the alignments are scored and a histogram of the scores is written to file file. Then this file must be inspected to determine the cutoff segment_cutoff. In the second run, all the alignments with a score higher than segment_cutoff are written to files in the PIR format, using the standard file naming convention: root_namefile_id*nnnn*0000file_ext, where *nnnn* is the alignment file counter. In addition, the alignments are also written out in the PAP format for easier inspection by eye. Thus, segment_cutoff has to be set to a very large value in the first run, to avoid writing alignment files. During a run, a message is written to the log every segment_report alignments; this is useful for knowing what is going on during very long runs.

Example: examples/commands/segment_matching.py

```
# Example for: Alignment.segment_matching()

from modeller import *

log.level(1, 1, 1, 1, 0)
env = Environ()

aln = Alignment(env, file='ednf2.pap', align_codes=('7rsa', 'edn', 'templ'),
                alignment_format='PAP', remove_gaps=True)

aln.segment_matching(file='segmatch.dat',
                     align_block=1, rr_file='$(LIB)/as1.sim.mat',
                     segment_shifts=(-8, 8, 0, 0),
                     segment_growth_n=(0, 0, 0, 0),
                     segment_growth_c=(0, 0, 0, 0),
                     min_loop_length=(0,2,0),
                     segment_report=1000000, segment_cutoff=0,
                     root_name='segmatch', file_ext='.ali', file_id='default')
```

6.17 The Sequence class: a single sequence within an alignment

The Sequence class contains a single sequence, in a model (see Section 6.6) or in an alignment (see Section 6.16).

For alignment template structures (*i.e.*, sequences for which a structure is also available) see the Structure class in Section 6.18.

Example: examples/python/alnsequence.py

```
# Example for alnsequence objects

from modeller import *

env = Environ()

aln = Alignment(env, file='../commands/toxin.ali')
print("Alignment contains %d sequences:" % len(aln))
for seq in aln:
    print("   Sequence %s from %s contains %d residues" \
          % (seq.code, seq.source, len(seq)))
```

6.17.1 Sequence.range — residue range

This is a pair of residue:chain strings, which identify the starting and ending residues and chains to read from a PDB file, when reading the structure to match the sequence. This matches the pair specified in a PIR alignment file header (see Section B.1) and in the model_segment argument to **Model.read()**.

6.17.2 Sequence.code — alignment code

This is a short text which identifies the sequence. Often, the PDB code followed by the chain ID (if any) is used. See also Section B.1.

6.17.3 Sequence.atom_file — PDB file name

This gives the name of the PDB file containing the associated 3D structure for the sequence, if available.

6.17.4 Sequence.source — source organism

This gives the name of the organism from which the sequence was obtained, if available.

6.17.5 Sequence.name — protein name

This gives the full name of the protein, if available.

6.17.6 Sequence.prottyp — protein sequence type

This gives the type of the sequence, usually sequence for a simple sequence, or structureX for a sequence which also has known 3D structure.

6.17.7 Sequence.pdb_accession — PDB accession code

The PDB accession code of the associated structure, or blank if unknown or the structure is not deposited in PDB.

6.17.8 Sequence.resolution — structure resolution

The resolution of the associated X-ray structure, or -1.0 if unknown or not applicable.

6.17.9 Sequence.rfactor — R factor

The R factor of the associated X-ray structure, or -1.0 if unknown or not applicable.

For alignment sequences and structures, this information is read from the PIR alignment file. For models, it is read from the PDB records by **Model.read()**.

6.17.10 Sequence.residues — list of all residues in the sequence

This is a standard Python list of all the residues in the sequence. This can be used to query individual residue properties (*e.g.* amino acid type) or to specify residues for use in restraints, *etc.*

Residues can be individually accessed in two ways:

- A string of the form 'RESIDUE_#[INS][:CHAIN_ID]', where RESIDUE_# is a residue number (generally an integer, although hybrid-36 notation is also understood) and INS an optional insertion code, as they occur in the PDB, mmCIF, or BinaryCIF file of a model[11]; the optional CHAIN_ID is the single character chain id as it occurs in the PDB file. For example, if `'s'` is a `Sequence` object, PDB residue '10' in chain 'A' is given by `'s.residues['10:A']'`; if the chain has no chain id, `'s.residues['10']'` would be sufficient. Note that the quotes are required to force the use of PDB numbers.
- By numeric index, starting from zero, in standard Python fashion. For example, if `'s'` is a `Sequence` object, `'s.residues[1]'` is the second residue. Contrast with `'s.residues['1']'` above, which returns the residue with PDB number '1'.

See Section 6.20 for more information about `Residue` objects. See also **Model.residue_range()**, for getting a contiguous range of residues in a model.

6.17.11 Sequence.chains — list of all chains in the sequence

This is a standard Python list of all the chains in the model. You can index this list either in standard Python fashion, or by using the one-letter PDB chain ID, for example if `'s'` is a `Sequence` object, and the first chain has ID 'A', both `'s.chains[0]'` and `'s.chains['A']'` will index this chain.

See Section 6.19 for more information about `Chain` objects.

Example: See **Selection.assess_dope()** command.

6.17.12 Sequence.transfer_res_prop() — transfer residue properties

```
transfer_res_prop()
```

The predicted secondary structure, along with the confidence of prediction, of this sequence is transferred to all other sequences in the alignment. Only available for sequences in alignments.

6.17.13 Sequence.get_num_equiv() — get number of equivalences

```
get_num_equiv(seq)
```

[11]residue numbers in mmCIF or BinaryCIF refer to author-provided values, if available (e.g. `atom_site.auth_seq_id`), otherwise to `label_seq_id`.

This returns the number of identical aligned residues between this sequence and `seq`, which must be another `Sequence` object from the same alignment. Only available for sequences in alignments.

Example: See **Alignment.id_table()** command.

6.17.14 Sequence.get_sequence_identity() — get sequence identity

```
get_sequence_identity(seq)
```

This returns the percentage sequence identity between this sequence and `seq`, which is defined as the number of identical aligned residues divided by the length of the shorter sequence. Only available for sequences in alignments.

Example: See **Alignment.id_table()** command.

6.18 The Structure class: a template structure within an alignment

The `Structure` class contains a single template structure from an alignment. It derives from the alignment `Sequence` class and is used in a very similar way (see Section 6.17 for more details), although it additionally provides special methods to handle structures, and atom information (just as for `Model` objects) is available, unlike regular alignment sequences which only deal with residues.

6.18.1 Structure.write() — write out PDB file

```
write(file)
```

The template structure is written out to the named file, in PDB format. file can be either a file name or a **modfile.File()** object open in write mode (in which case the structure is appended to the file).

6.18.2 Structure.reread() — reread coordinates from the atom file

```
reread()
```

The current coordinates in memory are 'forgotten', and they are reread from the atom file (Sequence.atom_file). This is useful if you want to restore the original template orientation after some command which changes it (*e.g.*, **Alignment.check()**).

6.18.3 Structure.read() — read coordinates from a PDB file

```
read(file, io=None)
```

Normally, coordinates are read automatically from Sequence.atom_file when needed (or when **Structure.reread()** is called). This function instead reads coordinates explicitly in PDB format from file, which is either a filename or a readable file handle (see **modfile.File()**), ignoring atom_file. This is useful if you want to load an alternative conformation, or you want to read the structure from a file handle rather than a named file. (The sequence of the PDB must still match that in the alignment, of course.)

Note that **Structure.reread()**, and certain structural alignment functions (**Alignment.salign()** and **Alignment.malign3d()**) will overwrite this conformation with that named in Sequence.atom_file.

6.19 The Chain class: a single chain in a model or alignment

The Chain class holds information about a single chain, in a Model (see Sequence.chains), an alignment sequence, or an alignment template structure.

Two Chain objects are considered equal if and only if they represent the same chain in the same sequence.

Example: examples/python/chains.py

```
# Example for 'chain' objects

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

mdl = complete_pdb(env, "1b3q")

# Print existing chain IDs and lengths:
print("Chain IDs and lengths: " \
      + str([(c.name, len(c.residues)) for c in mdl.chains]))

# Set new chain IDs:
mdl.chains['A'].name = 'X'
mdl.chains['B'].name = 'Y'

# Write out chain sequences:
for c in mdl.chains:
    c.write(file='1b3q%s.chn' % c.name, atom_file='1b3q',
            align_code='1b3q%s' % c.name)
```

6.19.1 Chain.name — chain ID

This is the name (ID) of the chain. Modeller allows for chain names up to 20 characters, but note that only a single character can be stored in traditional PDB files (two characters if IOData.hybrid36 is set); if you wish to use longer chain IDs, use mmCIF format instead of PDB.

6.19.2 Chain.residues — all residues in the chain

This is a list of all residues in the chain, as Residue objects (see Section 6.20). For example, 'm.chains['A'].residues['10']' would yield the residue with PDB number 10 in chain A in the model 'm'[12].

6.19.3 Chain.atoms — all atoms in the chain

This is a list of all atoms in the chain, as Atom objects (see Section 6.23). (Not available for alignment sequences.)

[12]Note that 'm.chains['A'].residues['10']' in most cases is the same as 'm.residues['10:A']'. However, if there are multiple chains in the model called 'A' the first syntax will return residue 10 in the first chain called A (and so will fail if that chain does not contain this residue) whereas the second will look for the first residue numbered 10 in any of the A chains. (Generally speaking, you should avoid having duplicate chain IDs or residue numbers!)

6.19.4 Chain.filter() — check if this chain passes all criteria

`filter(structure_types='structure', minimal_resolution=99.0, minimal_chain_length=30, max_nonstdres=10, chop_nonstd_termini=True, minimal_stdres=30)`

This checks the chain with various criteria, and returns `True` only if all of them are met. This is useful in combination with **Chain.write()** to produce sequences of chains. See also **Model.make_chains()**.

structure_types refers to the experimental method used to determine the structure. The following types are recognized: 'structureX' for X-ray, 'structureN' for NMR and 'structureM' for model, 'structureE' for electron microscopy, 'structureF' for fiber diffraction, 'structureU' for neutron diffraction, 'structure' for any structure.

chop_nonstd_termini, if set, removes a single non-standard residue (if present) from each terminus of the chain. This is done before the chain length criteria below are considered.

minimal_resolution refers to the cut-off value of the experimental resolution of the structure. Structures with resolutions larger than this threshold are not accepted.

minimal_chain_length refers to the lower limit of the chain length. Chains whose lengths are smaller than this value are not accepted.

max_nonstdres sets the maximum limit of non-standard residues that is tolerated.

minimal_stdres sets the minimum number of standard residues that are required to process the chain. Chains that don't have at least this number of standard residues are not accepted.

Example: examples/commands/make_chains.py

```
# Example for: chain.filter(), chain.write()

# This will read a PDB file (segment), and write out all of its chains
# satisfying the listed conditions into separate alignment files in the
# PIR format.

from modeller import *

env = Environ()
mdl = Model(env, file='../atom_files/pdb1lzd.ent')
for c in mdl.chains:
    if c.filter(minimal_chain_length=30, minimal_resolution=2.0,
                minimal_stdres=30, chop_nonstd_termini=True,
                structure_types='structureN structureX'):
        filename = '1lzd%s.chn' % c.name
        print("Wrote out " + filename)
        atom_file, align_code = c.atom_file_and_code(filename)
        c.write(filename, atom_file, align_code, format='PIR',
                chop_nonstd_termini=True)
```

6.19.5 Chain.write() — write out chain sequence to an alignment file

`write(file, atom_file, align_code, comment='', format='PIR', chop_nonstd_termini=True)`

This writes out the residue sequence of the chain to an alignment file.

file can be either a file name or a **modfile.File()** object open in write mode (in which case the structure is appended to the file).

atom_file and align_code specify the name of the chain's associated atom file, and its alignment code, respectively; suitable values can be obtained from **Chain.atom_file_and_code()**. comment, if given, specifies a comment to prepend to the alignment file.

format specifies the format of the output file; see **Alignment.write()**. `PIR` or `FASTA` formats are supported.

chop_nonstd_termini trims non-standard terminal residues in exactly the same way as for **Chain.filter()**.

This command is not available for alignment sequences, because the PDB residue numbers (only available in template structures or models) are needed to write the PIR header.

Example: See **Chain.filter()** command.

6.19.6 Chain.atom_file_and_code() — get suitable names for this chain

`atom_file_and_code(filename)`

Given a model filename, this returns suitable atom_file and align_code values for this chain, for example for giving to **Chain.write()**. Path names are stripped, and duplicate chain IDs are handled.

For example, a model filename of `/home/user/test.pdb` may return `test` for atom_file and `testA` for align_code when called for the `A` chain.

Example: See **Chain.filter()** command.

6.19.7 Chain.join() — join other chain(s) onto this one

`join(chain)`

Given another chain from the same model, alignment structure, or alignment sequence, this command will remove any chain breaks between the two chains. The passed chain (and any other chains between the two) will become part of the first. For example, if in a model containing five chains A, B, C, D and E chains D and B are joined, the model will end up with chains A, B, and E; former chains C and D will become part of the B chain.

Note that the chain must follow this chain in the sequence (*e.g.*, you can join chain B or C onto chain A, but not chain A onto B or C). If it does not, or it is from a different sequence, a `ValueError` is raised.

Note that this does not renumber the residues; you will need to do that separately if you don't want duplicate residue numbers.

Note that for models this does not affect the model topology — any existing C-terminal or N-terminal patches (*e.g.*, OXT atoms) are not removed, and no bonds are created between the termini of the joined chains (so TER records will be missed when writing out the model as a PDB file, for example). To regenerate the topology, write out the model (**Model.write()**) and then read it back in (**complete_pdb()**).

Example: examples/commands/join_chains.py

```
# Example for: Chain.join()

# This will take a model containing two chains and join them into one.

from modeller import *

env = Environ()
env.io.atom_files_directory = ['../atom_files']

mdl = Model(env)
mdl.read(file='2abx')
```

```
# Join the B chain onto the end of the A chain
mdl.chains['A'].join(mdl.chains['B'])

# Renumber all residues in the new chain starting from 1
for num, residue in enumerate(mdl.chains['A'].residues):
    residue.num = '%d' % (num + 1)

mdl.write(file='2abx-join.pdb')
```

6.20 The Residue class: a single residue in a model or alignment

The `Residue` class holds information about a single residue in a sequence (see Sequence.residues).

Two `Residue` objects are considered equal if and only if they represent the same residue in the same sequence.

Example: examples/python/residues.py

```
# Example for 'residue' objects

from modeller import *
from modeller.scripts import complete_pdb

def analyze_seq(description, seq):
    """Simple 'analysis' of a sequence of residues, from a model or alignment"""
    numcys = 0
    for res in seq:
        if res.pdb_name == 'CYS':
            numcys += 1
    print("%s contains %d residues, of which %d are CYS" \
          % (description, len(seq), numcys))

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

mdl = complete_pdb(env, "1fas")

# 'mdl.residues' is a list of all residues in the model
print("1-letter code of 1st residue: " + mdl.residues[0].code)
print("PDB name of residue '10' in chain A: " + mdl.residues['10:A'].pdb_name)

# Get the first aligned sequence from a file
aln = Alignment(env, file='../commands/toxin.ali')
firstseq = aln[0]

# Analyze all residues in the model, a subset, and all residues in the
# alignment sequence
analyze_seq("Model 1fas", mdl.residues)
analyze_seq("First 10 residues of 1fas", mdl.residue_range('1:A', '10:A'))
analyze_seq("Aligned sequence %s" % firstseq.code, firstseq.residues)
```

6.20.1 Residue.name — internal (CHARMM) residue type name

This is the name used internally to identify the residue type, and corresponds to the CHARMM 4-letter (or shorter) name used in `'restyp.lib'` and `'top_heav.lib'`. These names are a superset of those used in PDB.

6.20.2 Residue.pdb_name — PDB (IUPAC) type name

This is the 3-letter name of the residue, as used in PDB.

6.20.3 Residue.code — One-letter residue type code

This is the one-letter residue type code, as used in alignment files.

6.20.4 Residue.hetatm — HETATM indicator

If `True`, this residue is marked in PDB as a HETATM residue rather than an ATOM residue.

6.20.5 Residue.index — internal integer index

This is the index used internally to identify the residue; residues are numbered sequentially starting from 1.

Example: examples/commands/write_pdb_xref.py

```
# This demonstrates relating PDB residue numbers with residue indices.

from modeller import *

log.verbose()
env = Environ()
env.io.atom_files_directory = ['../atom_files']

mdl = Model(env, file='2abx')

print("Mapping from residue indices to PDB residue and chain names:")
for r in mdl.residues:
    print("%6d   %3s:%s   %s" % (r.index, r.num, r.chain.name, r.pdb_name))
```

6.20.6 Residue.num — PDB-style residue number plus insertion code

This is the full number of the residue, as used in PDB, comprising an integer number (positive or negative) and an optional single-character insertion code, generally a letter from A to Z. (Not available for alignment sequences.)

This is a string property (not an integer). Thus, both `'1'` and `'1A'` are valid residue numbers, but a Python integer (without quotes) is not.

To access just the residue number or the insertion code, see Residue.intnum or Residue.inscode.

Example: See Residue.index command.

6.20.7 Residue.intnum — PDB-style residue number

This is the number of the residue, as used in PDB, as an integer. (Not available for alignment sequences.) It does not include the insertion code (see Residue.inscode or Residue.num.)

6.20.8 Residue.inscode — PDB-style residue insertion code

This is the (optional) insertion code of the residue, as used in PDB, generally a letter from A to Z. (Not available for alignment sequences.)

6.20.9 Residue.curvature — Mainchain curvature

The mainchain curvature, as defined in **Model.write_data** (see the 'CRV' option). Only available for alignment structure residues. This is zero until an alignment using structure-dependent gap penalties is carried out (**Alignment.align2d** or **Alignment.salign** with gap_function = `True`).

6.20.10 Residue.atoms — all atoms in the residue

This is a list of all atoms in the residue, as `Atom` objects (see Section 6.23). (Not available for alignment sequences.)

6.20.11 Residue.chain — chain object

This is the `Chain` object to which the residue belongs. See Section 6.19.

6.20.12 Residue.phi — ϕ dihedral angle

This is a `Dihedral` object, with information about the residue's ϕ dihedral. (If no ϕ dihedral is defined for this residue, it is the special Python value `None` instead.) See Section 6.21 for more information on `Dihedral` objects. Not available for alignment sequence residues.

6.20.13 Residue.psi — ψ dihedral angle

This is a `Dihedral` object, with information about the residue's ψ dihedral; see Residue.phi for more information. Not available for alignment sequence residues.

6.20.14 Residue.omega — ω dihedral angle

This is a `Dihedral` object, with information about the residue's ω dihedral; see Residue.phi for more information. Not available for alignment sequence residues.

6.20.15 Residue.alpha — α dihedral angle

This is a `Dihedral` object, with information about the residue's α dihedral (*i.e.*, the virtual dihedral between four successive C_α atoms, starting with the previous residue); see Residue.phi for more information. Not available for alignment sequence residues.

6.20.16 Residue.chi1 — χ_1 dihedral angle

This is a `Dihedral` object, with information about the residue's χ_1 dihedral; see Residue.phi for more information. Not available for alignment sequence residues.

6.20.17 Residue.chi2 — χ_2 dihedral angle

This is a `Dihedral` object, with information about the residue's χ_2 dihedral; see Residue.phi for more information. Not available for alignment sequence residues.

6.20.18 Residue.chi3 — χ_3 dihedral angle

This is a `Dihedral` object, with information about the residue's χ_3 dihedral; see Residue.phi for more information. Not available for alignment sequence residues.

6.20.19 Residue.chi4 — χ_4 dihedral angle

This is a Dihedral object, with information about the residue's χ_4 dihedral; see Residue.phi for more information. Not available for alignment sequence residues.

6.20.20 Residue.chi5 — χ_5 dihedral angle

This is a Dihedral object, with information about the residue's χ_5 dihedral; see Residue.phi for more information. Not available for alignment sequence residues.

6.20.21 Residue.get_aligned_residue() — get aligned residue in another sequence

```
get_aligned_residue(seq)
```

Given a sequence (or template structure) in the same alignment, this returns another Residue object, for the residue in that sequence which is aligned with this residue. (If there is a gap in the other sequence, None is returned instead.) Not available for model sequences.

See also Alignment.positions.

6.20.22 Residue.add_leading_gaps() — add gap(s) before this residue

```
add_leading_gaps(ngap=1)
```

This adds ngap gaps in the alignment, immediately preceding this residue. Not available for model sequences.

6.20.23 Residue.add_trailing_gaps() — add gap(s) after this residue

```
add_trailing_gaps(ngap=1)
```

This adds ngap gaps in the alignment, immediately after this residue. (Since it makes no sense to have gaps aligned with gaps at the end of the alignment, this will have no effect when called for the last residue in the sequence.) Not available for model sequences.

6.20.24 Residue.remove_leading_gaps() — remove gap(s) before this residue

```
remove_leading_gaps(ngap=1)
```

This removes ngap gaps from the alignment, immediately before this residue. Not available for model sequences.

6.20.25 Residue.remove_trailing_gaps() — remove gap(s) after this residue

```
remove_trailing_gaps(ngap=1)
```

This removes ngap gaps from the alignment, immediately after this residue. (This usually has no effect if called for gaps after the last residue in a sequence, since those gaps are necessary in order to line up with other sequences in the alignment.) Not available for model sequences.

6.20.26 Residue.get_leading_gaps() — get number of gaps before this residue

`get_leading_gaps()`

This returns the number of gaps in the alignment immediately preceding this residue. Not available for model sequences.

6.20.27 Residue.get_trailing_gaps() — get number of gaps after this residue

`get_trailing_gaps()`

This returns the number of gaps in the alignment immediately following this residue. Not available for model sequences.

6.21 The `Dihedral` class: a single dihedral in a model or alignment

The `Dihedral` class holds information about one of a residue's dihedral angles. See Residue.phi for more information on accessing these objects.

6.21.1 Dihedral.value — current value in degrees

This is the current value of the dihedral angle in degrees, and ranges from -180.0 to 180.0.

6.21.2 Dihedral.atoms — atoms defining the angle

This is the list of the four atoms which define the dihedral angle.

6.21.3 Dihedral.dihclass — integer dihedral class

This is the current value of the dihedral class, as defined in `'modlib/resdih.lib'`.

6.22 The `Point` class: a point in Cartesian space

The `Point` class specifies an arbitrary point in the Cartesian space of a model.

`Point` objects can be created by calling **Model.point()**.

6.22.1 Point.x — x coordinate

This is the current x coordinate, in angstroms, of the point. Similar members exist for the y and z coordinates. You can also assign to Point.x to move the point in space.

6.22.2 Point.select_sphere() — select all atoms within radius

```
select_sphere(radius)
```

This returns a new selection (see Section 6.9) containing all atoms currently within the given distance from the point. Compare with **Selection.select_sphere()**.

Example: See **Selection()** command.

6.23 The Atom class: a single atom in a model or structure

The Atom class holds information about a single atom, in a Model (see Model.atoms) or an alignment template structure (Structure class). The Atom class is derived from the Point class, and thus all Point methods (*e.g.*, **Point.select_sphere()**) can also be called on Atom objects, and all Point members (*e.g.*Point.x) are available. See Section 6.22.

Two Atom objects are considered equal if and only if they represent the same atom in the same structure.

