MODELLER
A Program for Protein Structure Modeling
Release 9v4, r6262

Andrej Šali

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

email: modeller-care AT salilab.org
URL: http://salilab.org/modeller/

2008/05/05
# Contents

**Copyright notice**

**Acknowledgments**

## 1 Introduction

1.1 What is MODELLER? .......................................................... 1
1.2 MODELLER bibliography ................................................. 2
1.3 Obtaining and installing the program ................................... 3
1.4 Bug reports ................................................................. 4
1.5 Method for comparative protein structure modeling by MODELLER .......... 5
1.6 Using MODELLER for comparative modeling ............................. 8
1.6.1 Preparing input files .................................................. 8
1.6.2 Running MODELLER ................................................... 9

## 2 Automated comparative modeling with 
amodel

2.1 Simple usage ............................................................... 11
2.2 More advanced usage ..................................................... 12
2.2.1 Including water molecules, HETATM residues, and hydrogen atoms ........................................ 12
2.2.2 Changing the default optimization and refinement protocol ............. 13
2.2.3 Getting a very fast and approximate model .......................... 14
2.2.4 Building a model from multiple templates ......................... 15
2.2.5 Building an all hydrogen model .................................... 15
2.2.6 Refining only part of the model .................................... 16
2.2.7 Including disulfide bridges ........................................ 17
2.2.8 Providing your own restraints file ................................ 18
2.2.9 Using your own initial model ...................................... 19
2.2.10 Adding additional restraints to the defaults ....................... 19
2.2.11 Building multi-chain models with symmetry ...................... 20
2.2.12 Accessing output data after modeling is complete ............... 21
2.2.13 Fully automated alignment and modeling ......................... 22
2.3 Loop optimization .......................................................... 23
2.3.1 Automatic loop refinement after model building .................... 23
2.3.2 Defining loop regions for refinement .............................. 24
2.3.3 Refining an existing PDB file ..................................... 25
3 Frequently asked questions and history

3.1 Frequently asked questions (FAQ) and examples .............................................. 27

3.2 Modeller updates .................................................................................................. 34

3.2.1 Changes since release 9v3 ............................................................................... 34

3.2.2 Changes since release 9v2 ............................................................................... 34

4 Comparative modeling class reference .................................................................. 37

4.1 automodel reference ............................................................................................ 37

4.1.1 automodel() — prepare to build one or more comparative models ................. 37

4.1.2 automodel.library.schedule — select optimization schedule ....................... 37

4.1.3 automodel.md_level — control the model refinement level ......................... 38

4.1.4 automodel.outputs — all output data for generated models ....................... 38

4.1.5 automodel.rand_method — control initial model randomization ................. 38

4.1.6 automodel.generate_method — control initial model generation ............... 38

4.1.7 automodel.max_var_iterations — select length of optimizations ................. 38

4.1.8 automodel.repeat_optimization — number of times to repeat optimization .... 38

4.1.9 automodel.max_molpdf — objective function cutoff ..................................... 38

4.1.10 automodel.initial_malign3d — initial template alignment ......................... 38

4.1.11 automodel.starting_model — first model to build ..................................... 39

4.1.12 automodel.ending_model — last model to build ....................................... 39

4.1.13 automodel.final_malign3d — final template-model alignment ................... 39

4.1.14 automodel.write_intermediates — write intermediate files during optimization 39

4.1.15 automodel.trace_output — control optimization output ............................ 39

4.1.16 automodel.get_optimize_actions() — get actions to carry out during the initial optimization 39

4.1.17 automodel.get_refine_actions() — get actions to carry out during the refinement 39

4.1.18 automodel.select_atoms() — select region for optimization ...................... 39

4.1.19 automodel.auto_align() — generate an automatic initial alignment .......... 39

4.1.20 automodel.very_fast() — request rapid optimization ............................... 40

4.1.21 automodel.make() — build all models ....................................................... 40

4.1.22 automodel.cluster() — cluster all built models ....................................... 40

4.1.23 automodel.special_restraints() — add additional restraints ..................... 40

4.1.24 automodel.nonstd_restraints() — add restraints on ligands ....................... 40

4.1.25 automodel.special_patches() — add additional patches to the topology ....... 41

4.1.26 automodel.user_after_single_model() — analyze or refine each model ....... 41

4.1.27 automodel.get_model_filename() — get the model PDB name .................. 41

4.1.28 automodel.use_parallel_job() — parallelize model building .................... 41

4.2 allhmodel reference ............................................................................................. 41

4.2.1 allhmodel() — prepare to build all-hydrogen models .................................. 42

4.3 loopmodel reference ............................................................................................ 42

4.3.1 loopmodel() — prepare to build models with loop refinement .................... 42

4.3.2 loopmodel.loop.md_level — control the loop model refinement level .......... 42

4.3.3 loopmodel.loop.max_var_iterations — select length of optimizations .......... 42

4.3.4 loopmodel.loop.library.schedule — select optimization schedule .......... 42

4.3.5 loopmodel.loop.ending_model — first loop model to build ....................... 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.6</td>
<td><code>energy_data.dynamic_lennard</code> — calculate Lennard-Jones restraints</td>
</tr>
<tr>
<td>6.3.7</td>
<td><code>energy_data.dynamic_coulomb</code> — calculate Coulomb restraints</td>
</tr>
<tr>
<td>6.3.8</td>
<td><code>energy_data.dynamic_modeller</code> — calculate non-bonded spline restraints</td>
</tr>
<tr>
<td>6.3.9</td>
<td><code>energy_data.excl_local</code> — exclude certain local pairs of atoms</td>
</tr>
<tr>
<td>6.3.10</td>
<td><code>energy_data.radii_factor</code> — scale atomic radii</td>
</tr>
<tr>
<td>6.3.11</td>
<td><code>energy_data.lennard_jones_switch</code> — Lennard-Jones switching parameters</td>
</tr>
<tr>
<td>6.3.12</td>
<td><code>energy_data.coulomb_switch</code> — Coulomb switching parameters</td>
</tr>
<tr>
<td>6.3.13</td>
<td><code>energy_data.relative_dielectric</code> — relative dielectric</td>
</tr>
<tr>
<td>6.3.14</td>
<td><code>energy_data.covalent_cys</code> — use disulfide bridges in residue distance</td>
</tr>
<tr>
<td>6.3.15</td>
<td><code>energy_data.nonbonded_sel_atoms</code> — control interaction with picked atoms</td>
</tr>
<tr>
<td>6.3.16</td>
<td><code>energy_data.nlogn_use</code> — select non-bond list generation algorithm</td>
</tr>
<tr>
<td>6.3.17</td>
<td><code>energy_data.energy_terms</code> — user-defined global energy terms</td>
</tr>
<tr>
<td>6.4</td>
<td>The <code>io_data</code> class: coordinate file input parameters</td>
</tr>
<tr>
<td>6.4.1</td>
<td><code>io_data()</code> — create a new input parameters object</td>
</tr>
<tr>
<td>6.4.2</td>
<td><code>io_data.hetatm</code> — whether to read HETATM records</td>
</tr>
<tr>
<td>6.4.3</td>
<td><code>io_data.hydrogen</code> — whether to read hydrogen atoms</td>
</tr>
<tr>
<td>6.4.4</td>
<td><code>io_data.water</code> — whether to read water molecules</td>
</tr>
<tr>
<td>6.4.5</td>
<td><code>io_data.atom_files_directory</code> — search path for coordinate files</td>
</tr>
<tr>
<td>6.5</td>
<td>The <code>Libraries</code> class: stereochemical parameters and molecular topology</td>
</tr>
<tr>
<td>6.5.1</td>
<td><code>Libraries.topology</code> — topology library information</td>
</tr>
<tr>
<td>6.5.2</td>
<td><code>Libraries.parameters</code> — parameter library information</td>
</tr>
<tr>
<td>6.5.3</td>
<td><code>Topology.append()</code> — append residue topology library</td>
</tr>
<tr>
<td>6.5.4</td>
<td><code>Topology.clear()</code> — clear residue topology library</td>
</tr>
<tr>
<td>6.5.5</td>
<td><code>Topology.read()</code> — read residue topology library</td>
</tr>
<tr>
<td>6.5.6</td>
<td><code>Parameters.append()</code> — append parameters library</td>
</tr>
<tr>
<td>6.5.7</td>
<td><code>Parameters.clear()</code> — clear parameters library</td>
</tr>
<tr>
<td>6.5.8</td>
<td><code>Parameters.read()</code> — read parameters library</td>
</tr>
<tr>
<td>6.5.9</td>
<td><code>Topology.make()</code> — make a subset topology library</td>
</tr>
<tr>
<td>6.5.10</td>
<td><code>Topology.submodel</code> — select topology model type</td>
</tr>
<tr>
<td>6.5.11</td>
<td><code>Topology.write()</code> — write residue topology library</td>
</tr>
<tr>
<td>6.6</td>
<td>The <code>model</code> class: handling of atomic coordinates, and model building</td>
</tr>
<tr>
<td>6.6.1</td>
<td><code>model()</code> — create a new 3D model</td>
</tr>
<tr>
<td>6.6.2</td>
<td><code>model.seq_id</code> — sequence identity between the model and templates</td>
</tr>
<tr>
<td>6.6.3</td>
<td><code>model.resolution</code> — resolution of protein structure</td>
</tr>
<tr>
<td>6.6.4</td>
<td><code>model.last_energy</code> — last objective function value</td>
</tr>
<tr>
<td>6.6.5</td>
<td><code>model.remark</code> — text remark(s)</td>
</tr>
<tr>
<td>6.6.6</td>
<td><code>model.restraints</code> — all static restraints which act on the model</td>
</tr>
<tr>
<td>6.6.7</td>
<td><code>model.group_restraints</code> — all restraints which act on atom groups</td>
</tr>
<tr>
<td>6.6.8</td>
<td><code>model.atoms</code> — all atoms in the model</td>
</tr>
<tr>
<td>6.6.9</td>
<td><code>model.point()</code> — return a point in Cartesian space</td>
</tr>
<tr>
<td>6.6.10</td>
<td><code>model.atom_range()</code> — return a subset of all atoms</td>
</tr>
<tr>
<td>6.6.11</td>
<td><code>model.residue_range()</code> — return a subset of all residues</td>
</tr>
<tr>
<td>6.6.12</td>
<td><code>model.get_insertions()</code> — return a list of all insertions</td>
</tr>
</tbody>
</table>
6.6.13 `model.get_deletions()` — return a list of all deletions ........................................... 70
6.6.14 `model.loops()` — return a list of all loops .............................................................. 70
6.6.15 `model.read()` — read coordinates for MODEL ............................................................ 70
6.6.16 `model.build_sequence()` — build model from a sequence of one-letter codes .............. 71
6.6.17 `model.write()` — write MODEL .................................................................................. 72
6.6.18 `model.clear_topology()` — clear model topology ....................................................... 72
6.6.19 `model.generate_topology()` — generate MODEL topology ........................................ 72
6.6.20 `model.write_psf()` — write molecular topology to PSF file ........................................ 73
6.6.21 `model.patch()` — patch MODEL topology ................................................................ 73
6.6.22 `model.patch_ss_templates()` — guess MODEL disulfides from templates .................. 74
6.6.23 `model.patch_ss()` — guess MODEL disulfides from model structure ......................... 75
6.6.24 `model.build()` — build MODEL coordinates from topology ....................................... 76
6.6.25 `model.transfer_xyz()` — copy templates’ coordinates to MODEL ................................ 77
6.6.26 `model.res_num_from()` — residue numbers from MODEL2 to MODEL ......................... 78
6.6.27 `model.rename_segments()` — rename MODEL segments ........................................... 79
6.6.28 `model.to_iupac()` — standardize certain dihedral angles ............................................ 79
6.6.29 `model.reorder_atoms()` — standardize order of MODEL atoms .................................. 80
6.6.30 `model.orient()` — center and orient MODEL ............................................................... 81
6.6.31 `model.write_data()` — write derivative MODEL data .................................................. 81
6.6.32 `model.make_region()` — define a random surface patch of atoms ............................. 82
6.6.33 `model.color()` — color MODEL according to alignment .......................................... 83
6.6.34 `model.make_chains()` — Fetch sequences from PDB file .......................................... 84
6.6.35 `model.saxs_intens()` — Calculate SAXS intensity from model .................................. 85
6.6.36 `model.saxs_pr()` — Calculate \( P(r) \) of model ............................................................... 85
6.6.37 `model.saxs_chifun()` — Calculate SAXS score chi from model .................................. 85
6.6.38 `model.assess_ga341()` — assess a model with the GA341 method ........................... 85
6.7 The Restraints class: static restraints ................................................................................ 87
6.7.1 Restraints.rigid_bodies — all rigid bodies .......................................................... 87
6.7.2 Restraints.pseudo_atoms — all pseudo atoms .............................................................. 87
6.7.3 Restraints.excluded_pairs — all excluded pairs ............................................................. 87
6.7.4 Restraints.symmetry — all symmetry restraints ............................................................ 87
6.7.5 Restraints.symmetry.report() — report violated symmetry restraints ....................... 87
6.7.6 Restraints.make() — make restraints ............................................................................. 87
6.7.7 Restraints.make_distance() — make distance restraints .............................................. 90
6.7.8 Restraints.unpick_all() — unselect all restraints ............................................................ 91
6.7.9 Restraints.clear() — delete all restraints ....................................................................... 91
6.7.10 Restraints.pick() — pick restraints for selected atoms .................................................. 91
6.7.11 Restraints.unpick_redundant() — unselect redundant restraints ............................... 92
6.7.12 Restraints.remove_unpicked() — remove unselected restraints .................................... 92
6.7.13 Restraints.remove_unselected() — remove unselected or redundant restraints .......... 93
6.7.14 Restraints.add() — add restraint .................................................................................. 93
6.7.15 Restraints.unpick() — unselect restraints ................................................................... 94
6.7.16 Restraints.reindex() — renumber model restraints using another model ............... 94
6.7.17 Restraints.spline() — approximate restraints by splines ........................................... 95
6.7.18 Restraints.append() — read spatial restraints .......................................................... 96
6.7.19 Restraints.write() — write spatial restraints ............................................................. 96

6.8 The secondary_structure module: secondary structure restraints ..................................... 97
6.8.1 alpha() — make an α-helix .............................................................................................. 97
6.8.2 strand() — make a β-strand ............................................................................................ 98
6.8.3 sheet() — make a β-sheet ............................................................................................... 98

6.9 The selection class: handling of sets of atom coordinates .................................................. 99
6.9.1 selection() — create a new selection ............................................................................... 99
6.9.2 selection.add() — add objects to selection ..................................................................... 101
6.9.3 selection.extend_by_residue() — extend selection by residue ..................................... 102
6.9.4 selection.by_residue() — make sure all residues are fully selected ............................... 102
6.9.5 selection.select_sphere() — select all atoms within radius ........................................ 102
6.9.6 selection.only_mainchain() — select only mainchain atoms ........................................ 102
6.9.7 selection.only_sidechain() — select only sidechain atoms .......................................... 102
6.9.8 selection.only_atom_types() — select only atoms of given types .................................. 102
6.9.9 selection.only_residue_types() — select only atoms of given residue type ................ 103
6.9.10 selection.only_std_residues() — select only standard residues .................................... 103
6.9.11 selection.only_no_topology() — select only residues without topology ...................... 103
6.9.12 selection.only_het_residues() — select only HETATM residues .................................. 103
6.9.13 selection.only_water_residues() — select only water residues .................................... 103
6.9.14 selection.only_defined() — select only atoms with defined coordinates ...................... 103
6.9.15 selection.write() — write selection coordinates to a file ...................................... 103
6.9.16 selection.translate() — translate all coordinates ....................................................... 104
6.9.17 selection.rotate_origin() — rotate coordinates about origin .................................... 104
6.9.18 selection.rotate_mass_center() — rotate coordinates about mass center ................. 104
6.9.19 selection.transform() — transform coordinates with a matrix .................................... 104
6.9.20 selection.mutate() — mutate selected residues ........................................................ 105
6.9.21 selection.randomize_xyz() — randomize selected coordinates ..................................... 106
6.9.22 selection.superpose() — superpose model on selection given alignment .................. 107
6.9.23 selection.rotate_dihedrals() — change dihedral angles ............................................. 107
6.9.24 selection.unbuild() — undefine coordinates ............................................................... 110
6.9.25 selection.hot_atoms() — atoms violating restraints ................................................. 111
6.9.26 selection.energy() — evaluate atom selection given restraints .................................. 112
6.9.27 selection.debug_function() — test code self-consistency .......................................... 114
6.9.28 selection.assess_dope() — assess a model selection with the DOPE method ............ 115
6.9.29 selection.assess_dopehr() — assess a model with the DOPE-HR method .................. 116