Example: examples/python/atoms.py

```
# Example for 'atom' objects

from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

mdl = complete_pdb(env, "1fas")

# 'mdl.atoms' is a list of all atoms in the model
print("Name of C-alpha atom in residue 4 in chain A: %s " \
      % mdl.atoms['CA:4:A'].name)
a = mdl.atoms[0]
print("Coordinates of first atom: %.3f, %.3f, %.3f" % (a.x, a.y, a.z))

# Each 'residue' object lists its own atoms, as does each chain
a = mdl.residues['10:A'].atoms[0]
print("Biso for first atom in residue 10 in chain A %.3f" % a.biso)

a = mdl.chains[0].residues[-1].atoms[-1]
print("Biso for last atom in last residue in first chain: %.3f" % a.biso)
```

6.23.1 Atom.dvx — objective function derivative

This is the first derivative of the objective function, from the last energy function, with respect to the x coordinate of the atom. Similar members exist for dvy and dvz. (Only available for models, not template structures.)

6.23.2 Atom.vx — x component of velocity

This is the x component of the velocity, from the most recent molecular dynamics simulation. Similar members exist for vy and vz. (Only available for models, not template structures.)

6.23.3 Atom.biso — isotropic temperature factor

This is the isotropic temperature factor (B_{iso}), which can be set by **Selection.energy()**, **Model.write_data()** or **Model.make_region()**.

Example: See **Model.write_data()** command.

6.23.4 Atom.accessibility — atomic accessibility

This is the atomic accessibility. It is zero for new structures, but can be calculated by **Model.write_data()**.

6.23.5 Atom.occ — occupancy

This is the crystallographic occupancy of the atom.

6.23.6 Atom.charge — electrostatic charge

This is the electrostatic charge of the atom. (Only available for models, not template structures.)

6.23.7 Atom.mass — mass

This is the mass of the atom. (Only available for models, not template structures.)

6.23.8 Atom.name — PDB name

This is the name used to refer to the atom in PDB.

6.23.9 Atom.type — CHARMM atom type

This is a CHARMM AtomType object (see Section 6.24), which tracks per-type rather than per-atom properties such as the element symbol, radius and atomic mass. If the model has not yet had CHARMM atom types assigned, it is `None`.

The atom type can also be changed by assigning a CHARMM atom name (as a string, such as 'CT1'), or `None`, to it. This can be useful when using BLK residues, which otherwise use the same atom type (`'undf'`) for every atom.

(Only available for models, not template structures.)

6.23.10 Atom.residue — residue object

This returns the `Residue` object to which this atom belongs. See Section 6.20.

6.23.11 Atom.get_equivalent_atom() — get equivalent atom in another residue

```
get_equivalent_atom(res)
```

Given another `Residue` object, this returns an `Atom` object representing the structurally equivalent atom in that residue, or `None` if no such atom can be found. (This is primarily used by **Restraints.make_distance()**.) The rules for determining equivalency are:

- If either residue is non-standard, but is denoted as being 'similar' to a standard residue with the STD column in `'modlib/restyp.lib'`, treat it below as if it is the similar standard residue (*e.g.*, ASX is treated like ASN).
- If the two residues are of the same type, or either residue has no topology (*e.g.*, it is a BLK residue), find an atom with the same name.
- If the two residues are both standard amino acids, use the mapping defined in `'modlib/atmeqv.lib'` (this is not always a simple match on name; for example, CYS SG is equivalent to ASP CG, since they tend to be similarly placed in space).
- Otherwise, no atom is treated as equivalent (even if the names are the same; for example, the O atom in water residues is not equivalent to the backbone O in standard amino acids).

6.24 The `AtomType` class: a CHARMM atom type

The `AtomType` class holds information about a CHARMM atom type.

6.24.1 AtomType.name — CHARMM name

The CHARMM name of the atom type, such as CT1, NR1, *etc.*

6.24.2 AtomType.mass — atomic mass

The mass of each atom of this type, in atomic units. This can be modified.

6.24.3 AtomType.element — element

The 1 or 2 letter element symbol of each atom of this type. This can be modified.

6.25 The `EnergyProfile` class: a per-residue energy profile

The `EnergyProfile` class holds a per-residue energy profile, as returned by **Selection.get_dope_profile()**, **Selection.get_dopehr_profile()**, or **Model.get_normalized_dope_profile()**.

An energy profile acts like a Python list; each element in the list corresponds to a residue in the model for which the profile was calculated. Each element is an object that contains three attributes. `energy` is the contribution to the energy function that can be attributed to restraints on this residue; `num_restraints` is the number of restraints that act on this residue; `normalized_energy` is simply the energy divided by the number of restraints.

6.25.1 EnergyProfile.min_rms — minimal RMS violation

The root-mean-square of the minimal violation (see Section 5.3.1), over all restraints.

6.25.2 EnergyProfile.heavy_rms — heavy RMS violation

The root-mean-square of the heavy violation (see Section 5.3.1), over all restraints.

6.25.3 EnergyProfile.get_normalized() — get a normalized energy profile

```
get_normalized()
```

This returns a new energy profile, in which each residue's energy is divided by the number of restraints acting on that residue.

6.25.4 EnergyProfile.get_smoothed() — get a smoothed energy profile

```
get_smoothed(window=1)
```

This returns a new energy profile, in which each residue's energy is smoothed by weighted window averaging, using its own energy and the energy of `window` residues either side of it; energies of residues are weighted by how close they are to the center residue in sequence. (Note that this differs from that used by **Selection.energy()**.)

6.25.5 EnergyProfile.write_to_file() — write to file

```
write_to_file(filename)
```

This writes the energy profile to a named file, or an open file handle (see **modfile.File()**). Each line in the file contains the index of the residue (starting from 1, not the PDB numbering) and the residue's energy.

6.26 The Profile class: using sequence profiles

The `Profile` class holds a sequence profile. Sequence profiles are similar to multiple alignments, and can contain gaps, but do not contain all of the information of the `Alignment` class. Profiles can be matched against each other with **Profile.scan()** or enriched from a sequence database with **Profile.build()**.

6.26.1 Profile() — create a new profile

`Profile(env, aln=None, **vars)`

This creates a new `Profile` object. By default, the profile is empty. However, if you give a single `Alignment` object as an argument, the profile is initialized with the alignment contents (using **Alignment.to_profile()**), or if you specify any keyword arguments, they are passed to **Profile.read()**, to read in a profile from a file. See the **Profile.scan()** example.

6.26.2 Profile.read() — read a profile of a sequence

`read(file, profile_format)`

This command will read a profile from a specified file. Two formats are supported: `TEXT` and `BINARY`.

For the format of text files, see Section B.3. Binary format files are standard HDF5 files (see Section B.4).

For `TEXT` files, file can be either a file name or a readable file handle (see **modfile.File()**). For `BINARY` files, it must be a file name.

Example: examples/commands/read_profile.py

```
# Example file for: Profile.read(), Profile.to_alignment()

from modeller import *

env = Environ()

# Create a new, blank, profile
prf = Profile(env)

# Read in the profile file
prf.read(file='toxin.prf', profile_format='TEXT')

# Convert the profile to alignment
aln = prf.to_alignment()

# Write out the alignment
aln.write(file='readprofile.pir', alignment_format='PIR')
```

6.26.3 Profile.write() — write a profile

`write(file, profile_format)`

This command will write a profile to a specified file, together with a number of variables that are associated with the profile in the memory. Two formats are supported: `TEXT` and `BINARY`.

For `TEXT` files, `file` can be either a file name or a writeable file handle (see **modfile.File()**). For `BINARY` files, it must be a file name.

Example: examples/commands/aln_to_prof.py

```
from modeller import *
env = Environ()

# Read in the alignment file
aln = Alignment(env)
aln.append(file='toxin.ali', alignment_format='PIR', align_codes='ALL')

# Convert the alignment to profile format
prf = aln.to_profile()

# Write out the profile

# in text file
prf.write(file='alntoprof.prf', profile_format='TEXT')

# in binary format
prf.write(file='alntoprof.bin', profile_format='BINARY')
```

6.26.4 Profile.to_alignment() — profile to alignment

`to_alignment()`

Output: alignment

This command will convert a profile that is in memory into the alignment format (see Section B.1). The function of this command is complimentary to **Alignment.to_profile()**. The generated alignment is returned.

Note: Not all information of a 'PIR' format is encoded in a profile. (See **Profile.read()**). So converting a profile to an alignment may need manual attention to ensure that the alignment is useful for other routines.

Example: examples/commands/read_profile.py

```
# Example file for: Profile.read(), Profile.to_alignment()

from modeller import *

env = Environ()

# Create a new, blank, profile
prf = Profile(env)

# Read in the profile file
prf.read(file='toxin.prf', profile_format='TEXT')

# Convert the profile to alignment
aln = prf.to_alignment()
```

```
# Write out the alignment
aln.write(file='readprofile.pir', alignment_format='PIR')
```

6.26.5 Profile.scan() — Compare a target profile against a database of profiles

```
scan(profile_list_file, matrix_offset=0.0, profile_format='TEXT', rr_file='$(LIB)/as1.sim.mat',
gap_penalties_1d=(-900.0, -50.0), matrix_scaling_factor=0.0069, max_aln_evalue=0.1,
aln_base_filename='alignment', score_statistics=True, output_alignments=True,
output_score_file=None, pssm_weights_type='HH1', summary_file='ppscan.sum',
ccmatrix_offset=-200, score_type='CCMAT', psm=None)
```

This command scans the given target profile against a database of template profiles and reports significant alignments; the target profile should have been read previously with the **Profile.read()** command.

All the profiles listed in profile_list_file should be in a format that is understood by **Profile.read()**.

The profile_list_file should contain absolute or relative paths to the individual template profiles, one per line.

The template profiles can also be assembled into a PSSM database, that can then be read in for scanning. The PSSM database can be created using the **Environ.make_pssmdb()** command.

For the sake of both efficiency and speed, it is recommended to read in the template profiles as a database. (See example). The entire PSSM database, consisting of tens of thousands of PSSMs, can be read into the memory of an average PC.

See documentation under **Profile.read()** for help on profile_format.

rr_file is the residue-residue substitution matrix to use when calculating the position-specific scoring matrix (PSSM). The current implementation is optimized only for the BLOSUM62 matrix.

gap_penalties_1d are the gap penalties to use for the dynamic programming. matrix_offset is the value to be used to offset the substitution matrix (used in PSSM calculation). ccmatrix_offset is used to offset the scoring matrix during dynamic programming. The most optimal values for these parameters are:

matrix_offset = -450 (for the BLOSUM62 matrix) ccmatrix_offset = -100 gap_penalties_1d = (-700, -70)

max_aln_evalue sets the threshold for the E-values. Alignments with e-values better than the threshold will be written out.

aln_base_filename sets the base filename for the alignments. The output alignment filenames will be of the form ALN_BASE_FILENAME_XXXX.ali. The XXXX is a 4-digit integer (prefixed with sufficient zeroes) that is incremented for each alignment. For example, alignment_0001.ali

score_statistics is a flag that triggers the calculation of e-values. If set to `False`, the significance estimates for the alignments will not be calculated. The calculation of alignment significance is similar to that used for **Profile.build()**. This option can be useful when there are only a very small number of template profiles in profile_list_file, insufficient to calculate reliable statistics. Also see **Profile.build()**.

output_score_file is the name of a file into which to write the raw alignment scores, zscores and e-values for all the comparisons. (If it is set to `None`, no such output is written.) The various columns in the output file correspond to the following:

1. Index of the database profile
2. File name of the database profile
3. Length of the database profile
4. Logarithm of the length of the database profile
5. Alignment score
6. Length normalized z-score of the alignment

7. E-Value of the alignment

summary_file is the name of a file into which to output a summary of all the significant alignments. (If it is set to None, no such output is written.) The format of the summary file is the following:

1. File name of target profile (empty if the profile was created with **Alignment.to_profile()**)
2. Length of target profile
3. Number of the first aligned residue of the target profile
4. Number of the last aligned residue of the target profile
5. File name of the database profile
6. Length of the database profile
7. Number of the first aligned residue of the database profile
8. Number of the last aligned residue of the database profile
9. Number of equivalent positions in the alignment
10. Alignment score
11. Sequence identity of the alignment
12. Length normalized z-score of the alignment
13. E-Value of the alignment
14. Alignment file name

If output_alignments is set to False, alignments will not be written out.

In addition, the following details about every significant alignment is also written out to the log file (look for lines beginning with '>'):

1. File name of target profile (empty if the profile was created with **Alignment.to_profile()**)
2. File name of the database profile
3. Length of the database profile
4. Alignment score
5. Sequence identity of the alignment
6. Length normalized z-score of the alignment
7. E-Value of the alignment

Example: examples/commands/ppscan.py

```
# Example for: Profile.scan()

from modeller import *

env = Environ()

# First create a database of PSSMs
env.make_pssmdb(profile_list_file = 'profiles.list',
                matrix_offset     = -450,
                rr_file           = '${LIB}/blosum62.sim.mat',
                pssmdb_name       = 'profiles.pssm',
                profile_format    = 'TEXT',
                pssm_weights_type = 'HH1')

# Read in the target profile
prf = Profile(env, file='T3lzt-uniprot90.prf', profile_format='TEXT')
```

```
# Read the PSSM database
psm = PSSMDB(env, pssmdb_name = 'profiles.pssm', pssmdb_format = 'text')

# Scan against all profiles in the 'profiles.list' file
# The score_statistics flag is set to false since there are not
# enough database profiles to calculate statistics.
prf.scan(profile_list_file = 'profiles.list',
         psm               = psm,
         matrix_offset     = -450,
         ccmatrix_offset   = -100,
         rr_file           = '${LIB}/blosum62.sim.mat',
         gap_penalties_1d  = (-700, -70),
         score_statistics  = False,
         output_alignments = True,
         output_score_file = None,
         profile_format    = 'TEXT',
         max_aln_evalue    = 1,
         aln_base_filename = 'T3lzt-ppscan',
         pssm_weights_type = 'HH1',
         summary_file      = 'T3lzt-ppscan.sum')
```

6.26.6 Profile.build() — Build a profile for a given sequence or alignment

`build(sdb, gap_penalties_1d=(-900.0, -50.0), matrix_offset=0.0, rr_file='$(LIB)/as1.sim.mat', n_prof_iterations=3, max_aln_evalue=0.1, matrix_scaling_factor=0.0069, check_profile=True, output_score_file=None, gaps_in_target=False, score_statistics=True, pssm_weights_type='HH1', pssm_file=None, window_size=1024)`

This command iteratively scans a database of sequences to build a profile for the input sequence or alignment. The command calculates the score for a Smith-Waterman local alignment between the input sequence and each of the sequences in the database. The significance of the alignment scores (e-values) are calculated using a procedure similar to that described by [Pearson, 1998].

Alignments with e-values below max_aln_evalue are then added to the current alignment. A position-specific scoring matrix is then calculated for the current alignment and is used to search the sequence database. This procedure is repeated for n_prof_iterations or until there are are no significant alignments below the threshold, whichever occurs first.

The initial sequence or alignment can be read in either in the profile format, with **Profile.read()**, or as an alignment using **Alignment.append()**. In the latter case, the alignment has to be converted to the profile format using **Alignment.to_profile()**.

The output contains a multiple sequence alignment (assembled) of all the homologues of the input sequence found in the database. The output can be formatted as a profile with **Profile.write()** or converted into any of the standard alignment formats using **Profile.to_alignment()**. It can then be written out to a file with **Alignment.write()**.

The fit between the observed and theoretical distributions of the z-scores is calculated after each iteration and is reported in the log file. The fit is calculated using the Kolmogorov-Smirnov D-statistic. If the check_profile flag is set to `True`, then the command will not proceed if the fit deviates by more than 0.04 (D-statistic).

By default, regions of the alignment that introduce gaps in the target sequence are ignored (deleted) in the final multiple alignment. But if gaps_in_target is set to `True`, then the gaps are retained. (See below for comments).

The scores of each alignment between the input sequence and each database sequence, from all iterations, will be written out to the file specified in output_score_file (or if this is `None`, no such output will be written).

If pssm_file is set to a file name, the PSSM (position specific scoring matrix) calculated for each iteration is written to that file.

Comments:

1. The procedure has been optimized only for the BLOSUM62 similarity matrix.
2. The dynamic programming algorithm has been optimized for performance on Intel Itanium2 architecture. Nevertheless, the calculation is sufficiently CPU intensive. It takes about 20 min for an iteration, using an input sequence of 250aa against a database containing 500,000 sequences on an Itanium2 machine. It could take much longer on any other machine.
3. It is advisable to have gaps_in_target set to `False` when scanning against large databases, to avoid the local alignments inserting a huge number of gaps in the final alignments.
4. The statistics will not be accurate if the database does not have sequences that represent the entire range of lengths possible. In extreme cases, where statistics cannot be calculated at all, a `StatisticsError` will be raised.
5. The method can be used for fold-assignment by first building a profile for the target sequence by scanning against a large non-redundant sequence database (like swissprot) and then using the resulting profile to scan once against a database of sequences extracted from PDB structures. gaps_in_target can be set to `True` in the second step to get the complete alignments that can then be used for modeling.

See **SequenceDB.read()** for a discussion of the window_size parameter.

Example: examples/commands/build_profile.py

```
from modeller import *
log.verbose()
env = Environ()

#-- Prepare the input files

#-- Read in the sequence database
sdb = SequenceDB(env)
sdb.read(seq_database_file='pdb95.fsa', seq_database_format='FASTA',
         chains_list='ALL', minmax_db_seq_len=(1, 40000), clean_sequences=True)

#-- Write the sequence database in binary form
sdb.write(seq_database_file='pdb95.bin', seq_database_format='BINARY',
          chains_list='ALL')

#-- Now, read in the binary database
sdb.read(seq_database_file='pdb95.bin', seq_database_format='BINARY',
         chains_list='ALL')

#-- Read in the target sequence/alignment
aln = Alignment(env)
aln.append(file='toxin.ali', alignment_format='PIR', align_codes='ALL')

#-- Convert the input sequence/alignment into
#   profile format
prf = aln.to_profile()

#-- Scan sequence database to pick up homologous sequences
prf.build(sdb, matrix_offset=-450, rr_file='${LIB}/blosum62.sim.mat',
          gap_penalties_1d=(-500, -50), n_prof_iterations=5,
          check_profile=False, max_aln_evalue=0.01, gaps_in_target=False)

#-- Write out the profile
```

```
prf.write(file='buildprofile.prf', profile_format='TEXT')

#-- Convert the profile back to alignment format
aln = prf.to_alignment()

#-- Write out the alignment file
aln.write(file='buildprofile.ali', alignment_format='PIR')
```

6.26.7 PSSMDB() — create a new PSSM database

`PSSMDB(env, **vars)`

This creates a new `PSSMDB` object. If you give any arguments to the `PSSMDB()` constructor, they are passed to **PSSMDB.read()**, to read in the specified database. See the **Profile.scan()** example.

6.26.8 PSSMDB.read() — read a PSSM database from a file

`read(pssmdb_name, pssmdb_format)`

This reads in a PSSM database from a file. See the **Profile.scan()** example.

6.27 The `SequenceDB` class: using sequence databases

The `SequenceDB` class holds a database of sequences. Such a database is similar to a multiple sequence alignment, but contains less auxiliary information (for example, no sequence may contain gaps). This requires less memory than a true alignment, and is thus more suited for large databases of sequences. Such a database can be scanned for matches to an input sequence with **SequenceDB.search()**, used to build sequence profiles with **Profile.build()** or filtered by given criteria with **SequenceDB.filter()**.

6.27.1 SequenceDB() — create a new sequence database

```
SequenceDB(env, **vars)
```

This creates a new, empty, sequence database. If you give any arguments to this constructor, they are passed to **SequenceDB.read()**, to read in an initial database. See the **SequenceDB.filter()** example.

6.27.2 SequenceDB.read() — read a database of sequences

```
read(chains_list, seq_database_file, seq_database_format, clean_sequences=True,
minmax_db_seq_len=(0, 999999))
```

This command will read a database of sequences, either in the `PIR`, `FASTA`, or `BINARY` format.

If the format is `PIR` or `FASTA`:

- It is possible to clean all sequences of non-standard residue types by setting clean_sequences to `True`.
- Sequences shorter than minmax_db_seq_len[0] and longer than minmax_db_seq_len[1] are eliminated.
- Only sequences whose codes are listed in the chains_list file are read from the seq_database_file of sequences. If chains_list is `all`, all sequences in the seq_database_file file are read in, and there is no need for the chains_list file.

For the `PIR` and `FASTA` formats, make sure the order of sequences in the chains_list and seq_database_file is the same for faster access (there can of course be more sequences in the sequence file than there are sequence codes in the codes file).

Additionally, if the sequences are in 'PIR' format, then the protein type and resolution fields are stored in the database format. (see Section B.1 for description of 'PIR' fields).

The protein type field is encoded in a single letter format. 'S' for sequence and 'X' for structures of any kind. This information is transferred to the profile arrays when using **Profile.build()**. (See also **Profile.read()**).

The resolution field is used to pick representatives from the clusters in **SequenceDB.filter()**.

None of the options above apply to the `BINARY` format, which, in return, is very fast. Binary files are standard HDF5 files (see Section B.4).

When using PIR or FASTA files, the entire sequence database is stored in memory. Thus, extremely large databases, such as UniProt, will require your computer to have a large amount of system memory (RAM) available, to store the database and to provide working space. In cases where the database requires more than 2 gigabytes of memory, you will also need to use a 64-bit machine, such as Alpha, Itanium, or x86_64 (Opteron/EM64T). On the other hand, when using a binary file, only part of the file is read into memory on demand. (Functions which utilize sequence databases have a window_size parameter, which determines how much of the file is read in at a time. A larger window size will generally result in faster execution, at the expense of increased memory use.) Thus, binary files are strongly recommended whenever speed or memory is a concern.

If you are intending to read in a sequence database simply to write it out again in binary format, you should consider using the **SequenceDB.convert()** function instead, as this does not need to keep the whole database in memory.

Example: See **Profile.build()** command.

6.27.3 SequenceDB.close() — close an open database

`close()`

This will close any currently open database, freeing any memory used. (For a binary database, it will also release the HDF5 file handle.) After calling this function, the database will be empty.

Note that the database is automatically closed when its Python object is destroyed.

6.27.4 SequenceDB.write() — write a database of sequences

`write(chains_list, seq_database_file, seq_database_format, window_size=1024)`

This command will write a database of sequences currently in memory, either in the `PIR`, `FASTA`, or `BINARY` format. The chains_list file is written only for the `PIR` or `FASTA` formats.

`BINARY` files are standard HDF5 files, and are machine-independent. Note, however, that they are not compatible with older versions of MODELLER.

See **SequenceDB.read()** for a discussion of the window_size parameter.

Example: See **Profile.build()** command.

6.27.5 SequenceDB.convert() — convert a database to binary format

`convert(chains_list, seq_database_file, seq_database_format, outfile, clean_sequences=True, minmax_db_seq_len=(0, 999999))`

This command will read a database of sequences, in `PIR` or `FASTA` format, and write it out in `BINARY` format. See **SequenceDB.read()** for an explanation of the parameters used. outfile gives the name of the resulting binary file.

The conversion process is done one sequence at a time, so this requires substantially less system memory than **SequenceDB.read()** followed by **SequenceDB.write()**.

Any existing data in the database is discarded by this routine, and the database is empty when the function finishes.