6.10 The physical module: contributions to the objective function ........................................ 117
6.10.1 physical.values() — create a new set of physical values ......................................... 117

6.11 The optimizers module: optimization of the model ......................................................... 119
6.11.1 conjugate_gradients() — optimize atoms given restraints, with CG ......................... 119
6.11.2 quasi_newton() — optimize atoms with quasi-Newton minimization ....................... 121
6.11.3 molecular_dynamics() — optimize atoms given restraints, with MD ....................... 121
CONTENTS

6.11.4 actions.write_structure() — write out the model coordinates ........................................... 121
6.11.5 actions.trace() — write out optimization energies, etc .......................................................... 122
6.11.6 actions.charmm_trajectory() — write out a CHARMM trajectory ........................................... 122
6.11.7 User-defined optimizers ........................................................................................................... 123

6.12 The schedule class: variable target function optimization ........................................................... 126
6.12.1 schedule() — create a new schedule ....................................................................................... 126
6.12.2 schedule.make_for_model() — trim a schedule for a model ...................................................... 127
6.12.3 schedule.write() — write optimization schedule ..................................................................... 128

6.13 The group_restraints class: restraints on atom groups ................................................................. 129
6.13.1 group_restraints() — create a new set of group restraints ....................................................... 129
6.13.2 group_restraints.append() — read group restraint parameters .............................................. 129

6.14 The gbsa module: implicit solvation ............................................................................................... 131
6.14.1 gbsa.Scorer() — create a new scorer to evaluate GB/SA energies .............................................. 131

6.15 The alignment class: comparison of sequences and structures ..................................................... 132
6.15.1 alignment() — create a new alignment ..................................................................................... 132
6.15.2 alignment.comments — alignment file comments ....................................................................... 132
6.15.3 alignment.append() — read sequences and/or their alignment ................................................ 132
6.15.4 alignment.clear() — delete all sequences from the alignment .................................................. 133
6.15.5 alignment.read_one() — read sequences one by one from a file .............................................. 133
6.15.6 alignment.check_structure_structure() — check template structure superpositions ............. 134
6.15.7 alignment.check_sequence_structure() — check sequence/structure alignment for sanity ........ 135
6.15.8 alignment.check() — check alignment for modeling ................................................................... 135
6.15.9 alignment.compare_with() — compare two alignments .............................................................. 135
6.15.10 alignment.append_model() — copy model sequence and coordinates to alignment ............... 136
6.15.11 alignment.append_sequence() — add a sequence from one-letter codes .................................. 137
6.15.12 alignment.append_profile() — add profile sequences to the alignment ................................. 137
6.15.13 alignment.write() — write sequences and/or their alignment .................................................. 137
6.15.14 alignment.edit() — edit overhangs in alignment ..................................................................... 138
6.15.15 alignment.describe() — describe proteins ................................................................................ 139
6.15.16 alignment.id_table() — calculate percentage sequence identities .......................................... 140
6.15.17 alignment.compare_sequences() — compare sequences in alignment .................................... 140
6.15.18 alignment.align() — align two (blocks of) sequences ............................................................... 141
6.15.19 alignment.align2d() — align sequences with structures ............................................................ 142
6.15.20 alignment.malign() — align two or more sequences ................................................................. 144
6.15.21 alignment.consensus() — consensus sequence alignment ...................................................... 145
6.15.22 alignment.compare_structures() — compare 3D structures given alignment .......................... 145
6.15.23 alignment.align3d() — align two structures .............................................................................. 146
6.15.24 alignment.malign3d() — align two or more structures ............................................................... 148
6.15.25 alignment.salign() — align two or more sequences/structures of proteins .............................. 149
6.15.26 alignment.to_profile() — convert alignment to profile format ............................................... 158
6.15.27 alignment.segment_matching() — align segments ................................................................ 158

6.16 The Sequence class: a single sequence within an alignment ......................................................... 161
6.16.1 Sequence.range — residue range ............................................................................................... 161
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.16.2</td>
<td>Sequence.code — alignment code</td>
</tr>
<tr>
<td>6.16.3</td>
<td>Sequence.atom_file — PDB file name</td>
</tr>
<tr>
<td>6.16.4</td>
<td>Sequence.source — source organism</td>
</tr>
<tr>
<td>6.16.5</td>
<td>Sequence.name — protein name</td>
</tr>
<tr>
<td>6.16.6</td>
<td>Sequence.prottyp — protein sequence type</td>
</tr>
<tr>
<td>6.16.7</td>
<td>Sequence.resolution — structure resolution</td>
</tr>
<tr>
<td>6.16.8</td>
<td>Sequence.rfactor — R factor</td>
</tr>
<tr>
<td>6.16.9</td>
<td>Sequence.residues — list of all residues in the sequence</td>
</tr>
<tr>
<td>6.16.10</td>
<td>Sequence.chains — list of all chains in the sequence</td>
</tr>
<tr>
<td>6.16.11</td>
<td>Sequence.transfer_res_prop() — transfer residue properties</td>
</tr>
<tr>
<td>6.16.12</td>
<td>Sequence.get_num_equiv() — get number of equivalences</td>
</tr>
<tr>
<td>6.16.13</td>
<td>Sequence.get_sequence_identity() — get sequence identity</td>
</tr>
<tr>
<td>6.17</td>
<td>The Structure class: a template structure within an alignment</td>
</tr>
<tr>
<td>6.17.1</td>
<td>Structure.write() — write out PDB file</td>
</tr>
<tr>
<td>6.17.2</td>
<td>Structure.reread() — reread coordinates from atom file</td>
</tr>
<tr>
<td>6.18</td>
<td>The Chain class: a single chain in a model or alignment</td>
</tr>
<tr>
<td>6.18.1</td>
<td>Chain.name — chain ID</td>
</tr>
<tr>
<td>6.18.2</td>
<td>Chain.residues — all residues in the chain</td>
</tr>
<tr>
<td>6.18.3</td>
<td>Chain.atoms — all atoms in the chain</td>
</tr>
<tr>
<td>6.18.4</td>
<td>Chain.filter() — check if this chain passes all criteria</td>
</tr>
<tr>
<td>6.18.5</td>
<td>Chain.write() — write out chain sequence to an alignment file</td>
</tr>
<tr>
<td>6.18.6</td>
<td>Chain.atom_file_and_code() — get suitable names for this chain</td>
</tr>
<tr>
<td>6.19</td>
<td>The Residue class: a single residue in a model or alignment</td>
</tr>
<tr>
<td>6.19.1</td>
<td>Residue.name — internal (CHARMM) residue type name</td>
</tr>
<tr>
<td>6.19.2</td>
<td>Residue.pdb_name — PDB (IUPAC) type name</td>
</tr>
<tr>
<td>6.19.3</td>
<td>Residue.code — One-letter residue type code</td>
</tr>
<tr>
<td>6.19.4</td>
<td>Residue.hetatm — HETATM indicator</td>
</tr>
<tr>
<td>6.19.5</td>
<td>Residue.index — internal integer index</td>
</tr>
<tr>
<td>6.19.6</td>
<td>Residue.atoms — all atoms in the residue</td>
</tr>
<tr>
<td>6.19.7</td>
<td>Residue.chain — chain object</td>
</tr>
<tr>
<td>6.19.8</td>
<td>Residue.phi — ( \phi ) dihedral angle</td>
</tr>
<tr>
<td>6.19.9</td>
<td>Residue.psi — ( \psi ) dihedral angle</td>
</tr>
<tr>
<td>6.19.10</td>
<td>Residue.omega — ( \omega ) dihedral angle</td>
</tr>
<tr>
<td>6.19.11</td>
<td>Residue.alpha — ( \alpha ) dihedral angle</td>
</tr>
<tr>
<td>6.19.12</td>
<td>Residue.chi1 — ( \chi_1 ) dihedral angle</td>
</tr>
<tr>
<td>6.19.13</td>
<td>Residue.chi2 — ( \chi_2 ) dihedral angle</td>
</tr>
<tr>
<td>6.19.14</td>
<td>Residue.chi3 — ( \chi_3 ) dihedral angle</td>
</tr>
<tr>
<td>6.19.15</td>
<td>Residue.chi4 — ( \chi_4 ) dihedral angle</td>
</tr>
<tr>
<td>6.19.16</td>
<td>Residue.chi5 — ( \chi_5 ) dihedral angle</td>
</tr>
<tr>
<td>6.19.17</td>
<td>Residue.get_aligned_residue() — get aligned residue in another sequence</td>
</tr>
<tr>
<td>6.19.18</td>
<td>Residue.addLeading_gaps() — add gap(s) before this residue</td>
</tr>
<tr>
<td>6.19.19</td>
<td>Residue.addTrailing_gaps() — add gap(s) after this residue</td>
</tr>
<tr>
<td>6.19.20</td>
<td>Residue.removeLeading_gaps() — remove gap(s) before this residue</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.26.4 saxsdata.saxs_pr_read() — Read in P(r) data</td>
<td>190</td>
</tr>
<tr>
<td>6.27 The info object: obtaining information about the MODELLER build</td>
<td></td>
</tr>
<tr>
<td>6.27.1 info.version — the full MODELLER version number</td>
<td>192</td>
</tr>
<tr>
<td>6.27.2 info.version_info — the version number, as a tuple</td>
<td>192</td>
</tr>
<tr>
<td>6.27.3 info.build_date — the date this binary was built</td>
<td>192</td>
</tr>
<tr>
<td>6.27.4 info.exe_type — the executable type of this binary</td>
<td>192</td>
</tr>
<tr>
<td>6.27.5 info.debug — this binary's debug flag</td>
<td>192</td>
</tr>
<tr>
<td>6.27.6 info.bindir — MODELLER binary directory</td>
<td>192</td>
</tr>
<tr>
<td>6.27.7 info.time_mark() — print current date, time, and CPU time</td>
<td>192</td>
</tr>
<tr>
<td>6.27.8 info.jobname — name of the current job</td>
<td>192</td>
</tr>
<tr>
<td>6.28 The log object: controlling the amount of output</td>
<td></td>
</tr>
<tr>
<td>6.28.1 log.level() — Set all log output levels</td>
<td>193</td>
</tr>
<tr>
<td>6.28.2 log.none() — display no log output</td>
<td>193</td>
</tr>
<tr>
<td>6.28.3 log.minimal() — display minimal log output</td>
<td>193</td>
</tr>
<tr>
<td>6.28.4 log.verbose() — display verbose log output</td>
<td>193</td>
</tr>
<tr>
<td>6.28.5 log.very_verbose() — display verbose log output, and dynamic memory information</td>
<td>193</td>
</tr>
<tr>
<td>6.29 The modfile module: handling of files</td>
<td></td>
</tr>
<tr>
<td>6.29.1 modfile.default() — generate an 'automatic' filename</td>
<td>194</td>
</tr>
<tr>
<td>6.29.2 modfile.delete() — delete a file</td>
<td>194</td>
</tr>
<tr>
<td>6.29.3 modfile.inquire() — check if file exists</td>
<td>194</td>
</tr>
<tr>
<td>6.29.4 modfile.File() — open a handle to a MODELLER file</td>
<td>194</td>
</tr>
<tr>
<td>6.30 The scripts module: utility scripts</td>
<td></td>
</tr>
<tr>
<td>6.30.1 cispeptide() — creates cis-peptide stereochemical restraints</td>
<td>195</td>
</tr>
<tr>
<td>6.30.2 complete_pdb() — read a PDB file, and fill in any missing residues</td>
<td>195</td>
</tr>
<tr>
<td>6.31 Parallel job support</td>
<td></td>
</tr>
<tr>
<td>6.31.1 job() — create a new parallel job</td>
<td>195</td>
</tr>
<tr>
<td>6.31.2 sge_pe_job() — create a job using all Sun GridEngine (SGE) slave processes</td>
<td>196</td>
</tr>
<tr>
<td>6.31.3 sge_qsub_job() — create a job which can be expanded with Sun GridEngine 'qsub'</td>
<td>196</td>
</tr>
<tr>
<td>6.31.4 job.queue_task() — submit a task to run within the job</td>
<td>196</td>
</tr>
<tr>
<td>6.31.5 job.run_all_tasks() — run all queued tasks, and return results</td>
<td>197</td>
</tr>
<tr>
<td>6.31.6 job.start() — start all slaves for message-passing</td>
<td>198</td>
</tr>
<tr>
<td>6.31.7 Communicator.send_data() — send data</td>
<td>198</td>
</tr>
<tr>
<td>6.31.8 Communicator.get_data() — get data</td>
<td>199</td>
</tr>
<tr>
<td>6.31.9 slave.run_cmd() — run a command on the slave</td>
<td>199</td>
</tr>
<tr>
<td>6.31.10 local_slave() — create a slave running on the local machine</td>
<td>199</td>
</tr>
<tr>
<td>6.31.11 sge_pe_slave() — create a slave running on a Sun GridEngine parallel environment slave node</td>
<td>199</td>
</tr>
<tr>
<td>6.31.12 sge_qsub_slave() — create a 'qsub' slave running on a Sun GridEngine node</td>
<td>200</td>
</tr>
<tr>
<td>6.31.13 ssh_slave() — create a slave on a remote host accessed via ssh</td>
<td>200</td>
</tr>
<tr>
<td>7 MODELLER low-level programming</td>
<td></td>
</tr>
<tr>
<td>7.1 User-defined features and restraint forms</td>
<td></td>
</tr>
<tr>
<td>7.1.1 User-defined feature types</td>
<td>201</td>
</tr>
<tr>
<td>7.1.2 User-defined restraint forms</td>
<td>202</td>
</tr>
<tr>
<td>7.1.3 User-defined energy terms</td>
<td>204</td>
</tr>
</tbody>
</table>
## CONTENTS

### 7.2 MODELLER programming interface (API) ................................................................. 205

### A Methods ....................................................................................................................... 209

A.1 Dynamic programming for sequence and structure comparison and searching .................................................. 209

A.1.1 Pairwise comparison ........................................................................................................... 209

A.1.2 Variable gap penalty ......................................................................................................... 210

A.1.3 Local versus global alignment ............................................................................................ 210

A.1.4 Similarity versus distance scores ....................................................................................... 211

A.1.5 Multiple comparisons ....................................................................................................... 211

A.2 Optimization of the objective function by MODELLER ................................................................................. 211

A.2.1 Function ............................................................................................................................ 211

A.2.2 Optimizers ......................................................................................................................... 212

A.3 Equations used in the derivation of the molecular pdf .......................................................................................... 215

A.3.1 Features and their derivatives ............................................................................................ 215

A.3.2 Restraints and their derivatives ......................................................................................... 218

A.4 Flowchart of comparative modeling by MODELLER ...................................................................................... 222

A.5 Loop modeling method ........................................................................................................... 223

### B File formats .................................................................................................................... 225

B.1 Alignment file (PIR) ............................................................................................................ 225

B.2 Restraints file ...................................................................................................................... 227

B.2.1 Restraints ........................................................................................................................ 227

B.2.2 Excluded pairs .................................................................................................................. 228

B.2.3 Pseudo atoms .................................................................................................................. 228

B.2.4 Symmetry restraints ....................................................................................................... 229

B.2.5 Rigid bodies .................................................................................................................... 229

B.3 Profile file ........................................................................................................................ 230

B.4 Binary files ........................................................................................................................ 231

### C Converting Top scripts from old MODELLER versions .............................................................. 233

C.1 Running old scripts unchanged ............................................................................................. 233

C.2 Converting Top scripts to Python .......................................................................................... 233

C.2.1 Top commands and variables ......................................................................................... 233

C.2.2 Top models and alignments .............................................................................................. 234

C.2.3 Top to Python correspondence ....................................................................................... 234
List of Figures

1.1 *Comparative protein modeling by satisfaction of spatial restraints* .......................... 6
1.2 *Sample spatial restraint* ......................................................................................... 7
1.3 *Optimization of the objective function* ............................................................... 7

5.1 *Mathematical forms of restraints* ........................................................................... 56
List of Tables

5.1 List of file types .................................................. 55
6.1 List of “physical” restraint types .................................. 118
B.1 Numerical restraint forms ......................................... 228
B.2 Numerical feature types ........................................... 228
B.3 Numerical pseudo atom types ..................................... 229
C.1 Correspondence between TOP and Python commands ...... 237
C.2 Correspondence between TOP and Python variables ......... 237
Copyright notice

MODELLER, a protein structure modeling program.
Copyright © 1989–2008 Andrej Šali.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for any purpose. The entire risk as to the quality and performance of the program is with the user.
Distribution of the program is allowed only with the author’s written consent.