Example: examples/commands/convert_sequence_db.py

```
from modeller import *
log.verbose()
env = Environ()

sdb = SequenceDB(env)
sdb.convert(seq_database_file='pdb95.fsa', seq_database_format='FASTA',
            chains_list='ALL', minmax_db_seq_len=[1, 40000],
            clean_sequences=True, outfile='pdb95.bin')
```

6.27.6 SequenceDB.search() — search for similar sequences

`search(aln, seq_database_file, search_group_list, search_randomizations=0, search_top_list=20, off_diagonal=100, overhang=0, gap_penalties_1d=(-900.0, -50.0), signif_cutoff=(4.0, 5.0), rr_file='$(LIB)/as1.sim.mat', matrix_offset=0.0, fast_search_cutoff=1.0, data_file=False,`

```
search_sort='LONGER', output='LONG', alignment_features='INDICES CONSERVATION',
local_alignment=False, fast_search=False, window_size=1024, io=None, **vars)
```

This command searches a sequence database for proteins that are similar to a given target sequence.

The target sequence should be the only sequence in the provided alignment, aln.

The database of sequences to be scanned against must be read previously by the **SequenceDB.read()** command.

The command uses the dynamic programming method for the best sequence alignment, given the gap creation and extension penalties specified by gap_penalties_1d and residue type scores read from file rr_file. gap_penalties_1d[0] is a gap creation penalty and gap_penalties_1d[1] is a gap extension penalty.

The search_top_list top hits are written to the `log` file at the end. The hits are sorted according to the fractional sequence identity score obtained by dividing the number of identical residue pairs by the length of the longer sequence (search_sort = `'LONGER'`) or the shorter sequence (search_sort = `'SHORTER'`).

The final list of hits contains three different significance values:

1. `SIGNI`. Z-score from sequence randomizations. This is the most accurate significance score, but the slowest one to calculate. For each pairwise comparison, the two sequences are shuffled a specified number of times (search_randomizations) to obtain the mean and standard deviation of "random" scores from which the Z-score for an alignment score of a given pair of sequences is calculated.
2. `SIGNI2`. Z-score for sequence identity from the database scan. After comparison of the target sequence with all sequences in the database is done, the comparisons are sorted by the length of the database sequence. The pairwise sequence identities of the 20 sequences closest in length to the target sequence are used to calculate the average and standard deviation of the percentage sequence identities for subsequent calculation of the Z-score for the percentage sequence identity of a given pairwise alignment.
3. `SIGNI3`. Z-score for alignment score from the database scan. The procedure is the same as for `SIGNI2`, except that the alignment scores are used instead of the pairwise sequence identities.

The calculation of the Z-scores assumes that the random scores are distributed according to the Gaussian distribution, instead of the extreme value distribution [Karlin & Altschul, 1990], which is more correct.

search_randomizations specifies how many alignments of the shuffled sequences are done to calculate the significance score for the overall sequence similarity. If 0, the significance is not calculated. If more than 5 randomizations are done, the significance score, not sequence identity, is used for sorting the hit list.

When fast_search is `True` only those sequences that have a database-scan alignment score significance (`SIGNI3` in output) above fast_search_cutoff are used for the "full" randomization-based significance calculation. Since the mean and the standard deviation of the distribution obtained by randomizing the two compared sequences are much more appropriate than the corresponding quantities for the target/database comparisons, fast_search should be `True` only when you are in a hurry and the database is large.

If data_file is `True` the final results (list of PDB codes with significances, *etc.*) are also written to a separate file `'seqsearch.dat'`.

If output is `'LONG'`, the best alignment for each sequence in the database and its various scores are also written to the `log` file. If output is `'VERY_LONG'`, individual scores obtained for randomized sequences are also written to the `log` file (this is almost never needed).

If the selected significance score is larger than signif_cutoff[0] and not more than signif_cutoff[1] units worse than the best hit, all the members of the same group, as defined in search_group_list, are added to the alignment (the original query sequence is removed). These sequences are taken from seq_database_file, which is often (but not always) the same file previously provided to **SequenceDB.read()**, and must be in PIR format. Subsequent **Alignment.malign()**, **Environ.dendrogram()** and **Alignment.write()** can then be used to write out all related PDB chains aligned to the target sequence.

See **SequenceDB.read()** for a discussion of the window_size parameter.

Example: examples/commands/sequence_search.py

```
# Example for: SequenceDB.search()

# This will search the MODELLER database of representative protein chains
# for chains similar to the specified sequence.

from modeller import *

log.verbose()
env = Environ()

# Read in the sequences of all PDB structures
try:
    sdb = SequenceDB(env, seq_database_file='pdball.pir',
                     seq_database_format='PIR',
                     chains_list='very-short-for-test.cod')
except IOError:
    print("""
Could not read sequence database file. This file is not included by default
in the Modeller distribution, but you can download it from the Modeller
downloads page (https://salilab.org/modeller/supplemental.html).

Note: it is recommended to use Profile.build() rather than SequenceDB.search().
See step 1 of the Modeller basic tutorial at
https://salilab.org/modeller/tutorial/basic.html
""")
    raise

# Read in the query sequence in alignment format
aln = Alignment(env, file='toxin.ali', align_codes='2nbt')

sdb.search(aln, search_randomizations=20, # should use 100 in real life
           seq_database_file='pdball.pir',
           search_group_list='pdb_95.grp',
           off_diagonal=9999, gap_penalties_1d=(-800, -400),
           signif_cutoff=(1.5, 5.0))

aln.malign()
aln.write(file='toxin-search.pap', alignment_format='PAP')
```

6.27.7 SequenceDB.filter() — cluster sequences by sequence-identity

```
filter(seqid_cut, output_grp_file, output_cod_file, gap_penalties_1d=(-900.0, -50.0),
matrix_offset=0.0, rr_file='$(LIB)/as1.sim.mat', max_diff_res=30, window_size=512)
```

This command clusters a set of sequences by sequence identity. The command uses a greedy algorithm: the first sequence in the file becomes the first group representative. All other sequences are compared with this and if they are similar enough, as specified in seqid_cut, they are added as members of this group. These sequences are not used for further comparisons. The next non-member sequence becomes the next group representative and so on.

The initial set of sequences must be read previously by the **SequenceDB.read()** command with seq_database_format being either 'PIR' or 'FASTA'.

rr_file is the residue-residue substitution matrix and matrix_offset its offset. The command only handles similarity matrices for efficiency purposes.

The command uses the Smith-Waterman dynamic programming method (as in **Alignment.align()**) for the best sequence alignment, given the gap creation and extension penalties specified by gap_penalties_1d and residue type scores read from file rr_file. gap_penalties_1d[0] is a gap creation penalty and gap_penalties_1d[1] is a gap extension penalty.

The final list of groups and their members is written out to output_grp_file. The codes of the representative sequences is written out to output_cod_file.

The clustering algorithm evaluates the following conditions in hierarchical order before adding a sequence to a group:

1. The difference in length: If the difference in the number of residues between the group representative and the sequence being compared is greater than max_diff_res, the sequence will not be included into that group.
2. The number of unaligned residues: After the local alignment is performed, a sequence will not be considered for addition into a group unless the difference between the smaller of the two sequences and the number of aligned positions in the alignment is less than max_unaligned_res.
3. Sequence Identity: Finally, if the sequence identity calculated from the alignment is greater than seqid_cut, the sequence is added to a group.

If the initial set of sequences read were in 'PIR' format with values in the resolution field, then the group representative is the sequence with the highest resolution. This is especially useful when clustering sequences from the PDB.

See **SequenceDB.read()** for a discussion of the window_size parameter. Note that this function acts on two regions of the database simultaneously (it does an all-against-all comparison) and so the default window size is half that of other functions.

Example: examples/commands/seqfilter.py

```
from modeller import *

log.verbose()
env = Environ()

sdb = SequenceDB(env, seq_database_file='sequences.pir',
                 seq_database_format='PIR',
                 chains_list='ALL', minmax_db_seq_len=[30, 3000],
                 clean_sequences=True)

sdb.filter(rr_file='${LIB}/id.sim.mat', gap_penalties_1d=[-3000, -1000],
           max_diff_res=30, seqid_cut=95, output_grp_file='seqfilt.grp',
           output_cod_file='seqfilt.cod')
```

6.28 The `Density` class: handling electron microscopy density data

The `Density` class stores all information from an electron microscopy density map file. Protein models can then be docked to this density (using **Density.grid_search()**) to improve their quality.

6.28.1 Density() — create a new density map

```
Density(env, **vars)
```

This creates a new, empty, density map. If you give any arguments to the `Density()` constructor, they are passed to **Density.read()**, to read in an initial density. See the **Density.grid_search()** example.

6.28.2 Density.resolution — Map resolution

The resolution, as set by **Density.read()**. Can be overridden after reading the map.

6.28.3 Density.sigma_factor — Sigma factor

The sigma factor determines the width of the Gaussian distribution used to represent each atom in the model. The default value of $1/(2 * \sqrt(2log2))$ (0.4246609) results in the width of the Gaussian at half the maximum height being equal to Density.resolution.

6.28.4 Density.voxel_size — Map voxel size

The map voxel size, as set by **Density.read()**.

6.28.5 Density.px — Origin of the map

The x coordinate of the origin of the map (in angstroms), as set by **Density.read()**.

6.28.6 Density.py — Origin of the map

The y coordinate of the origin of the map (in angstroms), as set by **Density.read()**.

6.28.7 Density.pz — Origin of the map

The z coordinate of the origin of the map (in angstroms), as set by **Density.read()**.

6.28.8 Density.grid — Density values

The actual density values, as a 3D array. This is similar to a `numpy` array in that it is accessed with a 3-tuple (*e.g.*, `d.grid[1,0,3]`) and has a `shape` member that gives the dimensions of the grid (again as a 3-tuple). The order of the dimensions is z, y, x.

6.28.9 Density.read() — read an EM (electron microscopy) density map file

```
read(file, resolution, em_map_size=0, voxel_size=0.0, em_density_format='XPLOR',
filter_type='NONE', filter_values=(0.0, 0.0), density_type='SPHERE', px=None, py=None, pz=None,
cc_func_type='CCF')
```

This command reads a density map from file, which should be provided as a grid of intensities, in the X-PLOR [Brünger, 1992], CCP4, or MRC format. (Note that CCP4 and MRC files are currently read in the same way, since the formats are very similar.) file can be a filename or a readable file handle (see **modfile.File()**).

Note that while non-cubic grids can be read in, **Density.grid_search()** only works with cubic grids.

The size of the grid and the map voxel size (voxel_size) are both taken from the density map file itself. If, however, em_map_size is specified and is non-zero, the grid must be cubic and all dimensions must match this value. If voxel_size is specified and differs from that given in the file header, the user-specified value is used in preference.

The density map resolution is given by resolution.

px, py, and pz specify the origin of the map, in angstroms. If not given, it is read from the MRC or CCP4 map header (X-PLOR files do not include this information, so the origin will default to zero in this case.)

When fitting the probe into the EM grid, the probe structure is converted first into probe density, by using the function indicated in the density_type variable. Each atom can be represented by one of several atomic density functions, including, the uniform sphere model (`'SPHERE'`), the Gaussian function (`'GAUSS'`), a normalized Gaussian function (`'NORM'`), a hybrid Gaussian/sphere model (`'HYBRID'`), and an interpolation to the closest point on the grid (`'TRACE'`). The recommended function is `'SPHERE'`.

filter_type is used to filter the values of the EM density during this calculation. Filters that can be used are: `'THRESHOLD'` for a lower threshold (any density value below the first value set in filter_values will be set to 0); `'SQUARE'` for a square filter; and `'LAPLACIAN'` for a Laplacian filter. `'NONE'` is the default option, and means that no filter is used.

When calculating the cross-correlation coefficient between a probe model and the density map, ccf_func_type specifies if you want the normalized cross-correlation coefficient (`'CCF'`), or the local cross-correlation coefficient (`'LCCF'`).

Example: See **Density.grid_search()** command.

6.28.10 Density.grid_search() — dock a structure into an EM (electron microscopy) density map

```
grid_search(em_pdb_name, chains_num, em_density_format='XPLOR', num_structures=1,
dock_order='INPUT', start_type='CENTER', translate_type='NONE', number_of_steps=1,
angular_step_size=0, temperature=293.0, best_docked_models=1, em_fit_output_file='em_fit.out')
```

Requirements: PDB files

This command docks a structure of a protein/domain (probe) into a given cubic EM density map. See original paper for the description of the method and the most recommended protocols [Topf *et al.*, 2005].

Note that this only works with cubic density maps.

The probe is specified by the variable em_pdb_name. Before starting the protocol, the probe is positioned on the EM density grid based on the start_type variable:

- `'CENTER'` will translate the center of mass of the probe to the center of the grid.
- `'ENTIRE'` will divide the grid into cells similar in volume to the probe, and will translate the probe to the center of each of these cells consecutively.
- `'SPECIFIC'` will use the coordinates specified by the user (the input PDB coordinates) as a starting position.

The best fit between the probe and the EM density map is obtained by changing the position of the model so as to maximize the cross-correlation between the probe density and the EM density. See **Density.read()** for the density fitting procedure, controlled by the density_type variable.

The optimization of the cross-correlation score is performed by a 6D search of the probe on the EM grid, based on the variable translate_type. If 'NONE' is specified, only a rotational search over the three rotational Euler angles (ϕ, θ, ψ) is performed, with no translations. angular_step_size gives the maximal step size of searching for all combinations of Euler angles, which is recommended to be 30 degrees. The best fit from this coarse search is refined by a finer local search in all three Euler angles. For a protein of 150 residues this calculation typically takes less than 0.5 minutes on a 3.0 GHz Intel Xeon processor.

If translate_type is 'RANDOM', a Monte Carlo (MC) optimization is performed, and the number of MC steps has to be specified (by number_of_steps). A single MC step consists of (i) a random translation of the probe for one voxel on the EM grid, (ii) a search for the three Euler angles that maximize cross-correlation score, and (iii) an application of the Metropolis criterion [Metropolis *et al.*, 1953]. The temperature used for the Metropolis criterion is specified in temperature (typically about 5000 units). This calculation typically takes about 1-2 minutes.

When translate_type is 'EXHAUSTIVE', a local search is performed with the probe on the grid. The optimal orientations at its original position and all 26 (*i.e.*, 3x3x3 - 1) neighboring grid points are obtained successively by enumerating all three Euler angles. A Monte Carlo criterion is applied to each one of these 27 optimal orientations (usually with temperature lower than 5000, but this number has to be adjusted according to the optimization). number_of_steps gives the number of steps for which this process is repeated (typically 25 times). When the EM density map covers only the probe model and start_type is either 'CENTER' or 'SPECIFIC', this protocol can be used for a translational and rotational refinement of the initial placement of the model on the grid. When start_type is 'ENTIRE' this protocol will be applied for a local search only in those cells where the randomly oriented probe gives a positive cross-correlation score. The calculation typically takes about 10-15 minutes.

best_docked_models specifies how many best-fitted models should be saved by the program. This becomes more important at decreasing resolutions, as the best solution will not necessarily have the highest cross-correlation score.

em_fit_output_file names an output file which will be used to record the progress of the optimization.

Output: **targ_1_1.pdb** The fitted coordinates file. The name is formed by taking the first 4 letters from your input PDB file and adding '_1_1.pdb'. If best_docked_models = 2, you will get also targ_1_2.pdb, and so on. If you use the option start_type=ENTIRE, it will add targ_2_1.pdb, targ_2_2.pdb and so on.

bestCC_targ The results file which reports the CCF, with the number of required solutions (as indicated in best_docked_models).

targ_best.MRC The fitted structure converted to a map.

targ_init_1.pdb The initial structure superposed in the center of mass (if start_type=CENTER), or in a specific location (if start_type=SPECIFIC).

targ_init.MRC The initial structure converted to a map.

EM_map.MRC This is your original density map. It's a test to see if MODELLER is reading it correctly.

Example: examples/commands/em_grid_search.py

```
from modeller import *

log.verbose()
env = Environ()

den = Density(env, file='1cuk-a2.mrc', em_density_format='MRC',
              voxel_size=1., resolution=8., em_map_size=40,
              cc_func_type='CCF', density_type='SPHERE')

den.grid_search(em_density_format='MRC', num_structures=1,
                em_pdb_name=['1cuk-a2.pdb'], chains_num=[1],
                start_type='CENTER', number_of_steps=1, angular_step_size=30.,
                temperature=0., best_docked_models=1,
                em_fit_output_file='test-cr.log')
```

6.29 The SAXSData class: using small-angle X-ray (SAXS) data

The SAXSData class is used to store small-angle X-ray (SAXS) data.

6.29.1 SAXSData() — create a new SAXSData structure

```
SAXSData(env, **vars)
```

This creates a new SAXSData object. Use **SAXSData.ini_saxs()** to initialize it.

6.29.2 SAXSData.ini_saxs() — Initialization of SAXS data

```
ini_saxs(atmsel, filename='$(LIB)/formfactors-int_tab_solvation.lib', s_min=0.0, s_max=2.0,
maxs=100, nmesh=100, natomtyp=15, represtyp='heav', wswitch='uniform', s_hybrid=0.0, s_low=0.0,
s_hi=2.0, spaceflag='real', rho_solv=0.334, use_lookup=True, nr=5000, dr=0.1, nr_exp=300,
dr_exp=1.0, use_offset=False, use_rolloff=False, use_conv=False, mixflag=False, pr_smooth=False)
```

Routine to initialize the SAXSData structure. Here the sampling in reciprocal space needs to be specified; currently only equidistant sampling is possible. Moreover, the parameters for the scoring function and for its computation are set.

s_min and s_max specify the minimum and maximum frequency in reciprocal space, in $Å^{-1}$. maxs gives the maximum number of frequencies, and nmesh the actual number (which must be less than maxs).

natomtyp gives the number of "atoms", *i.e.* scattering centers. represtyp specifies the representation : 'heav', 'allh', or 'CA'. filename is the name of the library for formfactors. wswitch is the character for filter of scoring function: 'unity', 'sq', or 'hybrid'. If 'hybrid', then s_hybrid is the frequency above which s^2 weighting is applied. s_low and s_hi give the lower and upper cutoff for the bandpass filter in $Å^{-1}$. spaceflag specifies how $I(s)$ should be computed. 'real' space via $P(r)$ or 'reciprocal'. 'real' is more than a magnitude faster but less accurate for high resolution ($s > 0.5$).

rho_solv gives the electron density of solvent, in $e^{-}Å^{-3}$. (The default 0.334 corresponds to H_2O.) use_lookup, if True, uses lookup tables for SINC and COS functions, giving a significant increase in speed for 'reciprocal' mode.

nr gives the number of points for $P(r)$ sampling, and dr the spacing of these points in Å. nr_exp gives the number of points for $P_{exp}(r)$ sampling, and dr_exp their spacing.

If use_offset is True then allowance is made for an additive constant in the experimental spectrum. If use_rolloff is True, allowance is made for Gaussian rolloff in the model spectrum. If use_conv is True, the spectrum is multiplied with the formfactor of nitrogen (3Å) spectrum. If mixflag is True then more than one conformation is modeled simultaneously. If pr_smooth is True smoothing of p(r) is done.

6.29.3 SAXSData.saxs_read() — Read in SAXS data

```
saxs_read(filename)
```

Read in SAXS data. Make sure that sampling of s is the same as specified in **SAXSData.ini_saxs()**. The file is text format, containing 3 columns: spatial frequency s in $Å^{-1}$, Intensity, and experimental error (if determined). Comments start with '#'.

6.29.4 SAXSData.saxs_pr_read() — Read in P(r) data

```
saxs_pr_read(filename)
```

Read in P(r) data. The file is text format, containing 3 columns: radius s in Å, $P(r)$, and experimental error (if determined). Comments start with '#'.

See also **Model.saxs_pr()**.

6.30 The `info` object: obtaining information about the Modeller build

The `info` object holds information about the current Modeller build (most of this can also be found in the first few lines of the log file). This information is useful when reporting bugs, or for writing scripts which require a certain Modeller version.

6.30.1 info.version — the full Modeller version number

This is the current version, as a string. This contains all the information returned by info.version_info and info.build_date, but in a less easily machine-readable form.

6.30.2 info.version_info — the version number, as a tuple

This is only the version number, as a Python tuple containing the major and minor version numbers. For example, version 8v0 would return `(8, 0)`. (SVN builds always return `'SVN'` instead.)

6.30.3 info.build_date — the date this binary was built

This is the date on which this Modeller binary was built, as a string in 'YYYY/MM/DD HH:MM:SS' format.

6.30.4 info.exe_type — the executable type of this binary

This is a string identifying the machine and Fortran compiler type of this Modeller binary.

6.30.5 info.debug — this binary's debug flag

This is `True` if this binary was built with compiler debugging flags, or `False` if full code optimization was turned on. Debugging builds generally run slower than optimized builds, so release binaries will invariably return `False`.

6.30.6 info.bindir — Modeller binary directory

This is the directory in which Modeller binaries are installed.

6.30.7 info.time_mark() — print current date, time, and CPU time

```
time_mark()
```

This prints to the log file the total CPU time used in this run (in seconds) and the CPU time used since the last time this command was called. The date and time on which the run was started, plus the current date and time, are also printed, in 'YYYY/MM/DD HH:MM:SS' format.

6.30.8 info.jobname — name of the current job

This is the name of the current Modeller job, used to name log files. Usually, it is the name of the script, without any file extension, or `'(stdin)'` if input is being piped into the program.

6.31 The `log` object: controlling the amount of output

The `log` object allows you to control the amount of information output to the MODELLER log file. (It is also used internally to divert the system standard output, *e.g.* from the Python `'print'` statement, to the log file.)

6.31.1 log.level() — Set all log output levels

```
level(output=1, notes=0, warnings=0, errors=1, memory=0)
```

This sets all five of MODELLER's log outputs. (This is very similar in operation to the old OUTPUT_CONTROL variable.) Each argument should be an integer, either 0 to display no output, or 1 to display all output. An exception is the memory argument, which can also be set to 2 to display additional dynamic memory information.

6.31.2 log.none() — display no log output

```
none()
```

This instructs MODELLER to display no log output.

6.31.3 log.minimal() — display minimal log output

```
minimal()
```

This instructs MODELLER to only display important outputs, and errors.

6.31.4 log.verbose() — display verbose log output

```
verbose()
```

This instructs MODELLER to display all log output.

6.31.5 log.very_verbose() — display verbose log output, and dynamic memory information

```
very_verbose()
```

This instructs MODELLER to display all log output. Additionally, a breakdown of all dynamic memory used by MODELLER is displayed every time memory is allocated or freed. (Note that some information is held statically, and this is not tracked. Some routines require additional temporary storage, which is also not listed here. Finally, memory used by any Python variables is not accounted for.)

6.32 The `modfile` module: handling of files

The `modfile` module contains routines for dealing with files which are used by MODELLER. More complete facilities for most of these functions are available within the standard `'os'` Python module; however, the `modfile` module is provided for compatibility with old TOP scripts, and for systems with incomplete Python installations.

6.32.1 modfile.default() — generate an 'automatic' filename

```
default(root_name='undf', file_id='X', id1=1, id2=1, file_ext='')
```

This returns a generated file name, as `(root_name)(file_id)(id1)(id2)(file_ext)`, where id1 and id2 are printed as 4-digit numbers, padded with zeroes if necessary. For example, `'2ptn.B99990001.pdb'` results from root_name = `'2ptn'`, file_id = `'.B'`, id1 = 9999, id2 = 1, and file_ext = `'.pdb'`. This mimics the 'automatic' filename generation of previous versions of MODELLER when filenames were set to `'default'` or `'${DEFAULT}'`.

6.32.2 modfile.delete() — delete a file

```
delete(file)
```

This command deletes the named file.

6.32.3 modfile.inquire() — check if file exists

```
inquire(file)
```

Output: file_exists

This command returns 1 if the specified file exists, or 0 otherwise.

6.32.4 modfile.File() — open a handle to a MODELLER file

```
File(filename, mode='r')
```

This opens a file and returns a handle object, which can be used for methods that need an open file, such as **Alignment.read_one()**. Many methods (such as **Model.write()**) can also be given a writeable file handle, to have them append their output to that file rather than creating a new one. (They can also be given a Python filelike object, such as `sys.stdout` or `io.StringIO`, to write to a Python file; however, this is less efficient as it must call Python functions to do the IO.) Similarly, many methods (such as **Model.read()** can be given a readable file handle, or a Python filelike object.

The file is closed automatically when the handle object is deleted, or explicitly by calling its `close` method.

The mode argument functions similarly to that used by C or Python; *i.e.*, the following modes are acceptable: `'r'`, `'w'`, `'rb'` and `'wb'`, to open a file for reading in text mode, writing in text mode, reading in binary mode, or writing in binary mode, respectively. Note that while only Windows operating systems make a distinction between text and binary mode, MODELLER will do some additional checks on text format files to catch common mistakes (*e.g.*, trying to read a Unicode rather than plain text file) so you should use the `'b'` suffix on all platforms if you are using binary files.

Example: See **Alignment.read_one()** command.

6.33 The scripts module: utility scripts

The scripts module contains utility scripts for some common tasks.

6.33.1 cispeptide() — creates cis-peptide stereochemical restraints

```
cispeptide(rsr, atom_ids1, atom_ids2)
```

This generates restraints to constrain the given residue pair to cis-peptide conformation. Any existing trans restraints are first removed.

Example: See **Restraints.add()** command.

6.33.2 complete_pdb() — read a PDB, mmCIF, or BinaryCIF file, and fill in any missing atoms

```
complete_pdb(env, filename, special_patches=None, transfer_res_num=False, model_segment=None,
patch_default=True)
```

Output: mdl

This reads in a PDB, mmCIF, or BinaryCIF file specified by filename, and fills in any missing atoms using internal coordinates. If you want to read in a PDB, mmCIF, or BinaryCIF file from PDB or generated from an experiment or some other program, you should use this routine in preference to **Model.read()**, which does not handle missing atoms.

If special_patches is not None, it should be a Python function which takes a single argument (the model). It is called prior to filling in missing coordinates. This can be used to adjust the topology, *e.g.* by adding disulfide bridges. If you wish to patch terminal residues, you should also set patch_default to False to turn off the default patching.

If transfer_res_num is True, the residue numbering from the original PDB/mmCIF/BinaryCIF is retained (by default, residues are renumbered from 1, and chains are labeled from A).

If model_segment is set, it is used to read a subset of the residues from the file, just as in **Model.read()**.

In order for this routine to work correctly, you should first have read in topology and parameter libraries. The new Model object is returned on success.

Example: See **Selection.energy()**, **ConjugateGradients()** command.

6.34 The salign module: high-level usage of SALIGN

The salign module contains high-level methods to simplify the usage of SALIGN (see **Alignment.salign()**) for some common tasks.

6.34.1 iterative_structural_align() — obtain the best structural alignment

```
iterative_structural_align(aln, align_block=0)
```

Given an initial alignment of two or more structures, this will run **Alignment.salign()** in an iterative fashion, exploring a range of suitable parameter values, to determine the best structural alignment, as detailed in [Madhusudhan *et al.*, 2009].

Example: examples/salign/salign_iterative.py

```
# Illustrates the SALIGN iterative multiple structure alignment
from modeller import *
import modeller.salign

log.none()
env = Environ()
env.io.atom_files_directory = ['.', '../atom_files']

aln = Alignment(env)
for (code, chain) in (('1is4', 'A'), ('1uld', 'D'), ('1ulf', 'B'),
                      ('1ulg', 'B'), ('1is5', 'A')):
    mdl = Model(env, file=code, model_segment=('FIRST:'+chain, 'LAST:'+chain))
    aln.append_model(mdl, atom_files=code, align_codes=code+chain)

modeller.salign.iterative_structural_align(aln)

aln.write(file='1is3A-it.pap', alignment_format='PAP')
aln.write(file='1is3A-it.ali', alignment_format='PIR')
```

6.35 Parallel job support

The `parallel` module provides methods to parallelize MODELLER calculations over multiple processors. This may be employed to make use of more than one CPU on a multi-processor machine, or of several nodes in a cluster. Each remote process is termed a 'worker', while the process which starts and controls the parallel job is termed the 'manager'.

Two methods for accessing the workers from the manager are provided; the first is a task-based interface (**Job.queue_task()**, **Job.run_all_tasks()** and **Job.yield_tasks_unordered()**) which is largely transparent in use and fault-tolerant, while the second is a lower-level worker-manager message-passing interface (**Job.start()**, **Communicator.send_data()**, **Communicator.get_data()** and **Worker.run_cmd()**), which is similar to MPI (although coarse-grained) and requires you to handle errors yourself. It is recommended that you use the task interface unless you require a large amount of message passing.

This module relies on several modules in the Python standard library. These modules are not provided with MODELLER, so you must additionally install Python if it is not already present on your system. It is also experimental, and may need modification to work with your own networking or parallel system.

6.35.1 Job() — create a new parallel job

`Job(seq=(), modeller_path=None, host=None)`

This creates a new `Job` object, used to keep track of multiple worker processes. It is initially empty, but acts just like an ordinary Python list, so you can add or remove `Worker` objects (see below) using ordinary list operations (*e.g.*, append, del). Also, if you provide a list of suitable worker objects to Job(), they will automatically be added.

Each worker runs a MODELLER process. The system attempts to start this process in the same way as the MODELLER script used for the manager. If the manager is run using your machine's system Python, the worker is started by running `'python modlib/modeller/parallel/modworker.py'`, while if the manager was started using the `'mod10.7'` script, the worker will be too. In some cases, it may get this command line wrong, in which case you can specify the command explicitly using the modeller_path variable. For example, set it to `'mod10.7'` to force it to use the version of Python built in to MODELLER rather than the system Python.

Each worker, when started, tries to connect back over the network to the manager node. By default, they try to use the fully qualified domain name of the machine on which you create the `Job` object (the manager). If this name is incorrect (*e.g.*, on multi-homed hosts) then specify the true hostname with the host parameter. You can also set host to `'localhost'` if your machine does not have network connectivity and/or you are running only local workers.

Each worker will run in the same directory as the manager, so will probably fail if you do not have a shared filesystem on all nodes. The output from each worker is written to a logfile called `'${JOB}.workerN'` where `'${JOB}'` is info.jobname and `'N'` is the number of the worker, starting from zero.

Job.worker_startup_commands is a Python list, initially empty, of Python commands that will be run on each worker when it is started up. You can add your own worker initialization by adding to this list.

Once you have created the job, to use the task interface, submit one or more tasks with **Job.queue_task()**, and then run the tasks with **Job.run_all_tasks()** or **Job.yield_tasks_unordered()**.

To use the message-passing interface, first start all workers with **Job.start()**, and then use **Communicator.send_data()**, **Communicator.get_data()** and **Worker.run_cmd()** to pass messages and commands.

Example: See **Job.start()**, **Job.run_all_tasks()** command.

6.35.2 SGEPEJob() — create a job using all Sun GridEngine (SGE) worker processes

`SGEPEJob(seq=(), modeller_path=None, host=None)`

This functions identically to **Job()**, above, but automatically adds workers for every node in a Sun GridEngine (SGE) job using an SGE parallel environment. This is done by parsing the PE hostfile, which SGE should pass in the `'PE_HOSTFILE'` environment variable, and creating an `SGEPEWorker` object (see below) for each processor. Other workers can still be added to the job if desired.

This class should be used to create a job from a MODELLER script running on the manager (first) node in an SGE parallel environment job.

6.35.3 SGEQsubJob() — create a job which can be expanded with Sun GridEngine 'qsub'

`SGEQsubJob(options, maxworker, seq=(), modeller_path=None, host=None)`

This functions identically to **Job()**, above, but it automatically grows by adding new `SGEQsubWorker` workers (up to a maximum of maxworker) if you submit more tasks to the job than there are available workers. (These are grouped into a single SGE array job.) options specifies options for the new `SGEQsubWorker` objects. New workers are not automatically added when using the message-passing interface; you should manually add new `SGEQsubWorker` objects in this case.

This class should be used to create a job from a MODELLER script running on your SGE batch system head node (or other node which can run `'qsub'` and has a shared filesystem with the worker nodes).

6.35.4 Job.worker_startup_commands — Worker startup commands

This is a Python list, initially empty, of Python commands that will be run on each worker when it is started up. You can add your own worker initialization by adding to this list.

6.35.5 Job.queue_task() — submit a task to run within the job

```
queue_task(taskobj)
```

This adds the given `Task` object to the job's queue. All tasks in the queue can later be run with **Job.run_all_tasks()** or **Job.yield_tasks_unordered()**.

The task should be a instance of a class derived from `Task`, which provides a `'run'` method. This method will be run on the worker node; any arguments to this method are given on the manager when the object is created, and are automatically passed for you to the worker. Anything you return from this method is automatically passed back to the manager. (Note that **Communicator.send_data()** is used to send this data, which cannot send all internal MODELLER types.)

Note that generally you need to declare tasks in a separate Python module, and load them in with the `import` statement, as the tasks are passed using Python's `pickle` module, which will otherwise give an error such as `'AttributeError: 'module' object has no attribute 'mytask''`.

Example: See **Job.run_all_tasks()** command.

6.35.6 Job.run_all_tasks() — run all queued tasks, and return results

```
run_all_tasks()
```

This runs all of the tasks in the job's queue on any available worker. When all of the tasks have finished, this functions returns a list of all the return values from the tasks, in the same order that they were submitted.

Tasks are run in a simple round-robin fashion on the available workers. If a worker fails while running a task, that task is automatically resubmitted to another worker. If you submit more tasks than available workers, new workers are automatically added to the job if the job supports this functionality (*e.g.*, **SGEQsubJob()**).

See also **Job.yield_tasks_unordered()**.

Example: examples/python/mytask.py

```python
from modeller import *
from modeller.parallel import Task

class MyTask(Task):
    """A task to read in a PDB file on the worker, and return the resolution"""
    def run(self, code):
        env = Environ()
        env.io.atom_files_directory = ["../atom_files"]
        mdl = Model(env, file=code)
        return mdl.resolution
```

Example: examples/python/parallel-task.py

```python
from modeller import *
from modeller.parallel import Job, LocalWorker

# Load in my task from mytask.py (note: needs to be in a separate Python
# module like this, in order for Python's pickle module to work correctly)
from mytask import MyTask

log.minimal()
# Create an empty parallel job, and then add 2 worker processes running
# on the local machine
j = Job()
j.append(LocalWorker())
j.append(LocalWorker())

# Run 'mytask' tasks
j.queue_task(MyTask('1fdn'))
j.queue_task(MyTask('1b3q'))
j.queue_task(MyTask('1blu'))

results = j.run_all_tasks()

print("Got model resolution: " + str(results))
```

6.35.7 Job.yield_tasks_unordered() — run all queued tasks, and yield unordered results

`yield_tasks_unordered()`

This runs all of the tasks in the job's queue on any available worker, like **Job.run_all_tasks()**. However, it does not wait until all the tasks have completed; it will return results from tasks as they complete, as a Python generator. Note that this means the results will not generally be in the same order as the tasks were submitted.

See also **Job.run_all_tasks()**.

6.35.8 Job.start() — start all workers for message-passing

`start()`

This starts all non-running workers in the job, such that they can later be used for message passing. (There is no need to call this command if using the task interface, as the workers are automatically started when required.)

Example: examples/python/parallel-msg.py

```
from modeller import *
from modeller.parallel import Job, LocalWorker

# Create an empty parallel job, and then add a single worker process running
# on the local machine
j = Job()
j.append(LocalWorker())

# Start all worker processes (note: this will only work if 'modxxx' - where
# xxx is the Modeller version - is in the PATH; if not, use modeller_path
# to specify an alternate location)
j.start()

# Have each worker read in a PDB file (provided by us, the master) and
# return the PDB resolution back to us
for worker in j:
    worker.run_cmd('''
env = Environ()
env.io.atom_files_directory = ["../atom_files"]
log.verbose()
code = master.get_data()
mdl = Model(env, file=code)
master.send_data(mdl.resolution)
''')
    worker.send_data('1fdn')
    data = worker.get_data()
    print("%s returned model resolution: %f" % (str(worker), data))
```

6.35.9 Communicator.send_data() — send data

`send_data(data)`

This sends the given data to the communicator. For sending from the manager to workers, this communicator is simply the 'worker' object itself. For sending from the workers back to the manager, a communicator called 'manager' is provided for you.

Any data that can be processed by the Python 'pickle' module can be sent to and from workers. This includes most Python objects, simple data such as integer and floating point numbers, strings, and many MODELLER objects. Note, however, that internal MODELLER data is *n*ot passed in these objects, so if, for example, you were to pass a `Model` object, it would contain no atoms when it reached the worker. For complex data such as this, write it to a file at one end and read it back at the other.

It is an error for the manager to send data to a worker using this function, unless the worker is already waiting for data (*i.e.*, by itself calling **Communicator.get_data()**). Generally this means you should call **Worker.run_cmd()** before **Communicator.send_data()**.

If there is a problem with the network, this function will raise a `NetworkError` exception. You can trap this to rerun the calculation on a different worker, for example. Any other errors (*e.g.*, a syntax error in your script) will raise a `RemoteError` exception.

6.35.10 Communicator.get_data() — get data

`get_data(allow_heartbeat=False)`

This gets the next piece of data available from the worker or manager. It must be matched by a corresponding **Communicator.send_data()** call at the other end, or an error will result. Errors are as for **Communicator.send_data()**.

6.35.11 Worker.run_cmd() — run a command on the worker

`run_cmd(cmd)`

This runs a command (or several commands, separated by line breaks) on the worker. (It is not possible for the worker to run commands on the manager.) Errors are as for **Communicator.send_data()**.

6.35.12 LocalWorker() — create a worker running on the local machine

`LocalWorker()`

This creates a new worker process, which will run on the same machine as the manager. This is useful if the machine has multiple CPUs, or if the manager process is going to be largely idle. It should be added to a `Job` object to be useful.

6.35.13 SGEPEWorker() — create a worker running on a Sun GridEngine parallel environment worker node

`SGEPEWorker(nodename)`

This creates a new worker process, which runs on the worker node given by `nodename` as part of a Sun GridEngine (SGE) parallel job. The process is started using `'qrsh -inherit -V'`, so your SGE setup should be correctly configured to allow this. Generally you would use an `SGEPEJob` object instead, above, to automatically create worker processes for all SGE nodes.

6.35.14 SGEQsubWorker() — create a 'qsub' worker running on a Sun GridEngine node

`SGEQsubWorker(options, array=None)`

This submits a single-processor job to a Sun GridEngine system with the `'qsub'` command. Once the job starts, it connects back to your manager script and acts as a new worker process. Generally you would use an `SGEQsubJob` object instead, above, to automatically create worker processes when required.

6.35.15 SSHWorker() — create a worker on a remote host accessed via ssh

`SSHWorker(nodename, ssh_command='ssh')`

This creates a new worker process running on a remote host given by `nodename`, started using `'ssh'`. You can change the command used to start processes from ssh to, for example, `'rsh'`, with the `ssh_command` parameter. For most applications, you would need to set up passwordless rsh or ssh for this to be useful.

Chapter 7

MODELLER low-level programming

MODELLER provides many classes for alignment, searching, comparative modeling, and model evaluation, which are suitable for many purposes. However, for some advanced applications you may need to adjust some of the low-level functionality of the program, or call the MODELLER functions from your own programs.

7.1 User-defined features and restraint forms

MODELLER provides a variety of pre-defined features and mathematical restraint forms (see Section 5.3.1), but you can add your own by creating new Python classes. For cases where the conventional features and restraints approach is not practical, you can also add new energy function terms which act on all atoms in the system. This can be used to add entirely new kinds of restraint for novel modeling situations.

(Note that Python code is generally substantially slower than compiled C or FORTRAN. If you find yourself relying on a large amount of Python extensions to MODELLER, you may want to recompile the code with Cython, or rewrite your code as C extension modules.)

7.1.1 User-defined feature types

To create a new feature type, derive a new class from the base `features.Feature`. You should then set the `numatoms` member to the number of atoms your feature acts on, and also override the following functions: `eval`, `deriv`, and `is_angle`. You can also derive your class from any of the built-in MODELLER features (*e.g.*, `features.Angle`) if you desire.

The `eval` function is called from MODELLER with a `Model` object and the indices of the atoms defining the feature. Your function should return the value of the feature. The `deriv` function is similar, but is also passed the current feature value; you should return the derivatives of the feature with respect to x, y and z of each defining atom. The `is_angle` function should return `True` if your feature is an angle, in which case MODELLER will automatically deal with periodicity for you, and convert any feature values to degrees for the user. (Your `eval` and `deriv` functions should, however, return angle values in radians.)

Example: examples/python/user_feat.py

```
from modeller import *
from modeller.scripts import complete_pdb

env = Environ()

env.io.atom_files_directory = ['../atom_files']
log.verbose()
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')
```

```
class MyDist(features.Feature):
    """An implementation of Modeller's distance feature (type 1) in
       pure Python. For improved performance, see cuser_feat.py, which
       implements the feature in C."""

    numatoms = 2

    def eval(self, mdl, atom_indices):
        (a1, a2) = self.indices_to_atoms(mdl, atom_indices)
        dist = ((a1.x - a2.x) ** 2 + (a1.y - a2.y) ** 2
                + (a1.z - a2.z) ** 2) ** 0.5
        return dist

    def deriv(self, mdl, atom_indices, feat):
        (a1, a2) = self.indices_to_atoms(mdl, atom_indices)
        dx1 = (a1.x - a2.x) / feat
        dy1 = (a1.y - a2.y) / feat
        dz1 = (a1.z - a2.z) / feat
        dx2 = -dx1
        dy2 = -dy1
        dz2 = -dz1
        return ((dx1, dx2), (dy1, dy2), (dz1, dz2))

    def is_angle(self):
        return False


mdl = complete_pdb(env, "1fdn")
sel = Selection(mdl)
rsr = mdl.restraints
at = mdl.atoms
rsr.add(forms.Gaussian(group=physical.bond,
                       feature=MyDist(at['CA:1:A'], at['C:1:A']),
                       mean=1.5380, stdev=0.0364))
sel.energy()
```

7.1.2 User-defined restraint forms

To create a new restraint form, derive a new class from the base `forms.RestraintForm`. You should then override the following functions: `__init__`, `eval`, `vmin`, `rvmin`, `min_mean`, `vheavy`, `rvheavy`, `heavy_mean`, and `get_range`. Note that presently you can *only* derive from this base class, not from MODELLER built-in forms.

Restraint forms can act on one or more features (each of which has an accompanying integer modality, which you can use for any purpose), and can take any number of floating-point parameters as input. The features and parameters are stored in `self._features` and `self._parameters` respectively, but for convenience the base constructor `RestraintForm.__init__` can set initial values for these.

The `eval` function is called from MODELLER with the current feature values, their types and modalities, and the parameter vector. You should return the objective function contribution and, if requested, the derivatives with respect to each feature. The feature types are required by the `deltaf` function, which returns the difference between the current feature value and the mean (a simple subtraction is not sufficient, as some feature types are periodic). Note that you must use the passed parameter vector, as the class is not persistent, and as such the `self._parameters` variable (or any other object variable you may have set) is not available to this function.

The `get_range` function is used to define the feature range over which the form is clearly non-linear. It is simply passed a similar set of parameters to `eval`, and should return a 2-element tuple containing the minimum and maximum feature values. It is only necessary to define this function if the form acts on only a single feature and you want to be able to convert it to a cubic spline using **Restraints.spline()**.

The other functions are used to return the minimal and heavy restraint violations (both absolute and relative; see Section 5.3.1) and the means. The heavy and minimal means correspond to the global and local minima.

Example: examples/python/user_form.py

```
from modeller import *
from modeller.scripts import complete_pdb

env = Environ()

env.io.atom_files_directory = ['../atom_files']
log.verbose()
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

class MyGauss(forms.RestraintForm):
    """An implementation of Modeller's harmonic/Gaussian restraint (type 3)
       in pure Python"""

    rt = 0.5900991    # RT at 297.15K, in kcal/mol

    def __init__(self, group, feature, mean, stdev):
        forms.RestraintForm.__init__(self, group, feature, 0, (mean, stdev))

    def eval(self, feats, iftyp, modal, param, deriv):
        (mean, stdev) = param
        delt = self.deltaf(feats[0], mean, iftyp[0])
        val = self.rt * 0.5 * delt**2  / stdev**2
        if deriv:
            fderv = self.rt * delt / stdev**2
            return val, [fderv]
        else:
            return val

    def vmin(self, feats, iftyp, modal, param):
        (mean, stdev) = param
        return self.deltaf(feats[0], mean, iftyp[0])

    def rvmin(self, feats, iftyp, modal, param):
        (mean, stdev) = param
        return self.deltaf(feats[0], mean, iftyp[0]) / stdev

    def min_mean(self, feats, iftyp, modal, param):
        (mean, stdev) = param
        return [mean]

    def get_range(self, iftyp, modal, param, spline_range):
        (mean, stdev) = param
        return (mean - stdev * spline_range, mean + stdev * spline_range)

    # There is only one minimum, so the 'heavy' mean is the same as the 'min'
    vheavy = vmin
    rvheavy = rvmin
    heavy_mean = min_mean
```

```
mdl = complete_pdb(env, "1fdn")
sel = Selection(mdl)
rsr = mdl.restraints
at = mdl.atoms
rsr.add(MyGauss(group=physical.bond,
                feature=features.Distance(at['CB:1:A'], at['CA:1:A']),
                mean=1.5380, stdev=0.0364))
sel.energy()

# Restraints using user-defined forms can be converted to splines for speed.
# Only one-dimensional forms that define the get_range() method can be splined.
rsr.spline(MyGauss, features.Distance, physical.bond, spline_dx=0.05)
sel.energy()
```

7.1.3 User-defined energy terms

To create a new energy term, derive a new class from the base `terms.EnergyTerm`. You should then override the `eval` function. You can also override the `__init__` function if you want to define function parameters. Objects of this class can then be created and added to the EnergyData.energy_terms list.

The `eval` function is called from MODELLER with a `Model` object, and the indices of all selected atoms. You should return the objective function contribution and, if requested, the derivatives with respect to the Cartesian coordinates.

Example: examples/python/user_term.py

```
from modeller import *
from modeller.scripts import complete_pdb

env = Environ()
log.verbose()
env.io.atom_files_directory = ['../atom_files']
env.libs.topology.read(file='$(LIB)/top_heav.lib')
env.libs.parameters.read(file='$(LIB)/par.lib')

class MyTerm(terms.EnergyTerm):
    """Custom energy term, which tries to force all atoms to one side of
       the x=10.0A plane"""

    _physical_type = physical.absposition

    # Override the __init__ function so that we can pass in a 'strength'
    # parameter
    def __init__(self, strength):
        self.strength = strength
        terms.EnergyTerm.__init__(self)

    def eval(self, mdl, deriv, indats):
        atoms = self.indices_to_atoms(mdl, indats)
        e = 0.
        dvx = [0.] * len(indats)
        dvy = [0.] * len(indats)
        dvz = [0.] * len(indats)
        for (num, at) in enumerate(atoms):
            # Enforce a linearly increasing potential in the x direction
            if at.x > 10.0:
                e += (at.x - 10.0) * self.strength
                dvx[num] += self.strength
        if deriv:
            return (e, dvx, dvy, dvz)
        else:
            return e

t = env.edat.energy_terms
t.append(MyTerm(strength=1.0))

mdl = complete_pdb(env, "1fdn")
sel = Selection(mdl)
print(sel.energy())
```

7.2 Modeller programming interface (API)

On most platforms, the core of the Modeller program is actually a dynamic library ('.so', '.dylib' or '.dll' file). The Modeller program itself is just a thin 'wrapper' which uses both this library and the Python library to run scripts.