Python 2.3 is incorporated into MODELLER. This is Copyright © 2001–2003 Python Software Foundation; All Rights Reserved.

gzip is included in Windows builds, which is

    Copyright © 1999, 2001 Free Software Foundation, Inc.
    Copyright © 1992, 1993 Jean-loup Gailly
    gzip is free software; you can redistribute it and/or modify it under the terms of the GNU General
    Public License as published by the Free Software Foundation; either version 2, or (at your option)
    any later version.
    gzip is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without
    even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    See the GNU General Public License for more details.

Under the terms of the GPL, the source code for gzip can be provided on request by the MODELLER developers. Alternatively, it can be downloaded from http://gnuwin32.sourceforge.net/.

bzip2 is also included in Windows builds, which is

    Copyright (C) 1996-2002 by Julian Seward.
    This program is free software; you can redistribute it and/or modify it under the terms set out in
    the LICENSE file, which is included in the bzip2-1.0 source distribution.
    This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY;
    without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
    See the LICENSE file for more details.

The source code for bzip2 can be downloaded from http://gnuwin32.sourceforge.net/.

glib (and, on relevant platforms, its libiconv and libintl dependencies) is also included in MODELLER. These libraries are copyrighted under the GNU Library General Public License.
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.

**HDF5** is included in **MODELLER**, with the conditions below.

COPYRIGHT NOTICE AND STATEMENT FOR NCSA HIERARCHICAL DATA FORMAT (HDF) SOFTWARE LIBRARY AND UTILITIES


Contributors: National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC), Lawrence Livermore National Laboratory (LLNL), Sandia National Laboratories (SNL), Los Alamos National Laboratory (LANL), Jean-loup Gailly and Mark Adler (gzip library).

Redistribution and use in source and binary forms, with or without modification, are permitted for any purpose (including commercial purposes) provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or materials provided with the distribution.
3. In addition, redistributions of modified forms of the source or binary code must carry prominent notices stating that the original code was changed and the date of the change.
4. All publications or advertising materials mentioning features or use of this software are asked, but not required, to acknowledge that it was developed by the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign and to credit the contributors.
5. Neither the name of the University nor the names of the Contributors may be used to endorse or promote products derived from this software without specific prior written permission from the University or the Contributors, as appropriate for the name(s) to be used.
6. THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY AND THE CONTRIBUTORS “AS IS” WITH NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. In no event shall the University or the Contributors be liable for any damages suffered by the users arising out of the use of this software, even if advised of the possibility of such damage.

Portions of HDF5 were developed with support from the University of California, Lawrence Livermore National Laboratory (UC LLNL). The following statement applies to those portions of the product and must be retained in any redistribution of source code, binaries, documentation, and/or accompanying materials:

This work was partially produced at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy (DOE) and The Regents of the University of California (University) for the operation of UC LLNL.

DISCLAIMER: This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.
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.

            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.

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


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 [http://salilab.org/publications/](http://salilab.org/publications/). Here is a subset of these publications:


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 http://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 Cα – Cα 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 Cα–Cα 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 & Go, 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).
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 $C_{\alpha}-C_{\alpha}$ 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 $C_\alpha-C_\alpha$ 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 [Sali & 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.
CHAPTER 1. INTRODUCTION

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 [http://salilab.org/modeller/documentation.html](http://salilab.org/modeller/documentation.html). A number of additional tools useful in comparative modeling are listed at [http://salilab.org/bioinformatics_resources.shtml](http://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](http://salilab.org/bioinformatics_resources.shtml), a web server for automated comparative protein structure modeling; and [ModLoop](http://salilab.org/bioinformatics_resources.shtml), 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:

> P1;5fd1
AFVVTDC8CKYTYDCVEVCPVDCCFEGPNFLVHPDEICDCAEPECPAIFSEDEVPEDMQEFIQLNAELA
EVWNPTEKPDPLPAEDWGVGKQLQHLER*

> P1;1fdx
sequence:1fdx:1 :54 :ferredoxin:Peptococcus aerogenes: 2.00: -1.00
AYVINDSC--IACGACKPECPVNIQGS--IYAIADSCIDCGSCASVCPGAPNPED---------------*

See Section 3.1 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 [Sali & 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
1.6. USING MODELLER FOR COMPARATIVE MODELING

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 [http://www.python.org/doc/2.3.5/tut/](http://www.python.org/doc/2.3.5/tut/).

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

```
# Homology 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 homology 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:
   
   `mod9v4 model-default.py`

   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.
(Note: if you have Python installed on your system, you may be able to use MODELLER as an ordinary Python module. This works automatically on Mac OS X, Windows with Python 2.3, 2.4 or 2.5, or using the RPM install on Linux with Python 2.3, 2.4 or 2.5. 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. It also allows 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 file itself, modlib/modeller/automodel/automodel.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

```python
# Homology 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.ending_model = 1 # index of the last model
a.make() # do the actual homology modeling
```

11
Example: examples/automodel/alignment.ali

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

>P1;5fd1
AVFVTNCIKCXYTCVCPVDCYFEGEPNFVHPDECIDCALCEPECAIFSEDEVPEMDQEFIQLNAELA
EVWNITEKKDLPDAEDWDGKVGLQHRER*

>P1;1fdx
sequence:1fdx:1 : :54 : :ferredoxin:Peptococcus aerogenes: 2.00:-1.00
AYVINDSC--IACGACKPECVPNIQGS--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 6.28) 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 io.data.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 HETATM residue) then MODELLER can transfer this into your generated model. This is done first by setting env.io.hetatm to True, which instructs MODELLER to read HETATM records from your template PDB files, and then by using the BLK (‘.’) residue type in your alignment (both in the template(s) and the model sequence) to copy the ligand(s) as a rigid body into the model.

Example: examples/automodel/model-ligand.py

```python
# Homology 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

def __init__(self, *args, **kargs):
    self.HiddenSuper = True
    self.automodel.starting_model = 1
    self.automodel.ending_model = 1
    self.automodel.make()

# 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,
```
2.2. MORE ADVANCED USAGE

```python
alnfile = 'align-ligand.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 = 4 # index of the last model
# (determines how many models to calculate)
a.make() # do the actual homology modeling
```

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. 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 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 HETATM 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-HETATM, inter-HETATM, and HETATM-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 http://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.

### 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) [Sali & 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` 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`).

---

1If 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.
Example: `examples/automodel/model-changeopt.py`

```python
# 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`

```python
# Very fast homology 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 = automodel(env,
              alnfile='alignment.ali',  # alignment filename
              knowns='5fd1',  # codes of the templates
              )
```
2.2. MORE ADVANCED USAGE

```python
sequence='1fdx',  # code of the target
assess_methods=assess.GA341)  # request GA341 model assessment

a.very_fast()  # prepare for extremely fast optimization

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

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

2.2.4 Building a model from multiple templates

It is straightforward 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`

```python
# Homology 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 homology modeling
```

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`

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

log.verbose()
env = environ()
```
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 a chain with no ID. See Section 6.16.9 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. No chain ID is assigned when building a single chain model, but if you build a multi-chain model, chain IDs A, B, C, etc are assigned. 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 `energy.data.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`

```python
# Homology 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 (PDB numbering)
        return selection(self.residue_range('1:', '2:'))
```
# The same thing from chain A (required for multi-chain models):
# return selection(self.residue_range('1:A', '2:A'))

# Residues 4, 6, 10:
# return selection(self.residues['4'], self.residues['6'],
# self.residues['10'])

# All residues except 1-5:
# return selection(self) - selection(self.residue_range('1', '5'))

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

# 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 homology 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 for you (by using the model.patch_ss_templates() command).