You can use the Modeller library in your own programs. To do this, you must use the API functions defined in the Modeller header files, a collection of '.h' files which usually can be found in the `$MODINSTALL10v7/src/include` directory, when compiling your program, and then link against the Modeller library. It is most straightforward to do this in C (which we will use here as an example) although any language which can interface with C libraries can be used. See the comments in the main header file 'modeller.h' for simple usage instructions.

The Python interface is also built from these header files, using the SWIG package. All of the files used to build this interface can be found in the `$MODINSTALL10v7/src/swig` directory. You can use these to build an interface for a different version of Python; see the 'README' file in this directory for instructions.

If you installed the Modeller RPM package, you can run `pkg-config --cflags modeller` to get the necessary C compiler flags for you to be able to include the Modeller header(s). Similarly, the `--libs` option outputs the linker flags needed to link with the Modeller library. (If you did not install the RPM, you can get the same information by running `mod10.7 --cflags`.)

In many cases, it is more convenient to implement *extensions* to Modeller in C. These can work together with the main Modeller code and any Python scripts, and can be much faster than implementing the code in Python. See 'cuser_feat.py', 'cuser_form.py' and 'cuser_term.py' in the `examples/c-extensions/` directory for examples.

Example: examples/commands/c-example.c

```
#include <glib.h>
#include <stdio.h>
#include <stdlib.h>

#include <modeller.h>

/* Example of using Modeller from a C program. This simply reads in a PDB
 * file, prints out some data from that file, and then writes out a new
 * file in MMCIF format.
 *
 * To compile, use (where XXX is your Modeller version):
 * gcc -Wall -o c-example c-example.c `modXXX --cflags --libs` \
 *     `pkg-config --cflags --libs glib-2.0`
 * (If you use a compiler other than gcc, or a non-Unix system, you may need
 * to run 'modXXX --cflags --libs' manually and construct suitable compiler
 * options by hand.)
 *
 * To run, you must ensure that the Modeller dynamic libraries are in your
 * search path. This can be done on most systems by adding the directory
 * reported by 'modXXX --libs' to the LD_LIBRARY_PATH environment variable.
 * (On Mac, set DYLD_LIBRARY_PATH instead. On Windows, PATH. On AIX, LIBPATH.)
 *
 * You must also ensure that Modeller knows where it was installed,
 * and what the license key is. You can either do this by setting the
 * MODINSTALLXXX and KEY_MODELLERXXX environment variables accordingly, or
 * by calling the mod_install_dir_set() and mod_license_key_set() functions
 * before you call mod_start(). For example, if Modeller is installed in
 * /lib/modeller on a 32-bit Linux system, the following would work from the
 * command line (all on one line), where KEY is your license key:
 *     KEY_MODELLERXXX=KEY MODINSTALLXXX=/lib/modeller/
 *     LD_LIBRARY_PATH=/lib/modeller/lib/i386-intel8 ./c-example
 */


/* Exit, reporting the Modeller error, iff one occurred. */
void handle_error(int ierr)
{
  if (ierr != 0) {
    GError *err = mod_error_get();
    fprintf(stderr, "Modeller error: %s\n", err->message);
    g_error_free(err);
    exit(1);
  }
}

int main(void)
{
  struct mod_libraries *libs;
  struct mod_model *mdl;
  struct mod_io_data *io;
  struct mod_file *fh;
  int ierr, *sel1, nsel1;

  /* Uncomment these lines to hard code install location and license key,
     rather than setting MODINSTALLXXX and KEY_MODELLERXXX environment
     variables (see above) */
  /* mod_install_dir_set("/lib/modeller"); */
  /* mod_license_key_set("KEY"); */

  mod_start(&ierr);
  handle_error(ierr);
  mod_header_write();

  mod_log_set(2, 1);
  libs = mod_libraries_new(NULL);
  fh = mod_file_open("${LIB}/restyp.lib", "r");
  if (fh) {
    mod_libraries_read_libs(libs, fh, &ierr);
    mod_file_close(fh, &ierr);
  } else {
    ierr = 1;
  }
  handle_error(ierr);
  mod_libraries_rand_seed_set(libs, -8123);

  mdl = mod_model_new(NULL);
  io = mod_io_data_new();
  fh = mod_file_open("../atom_files/2nbt.pdb", "r");
  if (fh) {
    mod_model_read(mdl, io, libs, fh, "PDB", "FIRST:@LAST:  ", 7, 0, &ierr);
    mod_file_close(fh, &ierr);
  } else {
    ierr = 1;
  }
  handle_error(ierr);
  printf("Model of %s solved at resolution %f, rfactor %f\n", mdl->seq.name,
         mdl->seq.resol, mdl->seq.rfactr);
  fh = mod_file_open("new.cif", "w");
  if (fh) {
    mod_selection_all(mdl, &sel1, &nsel1);
    mod_model_write(mdl, libs, sel1, nsel1, fh, "MMCIF", 0, 1, "", &ierr);
    g_free(sel1);
    mod_file_close(fh, &ierr);
  } else {
    ierr = 1;
  }
  handle_error(ierr);
  mod_libraries_free(libs);
  mod_model_free(mdl);
  mod_io_data_free(io);

  mod_end();
  return 0;
}
```

Appendix A

Methods

A.1 Dynamic programming for sequence and structure comparison and searching

In this section, the basic dynamic programming method for sequence alignment is described [Šali & Blundell, 1990]. This method forms the core of the pairwise and multiple sequence and structure comparisons as well as of the sequence database searching.

A.1.1 Pairwise comparison

The residue by residue scores W_{ij} can be used directly in the sequence alignment algorithm of Needleman & Wunsch [Needleman & Wunsch, 1970] to obtain the comparison of two protein sequences or structures. The only difference between the two types of comparison is in the type of the comparison matrix. In the case of sequence, the amino acid substitution matrix is used. In the case of 3D structure, the Euclidean distance (or some function of it) between two equivalent atoms in the current optimal superposition is used [Šali & Blundell, 1990].

The problem of the optimal alignment of two sequences as addressed by the algorithm of Needleman & Wunsch is as follows. We are given two sequences of elements and an M times N score matrix $\mathcal{W}$ where M and N are the numbers of elements in the first and second sequence. The scoring matrix is composed of scores W_{ij} describing differences between elements i and j from the first and second sequence respectively. The goal is to obtain an optimal set of equivalences that match elements of the first sequence to the elements of the second sequence. The equivalence assignments are subject to the following "progression rule": for elements i and k from the first sequence and elements j and l from the second sequence, if element i is equivalenced to element j, if element k is equivalenced to element l and if k is greater than i, l must also be greater than j. The optimal set of equivalences is the one with the smallest alignment score. The alignment score is a sum of scores corresponding to matched elements, also increased for occurrences of non-equivalenced elements (*ie* gaps). For a detailed discussion of this and related problems see [Sankoff & Kruskal, 1983].

We summarize the dynamic programming formulae used by Modeller to obtain the optimal alignment since they differ slightly from those already published [Sellers, 1974, Gotoh, 1982]. The recursive dynamic programming formulae that give a matrix $\mathcal{D}$ are:

$$\begin{aligned}
D_{i,j} &= \min\begin{cases} P_{i,j} \\ D_{i-1,j-1} + W_{i,j} \\ Q_{i,j} \end{cases} \\
P_{i,j} &= \min\begin{cases} D_{i-1,j} + g(1) \\ P_{i-1,j} + v \end{cases} \\
Q_{i,j} &= \min\begin{cases} D_{i,j-1} + g(1) \\ Q_{i,j-1} + v \end{cases}
\end{aligned} \tag{A.1}$$

where $g(l)$ is a linear gap penalty function:

$$g(l) = u + v \cdot l\,. \tag{A.2}$$

Note that only a vector is needed for the storage of P and Q. The uppermost formula in Eq. A.1 is calculated for $i = M$ and $j = N$. Variable l is a gap length and parameters u and v are gap-penalty constants.

The arrays $\mathcal{D}$, $\mathcal{P}$ and $\mathcal{Q}$ are initialized as follows:

$$\begin{aligned}
D_{i,0} &= \begin{cases} 0, & i \le e \\ g(i-e), & e < i \le N \end{cases} \\
D_{0,j} &= \begin{cases} 0, & j \le e \\ g(j-e), & e < j \le N \end{cases} \\
P_{i,0} &= Q_{i,0} = \infty, \qquad i = 1, 2, \ldots, M \\
P_{0,j} &= Q_{0,j} = \infty, \qquad j = 1, 2, \ldots, N
\end{aligned} \tag{A.3}$$

where parameter e is the maximal number of elements at sequence termini which are not penalized with a gap-penalty if not equivalenced. A segment at the terminus of length e is termed an "overhang". Note a difference from [Gotoh, 1982] in the initialization of the $\mathcal{P}$ and $\mathcal{Q}$ arrays. Also note that only vectors Q_i and P_j need to be stored in computer, not the whole arrays.

The minimal score $d_{M,N}$ is obtained from

$$d_{M,N} = \min(D_{i,N}, D_{M,j}) \tag{A.4}$$

where $i = M, M-1, \ldots, M-e$ and $j = N, N-1, \ldots, N-e$ to allow for the overhangs. The equivalence assignments are obtained by backtracking in matrix $\mathcal{D}$. Backtracking starts from the element $D_{i,j} = d_{M,N}$.

A.1.2 Variable gap penalty

Please refer to [Madhusudhan *et al.*, 2006] for a full description of the variable gap penalty dynamic programming algorithm.

A.1.3 Local *versus* global alignment

The Kruskal and Sankoff version of the local alignment is implemented [Sankoff & Kruskal, 1983]; this is very similar to the [Smith & Waterman, 1981] method. All the routines for the local alignment are exactly the same as the routines for the global alignment except that during the construction of matrix D the alignment is restarted each time the score becomes higher than a cutoff. The second difference is that the backtracking starts from the lowest element in the matrix, wherever it is.

A.1.4 Similarity *versus* distance scores

Each scoring matrix contains a flag determining whether it is a distance or similarity matrix. An appropriate optimization is used automatically. This is achieved by using exactly the same code except that one side of comparisons is multiplied by -1 when dealing with similarities as opposed to distances.

A.1.5 Multiple comparisons

In the discussion of the previous section, we have assumed that the sequences or structures would be compared in a pairwise manner. However, such pairwise comparisons of several related proteins may not be self consistent, *ie* the following transitivity rule can be broken: If residue a from protein A is equivalent to residue b in protein B which in turn is equivalent to residue c in protein C then the residue a from protein A must also be equivalent to residue c from protein C. This property is not always attained in the set of usual pairwise comparisons relating a group of similar proteins. For this reason we proceed by simultaneously aligning all proteins. This is achieved by aligning the second sequence with the first one, the third sequence with the alignment of the first two, *etc.* A more general tree-like growth of the multiple alignment is not yet implemented.

If the number of all proteins is N, $N-1$ alignments must be made to obtain the final multiple comparison. It is noted that once an equivalence or gap is introduced it is not changed in later stages.

A.2 Optimization of the objective function by Modeller

This section describes the optimization methods implemented in Modeller. The general form of the objective function and the structure of optimization are similar to molecular dynamics programs, such as Charmm [MacKerell *et al.*, 1998].

A.2.1 Function

Modeller minimizes the *objective function* F with respect to Cartesian coordinates of $\sim 10,000$ atoms (3D points) that form a *system* (one or more molecules):

$$F = F(\mathbf{R}) = F_{symm} + \sum_i c_i(\mathbf{f}_i, \mathbf{p}_i) \tag{A.5}$$

where F_{symm} is an optional symmetry term defined in Eq. A.99, $\mathbf{R}$ are Cartesian coordinates of all atoms, c is a restraint i, $\mathbf{f}$ is a geometric feature of a molecule, and $\mathbf{p}$ are parameters. For a 10,000 atom system there can be on the order of 200,000 restraints. The form of c is simple; it includes a quadratic function, cosine, a weighted sum of a few Gaussian functions, Coulomb law, Lennard-Jones potential, cubic splines, and some other simple functions. The geometric features presently include a distance, an angle, a dihedral angle, a pair of dihedral angles between two, three, four atoms and eight atoms, respectively, the shortest distance in the set of distances (not documented further), solvent accessibility in $\mathring{A}^2$, and atom density expressed as the number of atoms around the central atom. A pair of dihedral angles can be used to restrain such strongly correlated features as the mainchain dihedral angles Φ and Ψ. Each of the restraints also depends on a few parameters $\mathbf{p}_i$ that generally vary from a restraint to a restraint. Some restraints can restrain *pseudo-atoms* such as a gravity center of several atoms.

Modeller allows some atoms to be *fixed* during optimization; *i.e.*, only selected atoms are allowed to be moved. Similarly, Modeller also allows only a subset of all restraints to be actually used in the calculation of the objective function. Each subset is indicated by a list of indices specifying the selected atoms or restraints.

There are two kinds of restraints, *static* and *dynamic*, that both contribute to the objective function as indicated in Eq. A.5:

$$F = F_{symm} + F_s + F_d\ . \tag{A.6}$$

The static restraints and their parameters are pre-defined; *i.e.*, they are given before the call to the optimizer and are not changed during optimization. The dynamic restraints are re-generated repeatedly during optimization. Usually, the CPU time is spent evenly between the two kinds of restraints, although the dynamic restraints become more important as the size of the system increases. All dynamic restraints are always selected and they can restrain only pairs of atoms. In all other respects, the two kinds of restraints are the same.

The dynamic restraints are obtained from a *dynamic pairs list* (the non-bonded pairs list). Each dynamic pair corresponds to at least one restraint, which may or may not be violated. The dynamic pairs list includes only the pairs of atoms that satisfy the following three conditions: (1) One or both atoms in a pair are allowed to move. (2) The two atoms are not connected through one, two, or three chemical bonds. (3) The two atoms are closer than a preset cutoff distance (*e.g.*, 4 Å). There are on the order of 5000 atom pairs in the dynamic pairs list when only soft-sphere overlap restraints are used. Currently, the restraint types on the dynamic atom pairs that can be selected include the soft-sphere overlap, Lennard-Jones, Coulomb interactions, and MODELLER non-bonded spline restraints.

The existence of the dynamic pairs list is justified by the fact that dynamic pairs are usually a small fraction of all possible atom–atom pairs ($N \cdot (N-1)/2$, where N is the number of atoms in a system). The use of the dynamic pairs list becomes especially beneficent as the size of the system increases.

The actual algorithm for creating the dynamic pairs list varies with the size of the system, whether or not all atoms are allowed to move, or whether or not the user wants to include the fixed environment in the calculation of non-bonded restraints involving the selected atoms. See Section 6.11 for more information.

The hash-function algorithm is used to determine whether or not two atoms are a dynamic atom pair. This algorithm is about 20 times slower than a lookup table but it requires much less memory and still spends a negligible fraction of the total CPU time. A hash-function table is prepared only once before the start of the optimization and any other operation involving an evaluation of the objective function (*e.g.*, **Selection.energy()**, **Selection.hot_atoms()**, Section 6.11).

The dynamic pairs list is not necessarily re-generated each time the objective function is evaluated, although the contribution of the restraint to the objective function is calculated in each call to the objective function routine with the current values of the Cartesian coordinates. The dynamic pairs list is re-generated only when maximal atomic shifts accumulate to a value larger than a preset cutoff. This cutoff is chosen such that there cannot be a violation of a restraint without having its atom pair on the dynamic pairs list. The dynamic pairs list is recalculated in $\sim 20\%$ and $\sim 2\%$ of the objective function calls at the beginning and the end of optimization, respectively.

Each evaluation of the objective function or of its first derivatives with respect to the Cartesian coordinates involves the following steps:

1. Calculate non-fixed pseudo-atoms from the current atomic positions.
2. Update the dynamic pairs list, if necessary.
3. Calculate the violations of selected restraints and all other quantities that are shared between the calculations of the objective function and its derivatives.
4. Sum the contributions of all violated restraints to the objective function and the derivatives.

A.2.2 Optimizers

MODELLER currently implements a Beale restart conjugate gradients algorithm [Shanno & Phua, 1980, Shanno & Phua, 1982] and a molecular dynamics procedure with the leap-frog Verlet integrator [Verlet, 1967]. The conjugate gradients optimizer is usually used in combination with the variable target function method [Braun & Gõ, 1985] which is implemented with the `AutoModel` class (Section A.4). The molecular dynamics procedure can be used in a simulated annealing protocol that is also implemented with the `AutoModel` class.

Molecular dynamics

Force in MODELLER is obtained by equating the objective function F with internal energy in kcal/mole. The atomic masses are all set to that of C^{12} (MODELLER unit is kg/mole). The initial velocities at a given temperature are obtained from a Gaussian random number generator with a mean and standard deviation of:

$$\bar{v}_x = 0 \tag{A.7}$$

$$\sigma_x = \sqrt{\frac{k_B T}{m_i}} \tag{A.8}$$

where k_B is the Boltzmann constant, m_i is the mass of one C^{12} atom, and the velocity is expressed in angstroms/femtosecond.

The Newtonian equations of motion are integrated by the leap-frog Verlet algorithm [Verlet, 1967]:

$$\dot{\boldsymbol{r}}_i\left(t+\frac{\delta t}{2}\right) = \dot{\boldsymbol{r}}_i\left(t-\frac{\delta t}{2}\right) - \frac{\partial F}{\partial \boldsymbol{r}_i(t)}\frac{\delta t}{m_i} \tag{A.9}$$

$$\boldsymbol{r}_i\left(t+\delta t\right) = \boldsymbol{r}_i(t) + \dot{\boldsymbol{r}}_i\left(t+\frac{\delta t}{2}\right)\delta t \tag{A.10}$$

where $\boldsymbol{r}_i$ is the position of atom i. In addition, velocity is capped at a maximum value, before calculating the shift, such that the maximal shift along one axis can only be cap_atom_shift. The velocities can be equilibrated every equilibrate steps to stabilize temperature. This is achieved by scaling the velocities with a factor f:

$$f = \sqrt{T/T_{kin}} \tag{A.11}$$

$$T_{kin} = \frac{E_{kin}}{\frac{1}{2}k_b N_f} \tag{A.12}$$

$$E_{kin} = \frac{1}{2}\sum_i^{N_{atoms}} m_i \dot{\boldsymbol{r}}_i^2 \tag{A.13}$$

where k_B is the Boltzmann constant, N_f the number of degrees of freedom, E_{kin} the current kinetic energy and T_{kin} the current kinetic temperature.

Langevin dynamics

Langevin dynamics (LD) are implemented as in [Loncharich *et al.*, 1992]. The equations of motion (Equation A.9) are modified as follows:

$$\dot{\boldsymbol{r}}_i\left(t+\frac{\delta t}{2}\right) = \dot{\boldsymbol{r}}_i\left(t-\frac{\delta t}{2}\right)\frac{1-\frac{1}{2}\gamma\delta t}{1+\frac{1}{2}\gamma\delta t} + \left(\boldsymbol{R}_i - \frac{\partial F}{\partial \boldsymbol{r}_i(t)}\right)\frac{\delta t}{m_i}\frac{1}{1+\frac{1}{2}\gamma\delta t} \tag{A.14}$$

where γ is a friction factor (in fs^{-1}) and $\boldsymbol{R}_i$ a random force, chosen to have zero mean and standard deviation

$$\sigma(\boldsymbol{R}_i) = \sqrt{\frac{2\gamma m_i k_B T}{\delta t}} \tag{A.15}$$

Self-guided MD and LD

Modeller also implements the self-guided MD [Wu & Wang, 1999] and LD [Wu & Brooks, 2003] methods. For self-guided MD, the equations of motion (Equation A.9) are modified as follows:

$$\boldsymbol{g}_i(t) = \left(1-\frac{\delta t}{t_l}\right)\boldsymbol{g}_i(t-\delta t) + \frac{\delta t}{t_l}\left(\lambda\boldsymbol{g}_i(t-\delta t) - \frac{\partial F}{\partial \boldsymbol{r}_i(t)}\right) \tag{A.16}$$

$$\dot{\boldsymbol{r}}_i\left(t+\frac{\delta t}{2}\right) = \dot{\boldsymbol{r}}_i\left(t-\frac{\delta t}{2}\right) + \left(\lambda\boldsymbol{g}_i(t) - \frac{\partial F}{\partial \boldsymbol{r}_i(t)}\right)\frac{\delta t}{m_i} \tag{A.17}$$

where λ is the guiding factor (the same for all atoms), t_l the guide time in femtoseconds, and $\boldsymbol{g}_i$ a guiding force, set to zero at the start of the simulation. (Position $\boldsymbol{r}_i$ is updated in the usual way.)

For self-guided Langevin dynamics, the guiding forces are determined as follows (terms are as defined in Equation A.14):

$$\boldsymbol{g}_i(t) = \left(1-\frac{\delta t}{t_l}\right)\boldsymbol{g}_i(t-\delta t) + \frac{\delta t}{t_l}\gamma m_i \dot{\boldsymbol{r}}_i\left(t-\frac{\delta t}{2}\right) \tag{A.18}$$

A scaling parameter χ is then determined by first making an unconstrained half step:

$$\dot{\boldsymbol{r}}'_i(t) = \dot{\boldsymbol{r}}_i\left(t-\frac{\delta t}{2}\right)+\frac{1}{2}\left(\lambda\boldsymbol{g}_i(t)+\boldsymbol{R}_i-\frac{\delta F}{\delta\boldsymbol{r}_i(t)}\right)\frac{\delta t}{m_i} \tag{A.19}$$

$$\zeta = \left(1+\frac{\gamma\delta t}{2}\right)\frac{\sum_i^N \lambda\boldsymbol{g}_i(t)\dot{\boldsymbol{r}}'_i(t)}{\sum_i^N m_i\dot{\boldsymbol{r}}'^2_i(t)} \tag{A.20}$$

$$\chi = \left(1+\frac{(\gamma+\zeta)\delta t}{2}\right)^{-1} \tag{A.21}$$

Finally, the velocities are advanced using the scaling factor:

$$\dot{\boldsymbol{r}}_i\left(t+\frac{\delta t}{2}\right)=(2\chi-1)\dot{\boldsymbol{r}}_i\left(t-\frac{\delta t}{2}\right)+\left(\lambda\boldsymbol{g}_i(t)+\boldsymbol{R}_i-\frac{\partial F}{\partial\boldsymbol{r}_i(t)}\right)\frac{\delta t}{m_i} \tag{A.22}$$

Rigid bodies

Where rigid bodies are used, these are optimized separately from the other atoms in the system. This has the additional advantage of reducing the number of degrees of freedom.

Rigid molecular dynamics

The state of each rigid body is specified by the position of the center of mass, $\boldsymbol{r}_{COM}$, and an orientation quaternion, $\tilde{q}$ [Goldstein, 1980]. (The quaternion has 4 components, q_1 through q_4, of which the first three refer to the vector part, and the last to the scalar.) The translational and rotational motions of each body are separated. Each body is translated about its center of mass using the standard Verlet equations (Equation A.9) using the force:

$$\frac{\partial F}{\partial\boldsymbol{r}_{COM}}=\sum_i\frac{\partial F}{\partial\boldsymbol{r_i}} \tag{A.23}$$

where the sum i operates over all atoms in the rigid body, and $\boldsymbol{r_i}$ is the position of atom i in real space.