Explicit manual restraints can be added by the model.patch_ss_templates() 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 $\chi_3$ 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

# Homology 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

\[2\] 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 $\chi_3$ dihedral angle restraints will depend on the conformation of the equivalent disulfides in the template structure, as described in Sali & Overington, 1994.
# (this routine is empty by default):
class MyModel(automodel):
    def special_patches(self, aln):
        # A disulfide between residues 8 and 45:
        self.patch(residue_type='DISU', residues=(self.residues['8'],
                                                  self.residues['45']))

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
            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 homology modeling

2.2.8 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',
              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 homology modeling
2.2.9 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.)

Example: `examples/automodel/model-myini.py`

```python
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 homology modeling
```

2.2.10 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 2.2.11 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`

```python
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):
        r1 = restraint.distance_ca(100) # distance restraint
        r2 = restraint.defined_sequence(50) # defined secondary structure
        r3 = restraint.read_restraints_file(|'. restrain-file.txt')
```
def special_restraints(self, aln):
    rsr = self.restraints
    at = self.atoms
    # Add some restraints from a file:
    # rsr.append(file='my_rssr1.rssr')
    # Residues 20 through 30 should be an alpha helix:
    rsr.add(secondary_structure.alpha(self.residue_range('20:', '30:')))
    # Two beta-strands:
    rsr.add(secondary_structure.strand(self.residue_range('1:', '6:')))
    rsr.add(secondary_structure.strand(self.residue_range('9:', '14:')))
    # An anti-parallel sheet composed of the two strands:
    rsr.add(secondary_structure.sheet(at['N:1'], at['O:14'],
                                     sheet_h_bonds=-5))
    # Use the following instead for a *parallel* sheet:
    # rsr.add(secondary_structure.sheet(at['N:1'], at['O:9'],
                                       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(form.gaussian(group=physical.xy_distance,
                          feature=features.distance(at['CA:35'],
                                                      at['CA:40']),
                          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 homology modeling

2.2.11 Building multi-chain models with symmetry

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.)

(Note that when building models containing multiple chains, you will need to specify the chain ID whenever
you refer to a residue. See Section 6.16.9 By default, your chains will be labeled A, B, etc but you can change this
by relabeling the chains by calling model.rename_segments() from within the automodel.special_patches() method.)

In the example below, the two chains are also constrained to have similar conformations by use of symmetry
restraints. Just as for the example in Section 2.2.10 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.

Example: examples/automodel/model-multichain.py
# Homology 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 homology modeling

2.2.12 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

```
Example: examples/automodel/model-outputs.py
```

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

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 = filter(lambda x: x['failure'] is None, a.outputs)

# Rank the models by DOPE score
key = 'DOPE score'
ok_models.sort(lambda a,b: cmp(a[key], b[key]))

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

2.2.13 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 6.15 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

```python
# A sample script for fully automated comparative modeling
from modeller import *
from modeller.automodel import *

# Load the automodel class
```
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.

Example: examples/automodel/model-loop.py

```python
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 homology modeling
```

Example: examples/automodel/alignment.seg

```
>P1;1fdx
sequence:::::ferredoxin:Peptococcus aerogenes:-1.00:-1.00
AYVINDSCIACGACKPECPVNIQGSIYIADSCIDCGASCSPGAPNED*

>P1;1fdn
structureX:1fdn:FIRST:@:55:@:ferredoxin:Clostrodium aciduriici: 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
*
```
CHAPTER 2. AUTOMATED COMPARATIVE MODELING WITH AUTOMODEL

# Homology 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
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 homology 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:', '28:'),

        # do homology modeling
2.3. LOOP OPTIMIZATION

```python
self.residue_range('45:', '50:'))

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`

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

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 from residue 19 to 28 inclusive
        return selection(self.residue_range('19:', '28:'))

m = MyLoop(env,
    inimodel='1fdx.B99990001.pdb',  # initial model of the target
    sequence='1fdx')                 # code of the target

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 archive of the Users Mail List at [http://salilab.org/archives/modeller_usage/](http://salilab.org/archives/modeller_usage/)

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 modeling routine. The alignment should be as follows when the chimera is a combination of proteins A and B:

   ```
   proteinA  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
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 \( \omega \) conformation.

MODELLER should usually be allowed to handle this automatically \textit{via} the omega dihedral angle restraints, which are calculated by default.

```python
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 \texttt{Restraints.append()} calculated by \texttt{Restraints.make()} or \texttt{Restraints.make_distance()} or added “manually” by \texttt{Restraints.add()} \texttt{Restraints.pick()} picks those restraints for objective function calculation that restrain the selected atoms only. The \texttt{automodel.homcsr()} routine contains examples of selecting atoms when generating restraints by \texttt{Restraints.make_distance()} There are also commands for adding and unselecting single restraints, \texttt{Restraints.add()} and \texttt{Restraints.unpick()} respectively. If you do \texttt{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 \ref{sec:optimization}.

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 \ref{sec:hydrogen} and \ref{sec:hetatm} for some examples.

```python
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 \ref{sec:water} 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, CHARMX and X-PLOR provide a reasonably straightforward solution via the residue topology and parameter libraries. MODELLER uses CHARMX 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 CHARMX 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 0 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 CHARMX 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 CHARMX residue types). The number gives the total residue charge.

The ATOM records specify the IUPAC (i.e., PDB) atom names and the CHARMX 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 a b c d d_ab \alpha_{abc} \Theta_{abcd} \alpha_{bcd} d_{cd}
```

specifies distances `d`, angles `\alpha`, and either dihedral angles or improper dihedral angles `\Theta` 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 libraries, `modlib/radii.lib` and `modlib/radii14.lib`. These libraries list the atomic radii for the different topology models, for the long range and 1–4 non-bonded soft-sphere terms, respectively. The full names of the files that are used during calculation are given by the environment variables \$\text{RADII}_\text{LIB}\$ and \$\text{RADII14}_\text{LIB}\$.

(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 \$\text{MODELS}_\text{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 \texttt{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 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 C C C C C C C C C C C C C C C C
ATOM 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

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 \$\text{RESTYP}_\text{LIB}\$. See the comments in the file for further information.

Every residue in the Charmm topology file has to have an entry in the \$\text{RESTYP}_\text{LIB}\$ library, but not every residue entry in the \$\text{RESTYP}_\text{LIB}\$ library needs an entry in the residue topology file. If you need to edit the \$\text{RESTYP}_\text{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 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.83). 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: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 2.2.10 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_α 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 build 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 the standard comparative modeling routine.

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 Accelrys, at [http://www.accelrys.com/products/modeler/index.html](http://www.accelrys.com/products/modeler/index.html).

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:

```python
from modeller import *
env = environ()
code = ‘1BY8’
mdl = model(env, file=code)
alm = alignment(env)
alm.append_model(mdl, align_codes=code)
alm.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_MODELLER9v4` environment variable, as is done by our own `MODWEB` and `MODLOOP` interfaces.

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.
3.2 ModeLLer updates

3.2.1 Changes since release 9v3

- The speed of GB/SA calculations has been dramatically increased; the cutoff distance used in the GB/SA calculation can also now be adjusted to balance speed against accuracy (the default cutoff has been decreased from 15.0Å to 8.0Å).

  - `alignment.append()` no longer takes `rewind_file` and `close_file` arguments; they did not work correctly anyway. New method `alignment.read_one()` added to read sequences one by one from a PIR or FASTA file.
  - `alignment.write()`, `model.write()`, `selection.write()`, `Chain.write()` and `Structure.write()` now take either a file name or a file handle (`modfile.File()`).
  - Split `alignment.check()` into `alignment.check_structure_structure()` and `alignment.check_sequence_structure()`. Both functions now return the number of pairs which exceeded the cutoffs, which are also adjustable.
  - `alignment.compare_with()` now returns the percentage of equivalent residue-residue pairs in the two alignments.
  - `io.data.atom_files_directory` is now a Python list of directories, rather than a colon-separated string. (This is so that everything works on Windows, where ‘:’ is a valid character in directory names.) Old scripts that use colon-separated strings should still work, but will trigger a warning.