For the rotational motion, the orientation quaternions are again integrated using the same Verlet equations. For this, the quaternion accelerations are calculated using the following relation [Rapaport, 1997]:

$$\ddot{\tilde{q}}=\frac{1}{2}W^T\begin{pmatrix}\dot{\omega}'_x\\ \dot{\omega}'_y\\ \dot{\omega}'_z\\ -2\sum_m\dot{q}_m^2\end{pmatrix} \tag{A.24}$$

where W is the orthogonal matrix

$$W=\begin{pmatrix} q_4 & q_3 & -q_2 & -q_1\\ -q_3 & q_4 & q_1 & -q_2\\ q_2 & -q_1 & q_4 & -q_3\\ q_1 & q_2 & q_3 & q_4\end{pmatrix} \tag{A.25}$$

and $\dot{\omega}'_k$ is the first derivative of the angular velocity (in the body-fixed frame) about axis k - *i.e.*, the angular acceleration. These angular accelerations are in turn calculated from the Euler equations for rigid body rotation, such as:

$$\dot{\omega}'_x=\frac{T_x+(I_y-I_z)\omega'_y\omega'_z}{I_x} \tag{A.26}$$

(Similar equations exist for the y and z components.) The angular velocities $\boldsymbol{\omega}'$ are obtained from the quaternion velocities:

$$\begin{pmatrix}\omega'_x\\ \omega'_y\\ \omega'_z\\ 0\end{pmatrix}=2W\dot{\tilde{q}} \tag{A.27}$$

The torque, $\boldsymbol{T}$, in the body-fixed frame, is calculated as

$$\boldsymbol{T} = A \sum_i (\boldsymbol{r_i} - \boldsymbol{r}_{COM}) \times -\frac{\partial F}{\partial \boldsymbol{r_i}} \tag{A.28}$$

and A is the rotation matrix to convert from world space to body space

$$A = 2 \begin{pmatrix} q_1^2 + q_4^2 - \frac{1}{2} & q_1 q_2 + q_3 q_4 & q_1 q_3 - q_2 q_4 \\ q_1 q_2 - q_3 q_4 & q_2^2 + q_4^2 - \frac{1}{2} & q_2 q_3 + q_1 q_4 \\ q_1 q_3 + q_2 q_4 & q_2 q_3 - q_1 q_4 & q_3^2 + q_4^2 - \frac{1}{2} \end{pmatrix} \tag{A.29}$$

and finally the x component of the inertia tensor, I_x, is given by

$$I_x = \sum_i m_i ({r'}^2_{i,y} + {r'}^2_{i,z}) \tag{A.30}$$

where $\boldsymbol{r'_i}$ is the position of each atom in body space (*i.e.* relative to the center of mass, and unrotated), and m_i is the mass of atom i (taken to be the mass of one C^{12} atom, as above). Similar relations exist for the y and z components.

The kinetic energy of each rigid body (used for temperature control) is given as a combination of translation and rotational components:

$$E_{kin}^{body} = \frac{1}{2} (\sum_i m) \dot{\boldsymbol{r}}^2_{COM} + \frac{1}{2} (I_x {\omega'}^2_x + I_y {\omega'}^2_y + I_z {\omega'}^2_z) \tag{A.31}$$

Initial translational and rotational velocities of each rigid body are set in the same way as for atomistic dynamics.

Rigid minimization

The state of each rigid body is specified by 6 parameters: the position of the center of mass, $\boldsymbol{r}_{COM}$, and the rotations in radians about the body-fixed axes: θ_x, θ_y, and θ_z. The first derivative of the objective function F with respect to the center of mass is obtained from Equation A.23, and those with respect to the angles from:

$$\frac{\partial F}{\partial \theta_k} = M_k \boldsymbol{r'_i} \cdot \frac{\partial F}{\partial \boldsymbol{r_i}} \tag{A.32}$$

The transformation matrices M_k are given as:

$$M_x = \begin{bmatrix} 0 & -\sin\theta_z \sin\theta_x - \cos\theta_z \sin\theta_y \cos\theta_x & \sin\theta_z \cos\theta_x - \cos\theta_z \sin\theta_y \sin\theta_x \\ 0 & -\cos\theta_z \sin\theta_x + \sin\theta_z \sin\theta_y \cos\theta_x & \cos\theta_z \cos\theta_x + \sin\theta_z \sin\theta_y \sin\theta_x \\ 0 & -\cos\theta_y \cos\theta_x & -\cos\theta_y \sin\theta_x \end{bmatrix} \tag{A.33}$$

$$M_y = \begin{bmatrix} -\cos\theta_z \sin\theta_y & -\cos\theta_z \cos\theta_y \sin\theta_x & \cos\theta_z \cos\theta_y \cos\theta_x \\ \sin\theta_z \sin\theta_y & \sin\theta_z \cos\theta_y \sin\theta_x & -\sin\theta_z \cos\theta_y \cos\theta_x \\ -\cos\theta_y & \sin\theta_y \sin\theta_x & -\sin\theta_y \cos\theta_x \end{bmatrix} \tag{A.34}$$

$$M_z = \begin{bmatrix} -\sin\theta_z \cos\theta_y & \cos\theta_z \cos\theta_x + \sin\theta_z \sin\theta_y \sin\theta_x & \cos\theta_z \sin\theta_x - \sin\theta_z \sin\theta_y \cos\theta_x \\ -\cos\theta_z \cos\theta_y & -\sin\theta_z \cos\theta_x + \cos\theta_z \sin\theta_y \sin\theta_x & -\sin\theta_z \sin\theta_x - \cos\theta_z \sin\theta_y \cos\theta_x \\ 0 & 0 & 0 \end{bmatrix} \tag{A.35}$$

The atomic positions $\boldsymbol{r_i}$ are reconstructed when necessary from the body's orientation by means of the following relation:

$$\boldsymbol{r_i} = M \boldsymbol{r'_i} + \boldsymbol{r}_{COM} \tag{A.36}$$

where M is the rotation matrix

$$M = \begin{bmatrix} \cos\theta_z \cos\theta_y & \sin\theta_z \cos\theta_x - \cos\theta_z \sin\theta_y \sin\theta_x & \sin\theta_z \sin\theta_x + \cos\theta_z \sin\theta_y \cos\theta_x \\ -\sin\theta_z \cos\theta_y & \cos\theta_z \cos\theta_x + \sin\theta_z \sin\theta_y \sin\theta_x & \cos\theta_z \sin\theta_x - \sin\theta_z \sin\theta_y \cos\theta_x \\ -\sin\theta_y & -\cos\theta_y \sin\theta_x & \cos\theta_y \cos\theta_x \end{bmatrix} \tag{A.37}$$

A.3 Equations used in the derivation of the molecular pdf

A.3.1 Features and their derivatives

Distance

Distance is defined by points i and j:

$$d = \sqrt{\boldsymbol{r}_{ij} \cdot \boldsymbol{r}_{ij}} = |\boldsymbol{r}_{ij}| = r_{ij} \tag{A.38}$$

where

$$\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j \ . \tag{A.39}$$

The first derivatives of d with respect to Cartesian coordinates are:

$$\frac{\partial d}{\partial \boldsymbol{r}_i} = \frac{\boldsymbol{r}_{ij}}{|\boldsymbol{r}_{ij}|} \tag{A.40}$$

$$\frac{\partial d}{\partial \boldsymbol{r}_j} = -\frac{\partial d}{\partial \boldsymbol{r}_i} \tag{A.41}$$

Angle

Angle is defined by points i, j, and k, and spanned by vectors ij and kj:

$$\alpha = \arccos \frac{\boldsymbol{r}_{ij} \cdot \boldsymbol{r}_{kj}}{|\boldsymbol{r}_{ij}||\boldsymbol{r}_{kj}|} \ . \tag{A.42}$$

It lies in the interval from 0 to 180°. Internal MODELLER units are radians.

The first derivatives of α with respect to Cartesian coordinates are:

$$\frac{\partial \alpha}{\partial \boldsymbol{r}_i} = \frac{\partial \alpha}{\partial \cos\alpha}\frac{\partial \cos\alpha}{\partial \boldsymbol{r}_i} = \frac{1}{\sqrt{1-\cos^2\alpha}}\frac{1}{r_{ij}}\left(\frac{\boldsymbol{r}_{ij}}{r_{ij}}\cos\alpha - \frac{\boldsymbol{r}_{kj}}{r_{kj}}\right) \tag{A.43}$$

$$\frac{\partial \alpha}{\partial \boldsymbol{r}_k} = \frac{\partial \alpha}{\partial \cos\alpha}\frac{\partial \cos\alpha}{\partial \boldsymbol{r}_k} = \frac{1}{\sqrt{1-\cos^2\alpha}}\frac{1}{r_{kj}}\left(\frac{\boldsymbol{r}_{kj}}{r_{kj}}\cos\alpha - \frac{\boldsymbol{r}_{ij}}{r_{ij}}\right) \tag{A.44}$$

$$\frac{\partial \alpha}{\partial \boldsymbol{r}_j} = -\frac{\partial \alpha}{\partial \boldsymbol{r}_i} - \frac{\partial \alpha}{\partial \boldsymbol{r}_k} \tag{A.45}$$

These equations for the derivatives have a numerical instability when the angle goes to 0 or to 180°. Presently, the problem is 'solved' by testing for the size of the angle; if it is too small, the derivatives are set to 0 in the hope that other restraints will eventually pull the angle towards well behaved regions. Thus, angle restraints of 0 or 180° should not be used in the conjugate gradients or molecular dynamics optimizations.

Dihedral angle

Dihedral angle is defined by points i, j, k, and l ($ijkl$):

$$\chi = \text{sign}(\chi) \arccos \frac{(\boldsymbol{r}_{ij} \times \boldsymbol{r}_{kj}) \cdot (\boldsymbol{r}_{kj} \times \boldsymbol{r}_{kl})}{|\boldsymbol{r}_{ij} \times \boldsymbol{r}_{kj}||\boldsymbol{r}_{kj} \times \boldsymbol{r}_{kl}|} \tag{A.46}$$

where

$$\text{sign}(\chi) = \text{sign}[\boldsymbol{r}_{kj} \cdot (\boldsymbol{r}_{ij} \times \boldsymbol{r}_{kj}) \times (\boldsymbol{r}_{kj} \times \boldsymbol{r}_{kl})] \ . \tag{A.47}$$

The first derivatives of χ with respect to Cartesian coordinates are:

$$\frac{\mathrm{d}\chi}{\mathrm{d}\boldsymbol{r}} = \frac{\mathrm{d}\chi}{\mathrm{d}\cos\chi}\frac{\mathrm{d}\cos\chi}{\mathrm{d}\boldsymbol{r}} \tag{A.48}$$

where

$$\frac{\mathrm{d}\chi}{\mathrm{d}\cos\chi} = \left(\frac{\mathrm{d}\cos\chi}{\mathrm{d}\chi}\right)^{-1} = -\frac{1}{\sin\chi} \tag{A.49}$$

and

$$\frac{\partial\cos\chi}{\partial\boldsymbol{r}_i} = \boldsymbol{r}_{kj}\times\boldsymbol{a} \tag{A.50}$$

$$\frac{\partial\cos\chi}{\partial\boldsymbol{r}_j} = \boldsymbol{r}_{ik}\times\boldsymbol{a} - \boldsymbol{r}_{kl}\times\boldsymbol{b} \tag{A.51}$$

$$\frac{\partial\cos\chi}{\partial\boldsymbol{r}_k} = \boldsymbol{r}_{jl}\times\boldsymbol{b} - \boldsymbol{r}_{ij}\times\boldsymbol{a} \tag{A.52}$$

$$\frac{\partial\cos\chi}{\partial\boldsymbol{r}_l} = \boldsymbol{r}_{ij}\times\boldsymbol{b} \tag{A.53}$$

$$\boldsymbol{a} = \frac{1}{|\boldsymbol{r}_{ij}\times\boldsymbol{r}_{kj}|}\left(\frac{\boldsymbol{r}_{kj}\times\boldsymbol{r}_{kl}}{|\boldsymbol{r}_{kj}\times\boldsymbol{r}_{kl}|} - \cos\chi\,\frac{\boldsymbol{r}_{ij}\times\boldsymbol{r}_{kj}}{|\boldsymbol{r}_{ij}\times\boldsymbol{r}_{kj}|}\right) \tag{A.54}$$

$$\boldsymbol{b} = \frac{1}{|\boldsymbol{r}_{kj}\times\boldsymbol{r}_{kl}|}\left(\frac{\boldsymbol{r}_{ij}\times\boldsymbol{r}_{kj}}{|\boldsymbol{r}_{ij}\times\boldsymbol{r}_{kj}|} - \cos\chi\,\frac{\boldsymbol{r}_{kj}\times\boldsymbol{r}_{kl}}{|\boldsymbol{r}_{kj}\times\boldsymbol{r}_{kl}|}\right)\ . \tag{A.55}$$

These equations for the derivatives have a numerical instability when the angle goes to 0. Thus, the following set of equations is used instead [van Schaik *et al.*, 1993]:

$$\boldsymbol{r}_{mj} = \boldsymbol{r}_{ij}\times\boldsymbol{r}_{kj} \tag{A.56}$$

$$\boldsymbol{r}_{nk} = \boldsymbol{r}_{kj}\times\boldsymbol{r}_{kl} \tag{A.57}$$

$$\frac{\partial\chi}{\partial\boldsymbol{r}_i} = \frac{r_{kj}}{r_{mj}^2}\boldsymbol{r}_{mj} \tag{A.58}$$

$$\frac{\partial\chi}{\partial\boldsymbol{r}_l} = -\frac{r_{kj}}{r_{nk}^2}\boldsymbol{r}_{nk} \tag{A.59}$$

$$\frac{\partial\chi}{\partial\boldsymbol{r}_j} = \left(\frac{\boldsymbol{r}_{ij}\cdot\boldsymbol{r}_{kj}}{r_{kj}^2} - 1\right)\frac{\partial\chi}{\partial\boldsymbol{r}_i} - \frac{\boldsymbol{r}_{kl}\cdot\boldsymbol{r}_{kj}}{r_{kj}^2}\frac{\partial\chi}{\partial\boldsymbol{r}_l} \tag{A.60}$$

$$\frac{\partial\chi}{\partial\boldsymbol{r}_k} = \left(\frac{\boldsymbol{r}_{kl}\cdot\boldsymbol{r}_{kj}}{r_{kj}^2} - 1\right)\frac{\partial\chi}{\partial\boldsymbol{r}_l} - \frac{\boldsymbol{r}_{ij}\cdot\boldsymbol{r}_{kj}}{r_{kj}^2}\frac{\partial\chi}{\partial\boldsymbol{r}_i} \tag{A.61}$$

The only possible instability in these equations is when the length of the central bond of the dihedral, r_{kj}, goes to 0. In such a case, which should not happen, the derivatives are set to 0. The expressions for an improper dihedral angle, as opposed to a dihedral or dihedral angle, are the same, except that indices $ijkl$ are permuted to $ikjl$. In both cases, covalent bonds ij, jk, and kl are defining the angle.

Atomic solvent accessibility

This is the accessibility value calculated by the PSA algorithm (see **Model.write_data()**). This is usually set by the last call to **Restraints.make()** or **Restraints.make_distance()**. First derivatives are not calculated, and are always returned as 0.

Atomic density

Atomic density for a given atom is simply calculated as the number of atoms within a distance EnergyData.contact_shell of that atom. First derivatives are not calculated, and are always returned as 0.

Atomic coordinates

The absolute atomic coordinates x_i, y_i and z_i are available for every point i, primarily for use in anchoring points to planes, lines or points. Their first derivatives with respect to Cartesian coordinates are of course simply 0 or 1.

A.3.2 Restraints and their derivatives

The chain rule is used to find the partial derivatives of the feature pdf with respect to the atomic coordinates. Thus, only the derivatives of the pdf with respect to the features are listed here.

Single Gaussian restraint

The pdf for a geometric feature f (*e.g.*, distance, angle, dihedral angle) is

$$p = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{f-\bar{f}}{\sigma}\right)^2\right] . \tag{A.62}$$

A corresponding restraint c in the sum that defines the objective function F is

$$c = -\ln p = \frac{1}{2}\left(\frac{f-\bar{f}}{\sigma}\right)^2 - \ln\frac{1}{\sigma\sqrt{2\pi}} \tag{A.63}$$

(Note that since the second term is constant for a given restraint, it is ignored. c is also scaled by RT in kcal/mol with $T = 297.15K$ to allow these scores to be summed with CHARMM energies.)

The first derivatives with respect to feature f are:

$$\frac{\mathrm{d}c}{\mathrm{d}f} = \frac{f-\bar{f}}{\sigma}\,\frac{1}{\sigma} . \tag{A.64}$$

The relative heavy violation with respect to f is given as:

$$\frac{f-\bar{f}}{\sigma} \tag{A.65}$$

Multiple Gaussian restraint

The polymodal pdf for a geometric feature f (*e.g.*, distance, angle, dihedral angle) is

$$p = \sum_{i=1}^{n} \omega_i p_i = \sum_{i=1}^{n} \omega_i \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{f-\bar{f}_i}{\sigma_i}\right)^2\right] . \tag{A.66}$$

A corresponding restraint c in the sum that defines the objective function F is (as before, this is scaled by RT):

$$c = -\ln p = -\ln\sum_{i=1}^{n} \omega_i p_i \tag{A.67}$$

The first derivatives with respect to feature f are:

$$\frac{\mathrm{d}c}{\mathrm{d}f} = \frac{1}{p}\sum_{i=1}^{n} \omega_i p_i \cdot \left[\frac{f-\bar{f}_i}{\sigma_i}\,\frac{1}{\sigma_i}\right] . \tag{A.68}$$

When any of the normalized deviations $v_i = (f - \bar{f}_i)/\sigma_i$ is large, there are numerical instabilities in calculating the derivatives because v_i are arguments to the *exp* function. Robustness is ensured as follows. The 'effective' normalized deviation is used in all the equations above when the magnitude of normalized violation v is larger than cutoff `rgauss1` (10 for double precision). This scheme works up to `rgauss2` (200 for double precision); violations larger than that are ignored. This trick is equivalent to increasing the standard deviation σ_i. A slight disadvantage is that there is a discontinuity in the first derivatives at `rgauss1`. However, if continuity were imposed, the range would not be extended (this is equivalent to linearizing the Gaussian, but since it is already linear for large deviations, a linearization with derivatives smoothness would not introduce much change at all).

$$\begin{aligned}
M &= 37 \quad ; \quad M^2/2 \text{ has to be smaller than the largest argument to exp} && \text{(A.69)}\\
A &= \frac{1}{M}\frac{\texttt{rgauss2} - M}{\texttt{rgauss2} - \texttt{rgauss1}} && \text{(A.70)}\\
B &= \frac{\texttt{rgauss2}}{M}\frac{M - \texttt{rgauss1}}{\texttt{rgauss2} - \texttt{rgauss1}} && \text{(A.71)}\\
v &= \frac{f - \bar{f}_i}{\sigma_i} && \text{(A.72)}\\
F &= A\,|v| + B && \text{(A.73)}\\
v' &= v/F && \text{(A.74)}
\end{aligned}$$

Now, Eqs. A.66–A.68 are used with v' instead of v. For single precision, $M = 12$, `rgauss1` $= 4$, `rgauss2` $= 100$. The relative heavy violation with respect to f is given as:

$$\max_{\omega_i}(f - \bar{f}_i)/\sigma_i \qquad \text{(A.75)}$$

Multiple binormal restraint

The polymodal pdf for a geometric feature (f_1, f_2) (*e.g.*, a pair of dihedral angles) is

$$\begin{aligned}
p &= \sum_{i=1}^{n}\omega_i p_i = \sum_{i=1}^{n}\omega_i \frac{1}{2\pi\sigma_{1i}\sigma_{2i}\sqrt{(1-\rho_i^2)}} \cdot \\
&\exp\left\{-\frac{1}{2(1-\rho_i^2)}\left[\left(\frac{f_1 - \bar{f}_{1i}}{\sigma_{1i}}\right)^2 - 2\rho_i\frac{f_1 - \bar{f}_{1i}}{\sigma_{1i}}\frac{f_2 - \bar{f}_{2i}}{\sigma_{2i}} + \left(\frac{f_2 - \bar{f}_{2i}}{\sigma_{2i}}\right)^2\right]\right\} . && \text{(A.76)}
\end{aligned}$$

where $\rho < 1$. ρ is the correlation coefficient between f_1 and f_2. MODELLER actually uses the following series expansion to calculate p:

$$\begin{aligned}
p &= \sum_{i=1}^{n}\omega_i \frac{1}{2\pi\sigma_{1i}\sigma_{2i}\sqrt{(1-\rho_i^2)}} \cdot \\
&\exp\left\{-\frac{1}{1-\rho_i^2}\left[\frac{1-\cos(f_1 - \bar{f}_{1i})}{\sigma_{1i}^2} - \rho_i\frac{\sin(f_1 - \bar{f}_{1i})}{\sigma_{1i}}\frac{\sin(f_2 - \bar{f}_{2i})}{\sigma_{2i}} + \frac{1-\cos(f_2 - \bar{f}_{2i})}{\sigma_{2i}^2}\right]\right\} . && \text{(A.77)}
\end{aligned}$$

A corresponding restraint c in the sum that defines the objective function F is (as before, this is scaled by RT):

$$c = -\ln p = -\ln\sum_{i=1}^{n}\omega_i p_i \qquad \text{(A.78)}$$

The first derivatives with respect to features f_1 and f_2 are:

$$\frac{\partial c}{\partial f_1} = \frac{1}{p}\sum_{i=1}^{n}\left[\omega_i p_i \cdot \frac{1}{\sigma_{1i}(1-\rho_i^2)}\left(\frac{\sin(f_1 - \bar{f}_{1i})}{\sigma_{1i}} - \rho_i\frac{\cos(f_1 - \bar{f}_{1i})\sin(f_2 - \bar{f}_{2i})}{\sigma_{2i}}\right)\right] \qquad \text{(A.79)}$$

$$\frac{\partial c}{\partial f_2} = \frac{1}{p}\sum_{i=1}^{n}\left[\omega_i p_i \cdot \frac{1}{\sigma_{2i}(1-\rho_i^2)}\left(\frac{\sin(f_2 - \bar{f}_{2i})}{\sigma_{2i}} - \rho_i\frac{\cos(f_2 - \bar{f}_{2i})\sin(f_1 - \bar{f}_{1i})}{\sigma_{1i}}\right)\right] . \qquad \text{(A.80)}$$

The relative heavy violation with respect to f is given as:

$$\max_{\omega_i} \sqrt{-\frac{1}{2(1-\rho_i^2)}\left[\left(\frac{f_1-\bar{f}_{1i}}{\sigma_{1i}}\right)^2 - 2\rho_i \frac{f_1-\bar{f}_{1i}}{\sigma_{1i}}\frac{f_2-\bar{f}_{2i}}{\sigma_{2i}} + \left(\frac{f_2-\bar{f}_{2i}}{\sigma_{2i}}\right)^2\right]} \tag{A.81}$$

Lower bound

This is like the left half of a single Gaussian restraint:

$$p = \begin{cases} p_{gauss}\ ; & f < \bar{f} \\ 0\ ; & f \geq \bar{f} \end{cases} \tag{A.82}$$

where $\bar{f}$ is a lower bound and p_{gauss} is given in Eq. A.62. A similar equation relying on the first derivatives of a Gaussian p holds for the first derivatives of a lower bound.

Upper bound

This is like the right half of a single Gaussian restraint:

$$p = \begin{cases} p_{gauss}\ ; & f > \bar{f} \\ 0\ ; & f \leq \bar{f} \end{cases} \tag{A.83}$$

where $\bar{f}$ is an upper bound and p_{gauss} is given in Eq. A.62. A similar equation relying on the first derivatives of a Gaussian p holds for the first derivatives of an upper bound.

Cosine restraint

This is usually used for dihedral angles f (improper dihedrals generally use a Gaussian restraint instead):

$$c = |b| - b\cos(nf + a) \tag{A.84}$$

where b is CHARMM force constant, a is phase shift (tested for 0 and 180°), and n is periodicity (tested for 1, 2, 3, 4, 5, and 6). The CHARMM phase value from the CHARMM parameter library corresponds to $a - 180°$. The force constant b can be negative, in effect offsetting the phase a for 180° compared to the same but positive force constant.

$$\frac{\mathrm{d}c}{\mathrm{d}f} = bn\sin(nf + a) \tag{A.85}$$

Coulomb restraint

$$c = \frac{1}{\epsilon_r}\frac{q_i q_j}{f} s(f, f_1, f_2) \tag{A.86}$$

$$s(f, f_1, f_2) = \begin{cases} 1\ ; & f \leq f_1 \\ \frac{(f_2-f)^2(f_2+2f-3f_1)}{(f_2-f_1)^3}\ ; & f_o < f \leq f_2 \\ 0\ ; & f > f_2 \end{cases} \tag{A.87}$$

where q_i and q_j are the atomic charges of atoms i and j, obtained from the CHARMM topology file, that are at a distance f. ϵ_r is the relative dielectric, controlled by the EnergyData.relative_dielectric variable. Function $s(f, f_1, f_2)$ is a switching function that smoothes the potential down to zero in the interval from f_1 to f_2 ($f_2 > f_1$). The total Coulomb energy of a molecule is a sum over all pairs of atoms that are not in the same bonds or bond angles. 1–4 energy for the 1–4 atom pairs in the same dihedral angle corresponds to the `ELEC14` MODELLER term; the remaining longer-range contribution corresponds to the `ELEC` term.

The first derivatives are:

$$\frac{\mathrm{d}c}{\mathrm{d}f} = -\frac{c}{f} + \frac{c}{s}\frac{\mathrm{d}s}{\mathrm{d}f} \tag{A.88}$$

$$\frac{\mathrm{d}s}{\mathrm{d}f} = \begin{cases} 0\ ; & f \le f_1 \\ \frac{6(f_2-f)(f_1-f)}{(f_2-f_1)^3}\ ; & f_1 < f \le f_2 \\ 0\ ; & f > f_2 \end{cases} \tag{A.89}$$

The violations of this restraint are always reported as zero.

Lennard-Jones restraint

Usually used for non-bonded distances:

$$c = \left[\left(\frac{A}{f}\right)^{12} - \left(\frac{B}{f}\right)^{6}\right] s(f, f_1, f_2) \tag{A.90}$$

The parameters f_1 and f_2 of the switching function can be different from those in Eq. A.87. The parameters A and B are obtained from the CHARMM parameter file (NONBOND section) where they are given as E_i and r_j such that $E_{ij}(f) = -4\sqrt{E_iE_j}[(\rho_{ij}/f)^{12} - (\rho_{ij}/f)^6]$ in kcal/mole for f in angstroms and $\rho = (r_i + r_j)/2^{1/6}$; the minimum of E is $-\sqrt{E_iE_j}$ at $f = (r_i + r_j)$, and its zero is at $f = \rho$. The total Lennard-Jones energy should be evaluated over all pairs of atoms that are not in the same bonds or bond angles. The parameters A and B for 1–4 pairs in dihedral angles can be different from those for the other pairs; they are obtained from the second set of E_i and r_i in the CHARMM parameter file, if it exists. 1–4 energy corresponds to the `LJ14` MODELLER term; the remaining longer-range contribution corresponds to the `LJ` term.

The first derivatives are:

$$\frac{\mathrm{d}c}{\mathrm{d}f} = \frac{Cs}{f} - C\frac{\mathrm{d}s}{\mathrm{d}f} \tag{A.91}$$

$$C = -12\left(\frac{A}{f}\right)^{12} + 6\left(\frac{B}{f}\right)^{6} \tag{A.92}$$

As f tends toward zero, the repulsive part of the energy dominates, and approaches infinity. Near-infinite forces result in unstable trajectories during optimization. This is particularly a problem in the first few steps of optimization starting from randomized, interpolated, or otherwise non-physical atomic coordinates. To avoid this, the potential is simply artificially truncated: if A/f exceeds 6, f is treated as being equal to $A/6$.

The violations of this restraint are always reported as zero.

Spline restraint

Any restraint form can be represented by a cubic spline [Press *et al.*, 1992]:

$$c = Ac_j + Bc_{j+1} + Cc''_j + Dc''_{j+1} \tag{A.93}$$

$$A = \frac{f_{j+1} - f}{f_{j+1} - f_j} \tag{A.94}$$

$$B = 1 - A \tag{A.95}$$

$$C = \frac{1}{6}(A^3 - A)(f_{j+1} - f_j)^2 \tag{A.96}$$

$$D = \frac{1}{6}(B^3 - B)(f_{j+1} - f_j)^2 \tag{A.97}$$

where $f_j \le f \le f_{j+1}$.

The first derivatives are:

$$\frac{\mathrm{d}c}{\mathrm{d}f} = \frac{c_{j+1} - c_j}{f_{j+1} - f_j} - \frac{3A^2 - 1}{6}(f_{j+1} - f_j)c''_j + \frac{3B^2 - 1}{6}(f_{j+1} - f_j)c''_{j+1} \tag{A.98}$$

The values of c and c' beyond f_1 and f_n are obtained by linear interpolation from the termini. A violation of the restraint is calculated by finding the global minimum. A relative violation is estimated by using a standard deviation (*e.g.*, force constant) obtained by fitting a parabola to the global minimum.

Variable spacing of spline points could be used to save on memory. However, this would increase the execution time, so it is not used.

To calculate the relative heavy violation, the feature value $\bar{f}$ that results in the smallest value of the restraint is obtained by interpolation, and a Gaussian function is fitted locally around this value to obtain the standard deviation σ. These are then used in Eq. A.65.

Symmetry restraint

The asymmetry penalty added to the objective function is defined as

$$F_{symm} = \sum_{i<j} \omega_i \omega_j (d_{ij} - d'_{ij})^2 \tag{A.99}$$

where the sum runs over all pairs of equivalent atoms ij, ω_i is an atom weight for atom i, d_{ij} is an intra-molecular distance between atoms ij in the first segment, and d'_{ij} is the equivalent distance in the second segment.

For each $i < j$, the first derivatives are:

$$\frac{\partial c}{\partial \boldsymbol{d}_{ij}} = 2\omega_i \omega_j (d_{ij} - d'_{ij}) \frac{\boldsymbol{d}_{ij}}{d_{ij}} \tag{A.100}$$

$$\frac{\partial c}{\partial \boldsymbol{d}'_{ij}} = -2\omega_i \omega_j (d_{ij} - d'_{ij}) \frac{\boldsymbol{d}'_{ij}}{d'_{ij}} \tag{A.101}$$

Thus, the total first derivatives are obtained by summing the two expressions above for all i and $j > i$ distances.

A.4 Flowchart of comparative modeling by MODELLER

This section describes a flowchart of comparative modeling by MODELLER, as implemented in the `AutoModel` class (see chapter 2).

Input: script file, alignment file, PDB file(s) for template(s).

Output:

`job.log`	`log` file
`job.ini`	initial conformation for optimization
`job.rsr`	restraints file
`job.sch`	VTFM schedule file
`job.B9999????`	PDB atom file(s) for the model(s) of the target sequence
`job.V9999????`	violation profiles for the model(s)
`job.D9999????`	progress of optimization
`job.BL9999????`	optional loop model(s)
`job.DL9999????`	progress of optimization for loop model(s)
`job.IL9999????`	initial structures for loop model(s)

The main MODELLER routines used in each step are given in parentheses.

1. Read and check the alignment between the target sequence and the template structures (**Alignment.append()** and **Alignment.check()**).
2. Calculate restraints on the target from its alignment with the templates:

(a) Generate molecular topology for the target sequence (**Model.generate_topology()**). Disulfides in the target are assigned here from the equivalent disulfides in the templates (**Model.patch_ss_templates()**). Any user defined patches are also done here (as defined in the **AutoModel.special_patches()** routine).

(b) Calculate coordinates for atoms that have equivalent atoms in the templates as an average over all templates (**Model.transfer_xyz()**) (alternatively, read the initial coordinates from a file).

(c) Build the remaining unknown coordinates using internal coordinates from the CHARMM topology library (**Model.build()**).

(d) Write the initial model to a file with extension `.ini` (**Model.write()**).

(e) Generate stereochemical, homology-derived, and special restraints (**Restraints.make()**) (alternatively, skip this and assume the restraints file already exists):

stereochemical	restraint_type = 'bond angle dihedral improper'
mainchain dihedrals Φ, Ψ	restraint_type = 'phi-psi_binormal'
mainchain dihedral ω	restraint_type = 'omega_dihedral'
sidechain dihedral χ_1	restraint_type = 'chi1_dihedral'
sidechain dihedral χ_2	restraint_type = 'chi2_dihedral'
sidechain dihedral χ_3	restraint_type = 'chi3_dihedral'
sidechain dihedral χ_4	restraint_type = 'chi4_dihedral'
mainchain CA–CA distance	restraint_type = 'distance'
mainchain N–O distance	restraint_type = 'distance'
sidechain–mainchain distance	restraint_type = 'distance'
sidechain–sidechain distance	restraint_type = 'distance'
ligand distance restraints	**AutoModel.nonstd_restraints()** routine
user defined	**AutoModel.special_restraints()** routine
non-bonded pairs distance	restraint_type = 'sphere'; calculated on the fly

(f) Write all restraints to a file with extension `.rsr` (**Restraints.write()**).

3. Calculate model(s) that satisfy the restraints as well as possible. For each model:

   (a) Generate the optimization schedule for the variable target function method (VTFM).

   (b) Read the initial model (usually from the `.ini` file from 2.d) (**Model.read()**).

   (c) Randomize the initial structure by adding a random number between ±AutoModel.deviation angstroms to all atomic positions (**Selection.randomize_xyz()**).

   (d) Optimize the model:

   - Partially optimize the model by VTFM; Repeat the following steps as many times as specified by the optimization schedule:
     - Select only the restraints that operate on the atoms that are close enough in sequence, as specified by the current step of VTFM (**Restraints.pick()**).
     - Optimize the model by conjugate gradients, using only currently selected restraints (**ConjugateGradients()**).
   - Refine the model by simulated annealing with molecular dynamics, if so selected:
     - Do a short conjugate gradients optimization (**ConjugateGradients()**).
     - Increase temperature in several steps and do molecular dynamics optimization at each temperature (**MolecularDynamics()**).
     - Decrease temperature in several steps and do molecular dynamics optimization at each temperature (**MolecularDynamics()**).
     - Do a short conjugate gradients optimization (**ConjugateGradients()**).

   (e) Calculate the remaining restraint violations and write them out (**Selection.energy()**).

   (f) Write out the final model to a file with extension `.B9999????.pdb` where `????` indicates the model number (**Model.write()**). Also write out the violations profile.

   (g) Superpose the models and the templates, if so selected by AutoModel.final_malign3d = `True`, and write them out (**Alignment.append_model()**, **Alignment.malign3d()**).

   (h) Do loop modeling if so selected using the `LoopModel` class.

A.5 Loop modeling method

The loop modeling method first takes the generated model, and selects all standard residues around gaps in the alignment for additional loop modeling. (To select a different region for modeling, simply redefine the **LoopModel.select_loop_atoms()** routine to select the relevant atoms.) An initial loop conformation is then generated by simply positioning the atoms of the loop with uniform spacing on the line that connects the main-chain carbonyl oxygen and amide nitrogen atoms of the N- and C-terminal anchor regions respectively (to change this, override the **LoopModel.build_ini_loop()** method), and this model is written out to a file with the `.IL` extension.

Next, a number of loop models are generated from LoopModel.loop.starting_model to LoopModel.loop.ending_model. Each takes the initial loop conformation and randomizes it by $\pm 5\AA$ in each of the Cartesian directions. The model is then optimized thoroughly twice, firstly considering only the loop atoms and secondly with these atoms "feeling" the rest of the system. The loop optimization relies on an atomistic distance-dependent statistical potential of mean force for nonbond interactions [Melo & Feytmans, 1997]. This classifies all amino acid atoms into one of 40 atom classes (as defined in `$LIB/atmcls-melo.lib`) and applies a potential as MODELLER cubic spline restraints (as defined in `$LIB/melo-dist1.lib`). No homology-derived restraints are used during this procedure. Each loop model is written out with the `.BL` extension.

For more information, please consult the loop modeling paper [Fiser *et al.*, 2000] or look at the loop modeling class itself in `modlib/modeller/automodel/loopmodel.py`.

Appendix B

File formats

B.1 Alignment file (PIR)

The preferred format for comparative modeling is related to the PIR database format:

```
C; A sample alignment in the PIR format; used in tutorial

>P1;5fd1
structureX:5fd1:1    :A:106  :A:ferredoxin:Azotobacter vinelandii: 1.90: 0.19
AFVVTDNCIKCKYTDCVEVCPVDCFYEGPNFLVIHPDECIDCALCEPECPAQAIFSEDEVPEDMQEFIQLNAELA
EVWPNITEKKDPLPDAEDWDGVKGKLQHLER*

>P1;1fdx
sequence:1fdx:1    :A:54   :A:ferredoxin:Peptococcus aerogenes: 2.00:-1.00
AYVINDSC--IACGACKPECPVNIIQGS--IYAIDADSCIDCGSCASVCPVGAPNPED-----------------
-------------------------------*
```

The first line of each sequence entry specifies the protein code after the `>P1;` line identifier. The line identifier must occur at the beginning of the line. For example, `1fdx` is the protein code of the first entry in the alignment above. The protein code corresponds to the Sequence.code variable. Conventionally, this code is the PDB code followed by an optional one-letter chain ID, but this is not required; codes can be any unique identifier. If the code in the alignment file is made up of several words (separated by spaces or tabs), only the first is read in; the remainder are treated by MODELLER as comments.

The second line of each entry contains information necessary to extract atomic coordinates of the segment from the original PDB, mmCIF, or BinaryCIF coordinate set. The fields in this line are separated by colon characters, ':'. The fields are as follows:

Field 1: A specification of whether or not 3D structure is available and of the type of the method used to obtain the structure (`structureX`, X-ray; `structureN`, NMR; `structureM`, model; `sequence`, sequence). Only `structure` is also a valid value.

Field 2: The PDB, mmCIF, or BinaryCIF filename or code. While the protein code in the first line of an entry, which is used to identify the entry, must be unique for all proteins in the file, the name in this field, which is used to get structural data, does not have to be unique. It can be a full file name with path (*e.g.*, '/home/foo/pdbs/mystructure.pdb'), a file name without a path (*e.g.*, 'mystructure.pdb' or 'mystructure.cif'), or a PDB code (*e.g.*, '1abc'; MODELLER will automatically convert the code to a filename by adding '.pdb', '.cif' or '.ent' file extensions as necessary, and/or a 'pdb' prefix). In the latter two cases, where no path is given, MODELLER will search in the directories specified by IOData.atom_files_directory to find PDB, mmCIF or BinaryCIF files.

Fields 3–6: The residue and chain identifiers (see below) for the first (fields 3–4) and last residue (fields 5–6) of the sequence in the subsequent lines. There is no need to edit the coordinate file if a contiguous sequence of residues is required — simply specify the beginning and ending residues of the required contiguous region of the chain. If the beginning residue is not found, no segment is read in. If the ending residue identifier is not found in the coordinate file, the last residue in the coordinate file is used. By default, the whole file is read in.

The unspecified beginning and ending residue numbers and chain id's for a `structure` entry in an alignment file are taken automatically from the corresponding atom file, if possible. The first matching sequence in the atom file that also satisfies the explicitly specified residue numbers and chain id's is used. A residue number is not specified when a blank character or a dot, '.', is given. A chain id is not specified when a dot, '.', is given. This slight difference between residue and chain id's is necessary because a blank character is a valid chain id.

Field 7: Protein name. Optional.

Field 8: Source of the protein. Optional.

Field 9: Resolution of the crystallographic analysis. Optional.

Field 10: R-factor of the crystallographic analysis. Optional.

A residue identifier is simply the 5-letter PDB residue number (including insertion code, if any), and a chain identifier the PDB chain code[1]. For example, `'10I:A'` is residue number `'10I'` in chain `'A'`, and `'6:'` is residue number `'6'` in a chain without a name.

The residue number for the first position (resID1) in the model_segment range `'resID1:chainID1 resID2:chainID2'` can be either a real residue number or `'FIRST'` (which indicates the first residue in a matching chain). The residue number for the second position (resID2) in the model_segment range can be either: (1) a real residue number; (2) `'LAST'` (which indicates the last residue in a matching chain); (3) `'+nn'` (which requests the total number of residues to read, in which case the chain id is ignored); or `'END'` (which indicates the last residue in the PDB file). The chain id for either position in the model_segment range (chainID1 or chainID2) can be either: (1) a real chain id (including a blank/space/null/empty); or `'@'`, which matches any chain id.

Examples, assuming a two chain PDB file (chains A and B):

- `'15:A 75:A'` reads residues 15 to 75 in chain A.
- `'FIRST:@ 75:@'` reads the first 75 residues in chain A (the first chain).
- `'FIRST:@ LAST:@'` reads all residues in chain A, assuming `'FIRST'` is not a real number of the non-first residue in chain A.
- `'FIRST:@ +125:'` reads a total of 125 residues, regardless of the PDB numbering, starting from the first residue in chain A.
- `'10:@ LAST:'` reads all residues from 10 in chain A to the end of the file (chain id for the last residue is irrelevant), again assuming `'LAST'` is not a real residue number of a non-last residue.
- `'FIRST:@ END:'` reads the whole file no matter what, the chainID is ignored completely.

For the selection_segment the string containing `'@'` will match any residue number and chainID. For example, `'@:A'` is the first residue in chain `'A'` and `'@:@'` is the first residue in the coordinate file. The last chain can not be specified in a general way, except if it is the last residue in the file.

When an alignment file is used in conjunction with structural information, the first two fields must be filled in; the rest of them can be empty. If the alignment is not used in conjunction with structural data, all but the first field can be empty. This means that in comparative modeling, the template structures must have at least the first two fields specified while the target sequence must only have the first field filled in. Thus, a simple second line of an entry in an alignment file in the `'PIR'` format is

[1] for PDB format, this is a 1-letter chain ID; for mmCIF or BinaryCIF format, this is the author-provided chain/asym ID (`_atom_site.auth_asym_id`) if available, otherwise the canonical asym ID (`_atom_site.label_asym_id`), both of which can be multiple characters in length

```
structure:pdb_file:.:.:.:.:::
```

This entry will result in reading from PDB, mmCIF, or BinaryCIF file `pdb_file` the structure segment corresponding to the sequence in the subsequent lines of the alignment entry.

Each sequence must be terminated by the terminating character, '*'.

When the first character of the sequence line is the terminating character, '*', the sequence is obtained from the specified PDB, mmCIF, or BinaryCIF coordinate file (Section 5.1.3).

Chain breaks are indicated by '/'. There should not be more than one chain break character to indicate a single chain break (use gap characters instead, '-'). All residue types specified in `$RESTYP_LIB`, but not patching residue types, are allowed; there are on the order of 100 residue types specified in the `$RESTYP_LIB` library. To add your own residue types to this library, see Section 3.1, Question 8.

The alignment file can contain any number of blank lines between the protein entries. Comment lines can occur outside protein entries and must begin with the identifiers 'C;' or 'R;' as the first two characters in the line.

An alignment file is also used to input non-aligned sequences.

B.2 Restraints file

The first line of a restraints file should read `'MODELLER5 VERSION: MODELLER FORMAT'`. (`'USER'` format is no longer supported.)

After this, there is one entry per line. The format is free, except that the first character has to be at the beginning of the line. When the line starts with `'R'`, it contains a restraint, `'E'` indicates a pair of atoms to be excluded from the calculation of the dynamic non-bonded pairs list, `'P'` indicates a pseudo atom definition (Section 5.3.2), `'S'` a symmetry restraint, and `'B'` a rigid body.

B.2.1 Restraints

An 'R' line should look like:

```
R Form Modality Feature Group Numb_atoms Numb_parameters Numb_Feat (Modal2 Feat2 NumAt2 ...) Atom_indices
```

These parameters encode the restraints information as given in Section 5.3.

Here, `Form` is the restraint form type (see Table B.1). `Modality` is an integer argument to `Form`, and specifies the number of single Gaussians in a poly-Gaussian pdf, number of terms in a multiple binormal, periodicity n of the cosine in the cosine potential, and the number of spline points for cubic splines. `Feature` is the feature that this restraint acts on (see Table B.2.) `Group` is the physical feature type, and should be an index from Table 6.1. `Numb_atoms` is the total number of atoms this restraint acts on, `Numb_parameters` is the number of defined parameters, and `Numb_Feat` is the number of features the restraint acts on. `Numb_Feat` is typically 1, except for the multiple binormal (where it should be 2) and ND spline (where it can be any number). In cases where `Numb_Feat` is greater than 1, the modality, feature type, and number of atoms of each subsequent feature should be listed in order after `Numb_Feat`. (Note that `Numb_atoms` is the number of atoms acted on by the entire restraint, while `NumAt2` refers just to the atoms acted on by the 2nd feature.) Finally, the integer atom indices and floating point parameters are listed.

For example,

```
R    3   1   1   1   2   2   1  437   28        1.5000    0.1000
```

will create a Gaussian restraint on the distance between atoms 437 and 28, with mean of 1.5 and standard deviation of 0.1.

B.2.2 Excluded pairs

An 'E' line should look like:

Numeric form	Form type
1	`forms.LowerBound`
2	`forms.UpperBound`
3	`forms.Gaussian`
4	`forms.MultiGaussian`
5	`forms.LennardJones`
6	`forms.Coulomb`
7	`forms.Cosine`
8	`forms.Factor`
9	`forms.MultiBinormal`
10	`forms.Spline` or `forms.NDSpline`
50+	user-defined restraint forms

Table B.1: *Numerical restraint forms*

Numeric feature	Feature type
1	`features.Distance`
2	`features.Angle`
3,4	`features.Dihedral`
6	`features.MinimalDistance`
7	`features.SolventAccess`
8	`features.Density`
9	`features.XCoordinate`
10	`features.YCoordinate`
11	`features.ZCoordinate`
12	`features.DihedralDiff`
50+	user-defined feature types

Table B.2: *Numerical feature types*

```
E Atom_index_1 Atom_index_2
```

where the two numeric atom indices are given (see Section 5.3.3).

For example,

```
E    120 540
```

would exclude the nonbond interaction between atoms 120 and 540 from the list.

B.2.3 Pseudo atoms

A ’P’ line should look like:

```
P Pseudo_atom_index Pseudo_atom_type Numb_real_atoms Real_atom_indices
```

These parameters encode the pseudo atom information as given in Section 5.3.2. `Pseudo_atom_index` is the atom index, which should be a number between NATM+1 and NATM+NPSEUDO, where NATM is the number of real atoms in the model, and NPSEUDO the number of pseudo atoms. `Pseudo_atom_type` is the numerical pseudo atom type, as given in Table B.3. The pseudo atom is defined as an average of `Numb_real_atoms`, of indices `Real_atom_indices`. For example,

```
P    144   1   3    120 121 122
```

creates a pseudo atom at index 144, which is a gravity center of the 3 atoms 120, 121 and 122.