- When reading or writing compressed files, the uncompressed versions of these files are now written into writeable local storage (`${MODELLER_TMPDIR}` if set, otherwise the system temporary directory if possible, otherwise the current working directory). Previously, these files were written into the same directory as the compressed versions, which is problematic when that directory is read-only or is on network storage.

- Files compressed with p7zip can now be read, provided that the 7za binary is available on your system. Such files should contain only a single member (e.g., a file foo.pdb could be compressed into the archive foo.pdb.7z).

- Update PDB files used in the examples (remediated PDBs are now used).

- Added `selection.only_water_residues()` method, which selects only atoms in water residues.

- User-defined energy terms now take a cutoff parameter to use the non-bonded list.

- Bugfix: use the basename, not the full path name, to generate output file names in `density.grid_search()`.

- Bugfix: `model.transfer_xyz()` with `cluster_cut` ≤ 0.0 incorrectly calculated the number of atoms in each residue, which in some circumstances could cause a crash.

- Bugfix: miscellaneous `alignment.salign()` fixes.

- Bugfix: `alignment.append_model()` now copies the model’s resolution, R-factor, name and source to the alignment.

- Bugfix: if two or more different types for a single residue are listed in a PDB with alternate location codes (e.g. 80:A in 1en2), only the first is read by ModeLLer.

3.2.2 Changes since release 9v2

- Mac OS X 10.5 (Leopard) is now supported (with Python 2.5).

- Added `Structure.write()` method, to write out current template structures as PDB files.

- Added `Chain.write()`, `Chain.filter()` and `Chain.atom_file_and_code()` methods, to write out sequences of individual chains. These obsolete the `model.make_chains()` method.

- Added `selection.only_defined()` method, which selects only atoms with defined coordinates.
The GB/SA scorer is now implemented as a user-defined energy term; `energy_data.dynamic.gbsa` has thus been removed.

User-defined energy terms in C now get passed additional libraries and energy_data parameters.

Invalid headers in PIR alignment files now result in an error, to avoid potential problems with these headers going unnoticed.

File format errors now have their own Python exception - FileFormatError - which is a subclass of the generic ModellerError.

- `alignment.align()` and `alignment.consensus()` no longer take a `max_gap_length` parameter, as it is not used in any case.

Profile objects now have a `positions` member, which is a list of all aligned positions, similar to the same member for alignment objects.

- `selection.debug.function()` now returns the number of derivatives which exceeded the cutoffs.

Bundled version of HDF5 updated to 1.6.6.

Bugfix: always read the first PDB alternate location encountered for each residue, not only A or 1, which resulted in skipping of residues which used neither of these codes.

Bugfix: no longer limit atom file names in PIR comment lines to 80 characters.

Bugfix: user-defined mathematical forms no longer cause an error in `Restraints.pick()` when `restraints_filter` is not -999.

Bugfix: the scoring function will return an error if a coordinate has become infinite or not-a-number (e.g. due to a bad optimization).

Bugfix: the INITIAL_MALIGN3D option works again with TOP scripts.

Bugfix: `selection.superpose()` now returns correct values for `num_equiv_pos` and `num_equiv_dist` (previously these were erroneously the same as the cutoff values).

Bugfix: calling `selection.energy()` from within an optimizer periodic action no longer causes the optimization to cycle endlessly.

Bugfix: `alignment.append_model()` now puts the current model coordinates into the alignment, not the PDB in its original orientation.
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)

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

deviation controls the amount of randomization done by randomize.xyz or randomize.dihedrals. (This can also be set after the object is created, by assigning to 'automodel.deviation'.)

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 2.2.8 for an example.

If inifile is set, an initial model is read from the user-supplied file of the same name. See section 2.2.9 for an example.

assess_methods allows you to request assessment of the generated models (by default, none is done). You can provide a function, or list of functions, for this purpose, or use one or more 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()), or assess.DOPEHR, which uses the DOPE-HR method (see selection.assess.dopehr()). (This can also be set after the object is created, by assigning to 'automodel.assess_methods'.) See section 2.2.3 for an example. 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 \texttt{autosched.fastest} or you can provide your own \texttt{schedule} object (see Section \ref{sec:modeller-schedule}). See the \texttt{modlib/modeller/automodel/autosched.py} file for more information.

See Section \ref{sec:modeller-schedule} 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 \texttt{Python} value \texttt{None} (in which case no additional refinement is done) or use one of the predefined functions in the \texttt{refine} module — \texttt{refine.very_fast}, \texttt{refine.fast}, \texttt{refine.slow}, \texttt{refine.very_slow} or \texttt{refine.slow_large} — which perform different amounts of MD annealing. See the \texttt{modlib/modeller/automodel/refine.py} file for more information.

See Section \ref{sec:modeller-schedule} for an example.

### 4.1.4 automodel.outputs — all output data for generated models

This is produced after \texttt{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 \ref{sec:modeller-summary} 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 \texttt{None} then no randomization is done, and every model will be the same.) You can provide your own function for this purpose, or use one of the functions in the \texttt{randomize} module — \texttt{randomize.xyz} to randomize all coordinates of standard residues, or \texttt{randomize.dihedrals} to randomize dihedral angles.

### 4.1.6 automodel.generate_method — control initial model generation

This is used to build the initial model. It is usually set to \texttt{generate.transfer.xyz}, which builds the model based on the templates, but you can also set it to \texttt{generate.generate.xyz} to build it purely from the internal coordinates, or to \texttt{generate.read.xyz} to read it from a file (see section \ref{sec:modeller-read} for the easiest way to do this).

### 4.1.7 automodel.max_var_iterations — select length of optimizations

This is used to set \texttt{max_iterations} for every call to \texttt{conjugate_gradients()}. 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 \texttt{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 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 .B optimization trace file is produced, using the actions.trace() periodic action. 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.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. (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.charmm_trajectory().

4.1.17 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().

4.1.18 automodel.select_atoms() — select region for optimization

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.

4.1.19 `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 2.2.13 for an example.

4.1.20 `automodel.very_fast()` — request rapid optimization

`very_fast()`

This sets parameters to request very fast optimization of the model(s). See section 2.2.3 for an example.

4.1.21 `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.

4.1.22 `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()`.

4.1.23 `automodel.special_restraints()` — add additional restraints

`special_restraints(aln)`

By default, this routine 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 2.2.10 for an example.

4.1.24 `automodel.nonstd_restraints()` — add restraints on ligands

`nonstd_restraints(aln)`
This routine adds restraints to keep residues that have no defined topology (generally BLK residues, used to transfer ligands directly from templates, as described in Section 2.2.1) in a reasonable conformation. You can override this method if you need to change these restraints.

By default, three sets of restraints are built:

- 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.
- Residue-protein Cα distances are constrained to template values if these are 10Å or less. This preserves ligand coordination.

### 4.1.25 automodel.special_patches() — add additional patches to the topology

```python
special_patches(aln)
```

This routine, which is usually empty, can be redefined by the user to make additional changes to the model topology (for example, to add user-defined disulfides, to renumber the residues, or to rename the chains). See Section 2.2.7 for an example.

### 4.1.26 automodel.user_after_single_model() — analyze or refine each model

```python
user_after_single_model()
```

This routine is called after building each model, before the output PDB 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 2.2.11 for an example.

### 4.1.27 automodel.get_model_filename() — get the model PDB name

```python
get_model_filename(sequence, id1, id2, file_ext)
```

This routine returns the PDB file name of each generated model, usually of the form ‘foo.B000X000Y.pdb’. You can redefine this routine if you don’t like the standard naming scheme. See also `loopmodel.get_loop_model_filename()`.

### 4.1.28 automodel.use_parallel_job() — parallelize model building

```python
use_parallel_job(job)
```

This uses a parallel job object (see Section 6.31) 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.

### 4.2 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

```python
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

```python
loopmodel(env, sequence, alnfile=None, knowns=[], inimodel=None, deviation=None, library_schedule=None, csrfile=None, inifile=None, assess_methods=None, loop_assess_methods=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 file, in which case you should set `inimodel` to the name of the PDB 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`.)