Numeric pseudo atom type	Pseudo atom type
1	`pseudo_atom.GravityCenter`
2	`virtual_atom.CH1`
3	`virtual_atom.CH1A`
4	`pseudo_atom.CH2`
5	`virtual_atom.CH2`
6	`pseudo_atom.CH31`
7	`pseudo_atom.CH32`

Table B.3: *Numerical pseudo atom types*

B.2.4 Symmetry restraints

An 'S' line should look like:

```
S Num_Pairs Atom_indices1 Atom_indices2 Pair_weights
```

These parameters encode a single symmetry restraint, as given in Section 5.3.5. `Num_Pairs` is the number of atom pairs to be constrained. `Atom_indices1` and `Atom_indices2` are the numeric indices of the atoms in each pair, while `Pair_weights` are the weights for each pair.

For example,

```
S   2   4   8   10   12   1.0   0.5
```

creates a symmetry restraint in which atoms 4 and 10 are constrained to be similar to each other with weight 1.0, while atoms 8 and 12 are constrained with weight 0.5.

B.2.5 Rigid bodies

A 'B' line should look like:

```
B Scale_factor Atom_indices
```

These parameters encode the rigid body information as given in Section 5.3.4. `Atom_indices` are the numeric indices of all atoms in the rigid body, and `Scale_factor` the scaling factor from system state to body orientation.

For example,

```
B   5   10  25   30
```

creates a rigid body containing atoms 5, 10, 25 and 30.

B.3 Profile file

The format of the profile file (text) is as follows:

```
# Number of sequences:      4
# Length of profile  :     20
# N_PROF_ITERATIONS  :      3
# GAP_PENALTIES_1D   :   -900.0   -50.0
# MATRIX_OFFSET      :    0.0
# RR_FILE            : ${MODINSTALLCVS}/modlib//as1.sim.mat
```

```
    1 2ctx                                     X     0    71     1    71     0     0     0    0.     0.0      IRCFITPDITS---KDCPN-
    2 2abx                                     X     0    74     1    74     0     0     0    0.     0.0      IVCHTTATIPS-SAVTCPPG
    3 2nbt                                     X     0    66     1    66     0     0     0    0.     0.0      RTCLISPSS---TPQTCPNG
    4 1fas                                     X     0    61     1    61     0     0     0    0.     0.0      TMCYSHTTTSRAILTNCG--
```

The first six lines begin with a '#' in the first column and give a few general details of the profile.

The first line gives the number of sequences in the profile. The line should be in the following format: '(24x,i6)'.

The second line gives the number of positions in the profile. This should be in '(24x,i6)' format also.

The third line gives the value of the n_prof_iterations variable. The fourth line gives the value of the gap_penalties_1d variable. The fifth line gives the value of the matrix_offset variable. The sixth line gives the value of the rr_file variable.

The number of sequences in the profile and its length are used to allocate memory for the profile arrays, so they should provide an accurate description of the profile.

The values of the variables described in lines 3 through 6 are not used internally by MODELLER. But the **Profile.read()** command expects to find a total of six header lines. These records represent useful information when **Profile.build()** was used to construct the profile.

The remaining lines consist of the alignment of the sequences in the profile. The format of these lines is of the form: '(i5,1x,a40,1x,a1,1x,7(i5,1x),f5.0,1x,g10.2,1x,32767a1)'

The various columns that precede the sequence are:

1. The index number of the sequence in the profile.
2. The code of the sequence (similar to Sequence.code).
3. The type of sequence ('S' for sequence, 'X' for structure). This depends on the original source of the sequences. (See **Alignment.to_profile()** and **SequenceDB.read()**).
4. The iteration in which the sequence was selected as significant. (See **Profile.build()**).
5. The length of the database sequence.
6. The starting position of the target sequence in the alignment.
7. The ending position of the target sequence in the alignment.
8. The starting position of the database sequence in the alignment.
9. The ending position of the database sequence in the alignment.
10. The number of equivalent positions in the alignment.
11. The sequence identity of between the target sequence and the database sequence.
12. The e-value of the alignment. (See **Profile.build()**).
13. The sequence alignment.

Many of the fields described above are valid only when the profile that is written out is the result of **Profile.build()**.

B.4 Binary files

Binary files are standard HDF5 files. These files are in a very compact, non-human-readable format, and are thus very rapid for MODELLER to access. However, unlike some binary files, they are machine-independent (*e.g.*, they can be moved from a Windows machine to a Mac or a Linux box without problems). They can also be accessed using standard HDF5 tools.

Note that the binary files used by MODELLER 8v2 and earlier are not compatible with the current format. If you have any such files, they must be regenerated from their corresponding text files.

Appendix C

Converting Top scripts from old Modeller versions

Previous versions of Modeller used Top as their scripting language. Top is a language similar in syntax to Fortran, which is also used by Asgl. For increased power and flexibility, interoperability with other programs, and improved ease of use, Modeller now uses Python for its control language.

C.1 Running old scripts unchanged

For compatibility with old codes, Modeller will still run most Top scripts. By default, the program expects to read new-style Python scripts, but if the script file extension ends in `'.top'` (as in previous versions), it will be assumed to be a Top file and will be run as such. Note, however, that this behavior is deprecated and will probably be removed entirely in a later release. Also bear in mind that some newer commands and features will not be available in Top; thus, it is recommended that you convert your Top scripts to Python.

If you do wish to run old scripts unchanged, please note that the **GO_TO** function is no longer present in the Top language. If you have code which makes use of this function, you should use the **EXIT** and **CYCLE** flow control statements instead, which either terminate a **DO** loop or skip to its next iteration. Single-line **IF** statements are no longer supported either; you should use the **ELSE** and **END_IF** statements to build multi-line **IF** clauses instead. See the scripts in the `bin` directory for examples.

C.2 Converting Top scripts to Python

C.2.1 Top commands and variables

Top variables come in four varieties — strings, reals, integers, logicals — as either single values or lists. The equivalents in Python are the standard `'str'`, `'float'`, `'int'` and `'bool'` types, respectively, while lists can be represented as `'list'` or `'tuple'` objects. In essence, this means that strings must be quoted, logical `on` and `off` become bool `True` and `False` respectively, and lists must be comma-separated and enclosed in parentheses. Also, the '=' operator is not mandatory in Top, but it is in Python. See the Python documentation for more information. For example, consider the following assignments in Top:

```
SET STRVAR = foo                   # Set a string variable (quotes not required)
SET REALVAR 3.4                    # Set a real variable (= not required either)
SET INTVAR = 4                     # Set an integer variable
SET LOGVAR = on                    # Set a logical variable
SET INTLIST = 1 1 3 5              # Set a list of integers
SET STRLIST = 'one' 'two' 'three'  # Set a list of strings
```

The equivalent Python code would be:

```
STRVAR = 'foo'                     # Set a string variable
REALVAR = 3.4                      # Set a real variable
INTVAR = 4                         # Set an integer variable
LOGVAR = True                      # Set a logical variable
INTLIST = (1, 1, 3, 5)             # Set a list of integers
STRLIST = ('one', 'two', 'three')  # Set a list of strings
```

Variables in Top are case-insensitive; that is, the names GAP_PENALTIES_1D and gap_penalties_1d refer to the same variable. (Upper case is usually used, but this is just by convention.) Python variables are case-sensitive, so these two names refer to different variables. For consistency with other codes, all Python commands and variables used by Modeller are **lower-case**.

All variables in Top are global; that is, once they are set, their value is kept for the rest of the program (or until changed again). This is irrespective of whether the variable is set while calling a Top command, or whether an explicit **SET** command is used. For example, this Top script:

```
ALIGN GAP_PENALTIES_1D = 900 50
```

will behave identically to this code:

```
SET GAP_PENALTIES_1D = 900 50
ALIGN
```

In Python, on the other hand, each command takes a number of arguments. For example, the **align()** command takes a gap_penalties_1d argument. The value of this argument affects only this call to **align()**, *i.e.* it is a local variable. Thus, the exact equivalent to both of the Top scripts above would be:

```
aln.align(gap_penalties_1d=(900, 50))
```

where `'aln'` is an `Alignment` object (see section C.2.2). This only sets the 1D gap penalties for this invocation of **align()**, and so is less likely to cause problems later in your script. If you want to call a routine several times with the same set of arguments, it is recommended that you save the arguments in local Python variables, use subroutines or classes, or use `'for'` loops.

C.2.2 Top models and alignments

In Top, commands may operate implicitly on one or more of the standard models or alignments in memory. For example, **ALIGN** always operates on ALIGNMENT1, **READ_MODEL** always operates on MODEL1, and **READ_MODEL2** always operates on MODEL2. MODEL2 is a 'cut-down' model, used only for some operations (such as **SUPERPOSE**) and cannot be used to build full models, for example.

In Python, the models and alignments (and sequence databases, densities, *etc.*) are explicit, and are represented by classes. You can have as many models or alignments as you like, provided you have enough memory. Commands are simply methods of these classes. For example, consider the following:

```
env = Environ()
aln = Alignment(env)
aln.align(gap_penalties_1d=(900, 50))
```

This creates a new instance of the `Environ` class (as above, this is used to provide default variables and the like), and calls it `'env'` (you can call it whatever you like). This is then used to create a new `Alignment` class object, called `'aln'`. The following **align()** command then operates on the `'aln'` alignment object. (Note, however, that new alignments are empty, so this example wouldn't do anything interesting.)

C.2.3 Top to Python correspondence

Please use the tables below to see which Python commands and variables correspond to old Top commands and variables. Variables which are not listed in these tables have the same names as the old Top equivalents (albeit in lower case).

TOP command	Python equivalent
ADD_RESTRAINT	**Restraints.add()**
ALIGN	**Alignment.align()**
ALIGN2D	**Alignment.align2d()**
ALIGN3D	**Alignment.align3d()**
ALIGN_CONSENSUS	**Alignment.consensus()**
ALN_TO_PROF	**Alignment.to_profile()**
BUILD_MODEL	**Model.build()**
BUILD_PROFILE	**Profile.build()**
CALL	use Python subroutines
CHECK_ALIGNMENT	**Alignment.check()**
CLOSE	use Python file object
COLOR_ALN_MODEL	**Model.color()**
COMPARE	**Alignment.compare_structures()**
COMPARE_ALIGNMENTS	**Alignment.compare_with()**
CONDENSE_RESTRAINTS	**Restraints.condense()**
DEBUG_FUNCTION	**Selection.debug_function()**
DEFINE_INTEGER	use Python 'int' variables
DEFINE_LOGICAL	use Python 'bool' variables
DEFINE_REAL	use Python 'float' variables
DEFINE_STRING	use Python 'str' variables
DEFINE_SYMMETRY	**Restraints.add()**
DELETE_ALIGNMENT	del(Alignment)
DELETE_FILE	**modfile.delete()**
DELETE_RESTRAINT	**Restraints.unpick()**
DENDROGRAM	**Environ.dendrogram()**
DESCRIBE	**Alignment.describe()**
DO	use Python while or for loops
EDIT_ALIGNMENT	**Alignment.edit()**
EM_GRID_SEARCH	**Density.grid_search()**
END_SUBROUTINE	use Python subroutines
ENERGY	**Selection.energy()**
EXIT	use Python while or for loops
EXPAND_ALIGNMENT	use **Alignment.append_model()** in a loop
GENERATE_TOPOLOGY	**Model.generate_topology()**
ID_TABLE	**Alignment.id_table()**
IF	use Python if statement
INCLUDE	use Python import statement
INQUIRE	**modfile.inquire()**
IUPAC_MODEL	**Model.to_iupac()**
MAKE_CHAINS	**Model.make_chains()**
MAKE_REGION	**Model.make_region()**
MAKE_RESTRAINTS	**Restraints.make()**
MAKE_SCHEDULE	**Schedule.make_for_model()**
MAKE_TOPOLOGY_MODEL	**Topology.make()**
MALIGN	**Alignment.malign()**
MALIGN3D	**Alignment.malign3d()**
MUTATE_MODEL	**Selection.mutate()**
OPEN	use Python file object
OPERATE	use Python arithmetic
OPTIMIZE	**ConjugateGradients()** or **MolecularDynamics()**, or `<Schedule>` objects
ORIENT_MODEL	**Model.orient()**
PATCH	**Model.patch()**
PATCH_SS_MODEL	**Model.patch_ss()**
PATCH_SS_TEMPLATES	**Model.patch_ss_templates()**

PICK_ATOMS	Use `<Selection>` objects
PICK_HOT_ATOMS	**Selection.hot_atoms()**
PICK_RESTRAINTS	**Restraints.pick()**
PRINCIPAL_COMPONENTS	**Environ.principal_components()**
PROFILE_PROFILE_SCAN	**Profile.scan()**
PROF_TO_ALN	**Alignment.append_profile()** or **Profile.to_alignment()**
RANDOMIZE_XYZ	**Selection.randomize_xyz()**
READ	use Python file object
READ_ALIGNMENT	**Alignment.append()**
READ_ALIGNMENT2	**Alignment.append()**
READ_ATOM_CLASSES	**GroupRestraints()**
READ_DENSITY	**Density.read()**
READ_MODEL	**Model.read()**
READ_MODEL2	**Model.read()**
READ_PARAMETERS	**Parameters.read()** or **GroupRestraints.append()**
READ_PROFILE	**Profile.read()**
READ_RESTRAINTS	**Restraints.append()**
READ_RESTYP_LIB	**Environ()**
READ_SCHEDULE	Use `<Schedule>` objects
READ_SEQUENCE_DB	**SequenceDB.read()**
READ_TOPOLOGY	**Topology.read()** or **Topology.append()**
REINDEX_RESTRAINTS	**Restraints.reindex()**
RENAME_SEGMENTS	**Model.rename_segments()**
REORDER_ATOMS	**Model.reorder_atoms()**
RESET	do not use
RETURN	use Python subroutines
ROTATE_DIHEDRALS	**Selection.rotate_dihedrals()**
ROTATE_MODEL	**Selection.translate()**, **Selection.transform()**, or **Selection.rotate_origin()**
SALIGN	**Alignment.salign()**
SEGMENT_MATCHING	**Alignment.segment_matching()**
SEQFILTER	**SequenceDB.filter()**
SEQUENCE_COMPARISON	**Alignment.compare_sequences()**
SEQUENCE_SEARCH	**SequenceDB.search()**
SEQUENCE_TO_ALI	**Alignment.append_model()**
SET	use Python variables
SPLINE_RESTRAINTS	**Restraints.spline()**
STOP	do not use
STRING_IF	use Python if statement
STRING_OPERATE	use Python arithmetic
SUBROUTINE	use Python subroutines
SUPERPOSE	**Selection.superpose()**
SWITCH_TRACE	**actions.Trace()**
SYSTEM	**Environ.system()**
TIME_MARK	**info.time_mark()**
TRANSFER_RES_NUMB	**Model.res_num_from()**
TRANSFER_XYZ	**Model.transfer_xyz()**
UNBUILD_MODEL	**Selection.unbuild()**
WRITE	use Python file object
WRITE_ALIGNMENT	**Alignment.write()**
WRITE_DATA	**Model.write_data()**
WRITE_MODEL	**Model.write()**
WRITE_MODEL2	**Model.write()**
WRITE_PDB_XREF	use `<Residue>` objects
WRITE_PROFILE	**Profile.write()**
WRITE_RESTRAINTS	**Restraints.write()**
WRITE_SCHEDULE	**Schedule.write()**

WRITE_SEQUENCE_DB	**SequenceDB.write()**
WRITE_TOP	do not use
WRITE_TOPOLOGY_MODEL	**Topology.write()**

Table C.1: *Correspondence between* TOP *and Python commands.*

TOP variable	Python equivalent
ALIGN_CODES	Sequence.code
ATOM_FILES	Sequence.atom_file
ATOM_FILES_DIRECTORY	IOData.atom_files_directory
CONTACT_SHELL	EnergyData.contact_shell
COULOMB_SWITCH	EnergyData.coulomb_switch
COVALENT_CYS	EnergyData.covalent_cys
DYNAMIC_ACCESS	do not use
DYNAMIC_COULOMB	EnergyData.dynamic_coulomb
DYNAMIC_LENNARD	EnergyData.dynamic_lennard
DYNAMIC_MODELLER	EnergyData.dynamic_modeller
DYNAMIC_PAIRS	set automatically; do not use
DYNAMIC_SPHERE	EnergyData.dynamic_sphere
EXCL_LOCAL	EnergyData.excl_local
HETATM_IO	IOData.hetatm
HYDROGEN_IO	IOData.hydrogen
LENNARD_JONES_SWITCH	EnergyData.lennard_jones_switch
MOLPDF	return value from **Selection.energy()**
NLOGN_USE	EnergyData.nlogn_use
NONBONDED_SEL_ATOMS	EnergyData.nonbonded_sel_atoms
NUMB_OF_SEQUENCES	len(Alignment)
N_SCHEDULE	len(schedule)
OUTPUT_CONTROL	use **log.level()**
RADII_FACTOR	EnergyData.radii_factor
RELATIVE_DIELECTRIC	EnergyData.relative_dielectric
SCHEDULE_STEP	do not use
SPHERE_STDV	EnergyData.sphere_stdv
TOPOLOGY_MODEL	Topology.submodel
UPDATE_DYNAMIC	EnergyData.update_dynamic
WATER_IO	IOData.water

Table C.2: *Correspondence between* TOP *and Python variables.*

Bibliography

Braun, W. & Gõ, N. (1985). *J. Mol. Biol.* **186**, 611–626.

Brünger, A. T. (1992). *X-PLOR Manual Version 3.1.* Yale University New Haven, Connecticut.

Dong, G. Q., Fan, H., Schneidman-Duhovny, D., Webb, B., & Sali, A. (2013). *Bioinformatics,* **29**, 3158–3166. (Also available online).

Felsenstein, J. (1985). *Evolution,* **39**, 783–791.

Fiser, A., Do, R. K. G., & Šali, A. (2000). *Protein Sci.* **9**, 1753–1773. (Also available online).

Gallicchio, E. & Levy, R. M. (2004). *J. Comp. Chem.* **25**, 479–499.

Goldstein, H. (1980). *Classical Mechanics, 2nd edition.* Reading, Massachusetts: Addison-Wesley Publishing Company.

Gotoh, O. (1982). *J. Mol. Biol.* **162**, 705–708.

Hubbard, T. J. P. & Blundell, T. L. (1987). *Protein Eng.* **1**, 159–171.

IUPAC-IUB (1970). *Biochem.* **9**, 3471–3479.

John, B. & Šali, A. (2003). *Nucl. Acids Res.* **31**, 3982–3992. (Also available online).

Kabsch, W. & Sander, C. (1983). *Biopolymers,* **22**, 2577–2637.

Karlin, S. & Altschul, S. F. (1990). *Proc. Natl. Acad. Sci. USA,* **87**, 2264–2268.

Kendrew, J. C., Klyne, W., Lifson, S., Miyazawa, T., Némethy, G., Phillips, D. C., Ramachandran, G. N., & Scheraga, H. (1970). *J. Mol. Biol.* **52**, 1–17.

Loncharich, R. J., Brooks, B. R., & Pastor, R. W. (1992). *Biopolymers,* **32**, 523–535.

MacKerell, Jr., A. D., Bashford, D., Bellott, M., Dunbrack Jr., R. L., Evanseck, J. D., Field, M. J., Fischer, S., Gao, J., Guo, H., Ha, S., Joseph-McCarthy, D., Kuchnir, L., Kuczera, K., Lau, F. T. K., Mattos, C., Michnick, S., Ngo, T., Nguyen, D. T., Prodhom, B., Reiher, III, W. E., Roux, B., Schlenkrich, M., Smith, J. C., Stote, R., Straub, J., Watanabe, M., Wiorkiewicz-Kuczera, J., Yin, D., & Karplus, M. (1998). *J. Phys. Chem. B,* **102**, 3586–3616.

Madhusudhan, M. S., Martí-Renom, M. A., Sanchez, R., & Šali, A. (2006). *Prot. Eng. Des. & Sel.* **19**, 129–133. (Also available online).

Madhusudhan, M. S., Webb, B. M., Martí-Renom, M. A., Eswar, N., & Šali, A. (2009). *Prot. Eng. Des. & Sel.* **22** (9), 569–574. (Also available online).

Martí-Renom, M. A., Madhusudhan, M. S., & Šali, A. (2004). *Protein Sci.* **13**, 1071–1087. (Also available online).

Melo, F. & Feytmans, E. (1997). *J. Mol. Biol.* **267**, 207–222.

Melo, F., Sánchez, R., & Šali, A. (2002). *Protein Sci.* **11**, 430–448. (Also available online).

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). *J. Chem. Phys.* **21**, 1087–1092.

Needleman, S. B. & Wunsch, C. D. (1970). *J. Mol. Biol.* **48**, 443–453.

Nicholls, A., Sharp, K. A., & Honig, B. (1991). *Proteins,* **11**, 281–296.

Pearson, W. (1998). *J. Mol. Biol.* **276**, 71–84.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1992). *Numerical Recipes, 2nd edition.* Cambridge: Cambridge University Press.

Rapaport, D. C. (1997). *The Art of Molecular Dynamics Simulation.* Cambridge, UK: Cambridge University Press.

Richards, F. M. & Kundrot, C. E. (1988). *Proteins,* **3**, 71–84.

Richmond, T. J. & Richards, F. M. (1978). *J. Mol. Biol.* **119**, 537–555.

Šali, A. & Blundell, T. L. (1990). *J. Mol. Biol.* **212**, 403–428. (Also available online).

Šali, A. & Blundell, T. L. (1993). *J. Mol. Biol.* **234**, 779–815. (Also available online).

Šali, A. & Overington, J. (1994). *Protein Sci.* **3**, 1582–1596. (Also available online).

Sankoff, D. & Kruskal, J. B. (1983). *Time warps, string edits, and macromolecules: The theory and practice of sequence comparison.* Reading, MA: Addison-Wesley Publishing Company.

Sellers, P. H. (1974). *J. Comb. Theor.* **A16**, 253–258.

Shanno, D. F. & Phua, K. H. (1980). *ACM Trans. Math. Soft.* **6**, 618–622.

Shanno, D. F. & Phua, K. H. (1982). In: *Collected algorithms from ACM. Trans. Math. Software* volume 2(1).

Shen, M.-Y. & Šali, A. (2006). *Protein Sci.* **15**, 2507–2524. (Also available online).

Smith, T. F. & Waterman, M. S. (1981). *J. Mol. Biol.* **147**, 195–197.

Subbiah, S., Laurents, D. V., & Levitt, M. (1993). *Curr. Biol.* **3**, 141–148.

Sutcliffe, M. J., Haneef, I., Carney, D., & Blundell, T. L. (1987). *Protein Eng.* **1**, 377–384.

Topf, M., Baker, M. L., John, B., Chiu, W., & Šali, A. (2005). *J. Struct. Biol.* **149**, 191–203. (Also available online).

van Schaik, R. C., Berendsen, H. J., & Torda, A. E. (1993). *J. Mol. Biol.* **234**, 751–762.

Verlet, J. (1967). *Phys. Rev.* **159**, 98–103.

Wu, X. & Brooks, B. R. (2003). *Chem. Phys. Lett.* **381**, 512–518.

Wu, X. & Wang, S. (1999). *J. Chem. Phys.* **110**, 9401–9410.