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.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.

4.3.8 loopmodel.select_loop_atoms() — select region for loop optimization

select_loop_atoms()

By default, this selects all atoms near gaps in the alignment for loop optimization. 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.

4.3.9 loopmodel.get_loop_model_filename() — get the model PDB name

get_loop_model_filename(sequence, id1, id2, file_ext)

This routine returns the PDB file name of each generated loop model, usually of the form 'foo.BL000X000Y.pdb'. You can redefine this routine if you don’t like the standard naming scheme. See also automodel.get_model_filename().

4.3.10 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 file is written. It can be redefined by the user to perform analysis or additional refinement on each loop model.

4.3.11 loopmodel.read_potential() — read in the loop modeling potential

read_potential()

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

4.3.12 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 file untouched, use a one-line 'pass' routine.
4.4 dope_loopmodel reference

The dope_loopmodel 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 'dope_loopmodel' 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 dope_loopmodel() — prepare to build models with DOPE loop refinement

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

See loopmodel() for all arguments.

4.5 dopehr_loopmodel reference

This class is identical to dope_loopmodel, 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 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.

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 6.28.

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:

---

1MODELLER can raise the following Python exceptions: ZeroDivisionError, IOError, MemoryError, EOFError, TypeError, NotImplementedError, IndexError, and ValueError.
• \${LIB} is expanded into the \$LIB_MODELLER variable defined in modlib/libs.lib (equal to \$MODINSTALL9v4/modlib);

• \${JOB} is expanded into the root of the script filename, or \'(stdin)\' if instructions are being read from standard input;

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.)

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\', and \'.crd\' in this order, then also with the \'.pdb\' prefix. This search for the atom file is repeated through all the directories in \io_data.atom_files_directory\ (directories are separated by \':\'), unless the input atom filename starts with \'/\', in which case \io_data.atom_files_directory\ is neglected.

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.

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.

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 6.3).

For commands dealing with generating, patching, and mutating molecular topology, see the model class reference, in section 6.6.
5.3. SPATIAL RESTRAINTS

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 5.1 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’. The IUPAC atom names (as opposed to the atom types) are the same as in the templates.

The ‘undf’ 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 is obtained from the $RADII_LIB library using the block atom names (as written out to a PDB file), not the ‘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, Coulomb, and general non-bonded spline terms involving ‘undf’ atoms are ignored by MODELLER.

Please note that if you use ‘BLK’ residues, you must set io.data.hetatm to True, as ‘BLK’ residues are treated as HETATMs.

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 Sali & Blundell, 1993 Sali & 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 `energy_data.dynamic_sphere`).
- Lennard-Jones restraints (see `energy_data.dynamic_lennard`).
- Coulomb restraints (see `energy_data.dynamic_coulomb`).
- Non-bond spline restraints (see `energy_data.dynamic_modeller`).
- GBSA solvent restraints (see Section 6.14).
- EM density restraints (see `energy_data.density`).
- SAXS restraints (see `energy_data.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 6.2). 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.multi_binormal`, which generally only works properly with `features.dihedral`. Both feature types and mathematical forms are described below.

Feature types

Each feature is a [Python](//python.org) 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.minimal_distance(*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.
```
5.3. SPATIAL RESTRAINTS

features.solvent_access(*atom_ids)
Area (in $\text{Å}^2$) exposed to solvent of the given atom. Note that this feature cannot be used in optimization, as first derivatives are always returned as zero.

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.x_coordinate(*atom_ids)
Value of the x coordinate (in angstroms) of the given atom.

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

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

features.dihedral_diff(*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.lower_bound(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.81]

forms.upper_bound(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.82]

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.62]

forms.multi_gaussian(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.65]

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

forms.lennard_jones(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.89]

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.86]

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.83]

forms.multi_binormal(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


CHAPTER 5. MODELLER GENERAL REFERENCE

term. See Eq. [A.75]

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.96]

forms.nd_spline(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:

nd_spline.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.gravity_center(*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).
5.3. SPATIAL RESTRAINTS

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 tetra-
hedral atom and the other two substituents on it (specified by atom_ids).

Example: examples/python/pseudo_atoms.py

```python
from modeller import *
from modeller.scripts import complete_pdb
from modeller.optimizers import conjugate_gradients

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.gravity_center(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.upper_bound(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.x_coordinate(center),
    mean=0.0, stdev=0.1)
rsr.add(r)

# Keep sensible stereochemistry
```

rdr.make(allat, restraint_type='stereo', spline_on_site=False)

# Optimize with CG
cg = conjugate_gradients()
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.

   excluded_atom(atom_id1, atom_id2)

Excludes the given two atoms from the nonbonded list.

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 rigid_body object and adding it to the Restraints.rigid_bodies list.

   rigid_body(*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 conjugate_gradients() and quasi_newton() 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

```python
from modeller import *

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

# Keep residues 1-10 in chain A rigid:
r = rigid_body(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.98]

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 2.2.11 for an example of using symmetry restraints with the automodel class.

Example: examples/commands/define_symmetry.py

```python
# 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 conjugate_gradients, molecular_dynamics

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 = energy_data(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 = conjugate_gradients()
md = molecular_dynamics(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)
### 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 or USER format
- **.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>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.top</td>
<td>TOP script with instructions for a MODELLER job</td>
</tr>
<tr>
<td>.log</td>
<td>log output produced by a MODELLER run</td>
</tr>
<tr>
<td>.ali</td>
<td>alignment or sequences in the PIR format</td>
</tr>
<tr>
<td>.pap</td>
<td>alignment or sequences in the PAP format</td>
</tr>
<tr>
<td>.aln</td>
<td>alignment or sequences in the QUANTA format</td>
</tr>
<tr>
<td>.aln</td>
<td>alignment or sequences in the INSIGHTII format</td>
</tr>
<tr>
<td>.seq, .chn</td>
<td>sequence(s) in the PIR alignment format</td>
</tr>
<tr>
<td>.cod</td>
<td>list of sequence codes</td>
</tr>
<tr>
<td>.grp</td>
<td>list of families in PDB</td>
</tr>
<tr>
<td>.atm, .pdb, .ent</td>
<td>atom coordinates in the PDB or GRASP format</td>
</tr>
<tr>
<td>.crd</td>
<td>atom coordinates in the CHARMM format</td>
</tr>
<tr>
<td>.fit.pdb</td>
<td>fitted protein structures in the PDB format</td>
</tr>
<tr>
<td>.ini</td>
<td>initial MODELLER model</td>
</tr>
<tr>
<td>.B*</td>
<td>MODELLER model in the PDB format</td>
</tr>
<tr>
<td>.D*</td>
<td>the progress of optimization</td>
</tr>
<tr>
<td>.BL*</td>
<td>MODELLER model in the PDB format, in loop modeling</td>
</tr>
<tr>
<td>.DL*</td>
<td>the progress of optimization, in loop modeling</td>
</tr>
<tr>
<td>.IL*</td>
<td>initial MODELLER model, in loop modeling</td>
</tr>
<tr>
<td>.V*</td>
<td>violations profile</td>
</tr>
<tr>
<td>.E*</td>
<td>energy profile</td>
</tr>
<tr>
<td>.rsr</td>
<td>restraints in MODELLER or USER format</td>
</tr>
<tr>
<td>.sch</td>
<td>schedule file for the variable target function optimization</td>
</tr>
<tr>
<td>.mat</td>
<td>matrix of pairwise protein distances from an alignment</td>
</tr>
<tr>
<td>.mat</td>
<td>matrix of pairwise residue type–residue type distance scores</td>
</tr>
<tr>
<td>.sim.mat</td>
<td>matrix of pairwise residue type–residue type similarity scores</td>
</tr>
<tr>
<td>.lib</td>
<td>various MODELLER libraries</td>
</tr>
<tr>
<td>.psa</td>
<td>residue solvent accessibilities</td>
</tr>
<tr>
<td>.sol</td>
<td>atomic solvent accessibilities</td>
</tr>
<tr>
<td>.ngh</td>
<td>residue neighbors</td>
</tr>
<tr>
<td>.dih</td>
<td>mainchain and sidechain dihedral angles</td>
</tr>
<tr>
<td>.ssm</td>
<td>secondary structure assignment</td>
</tr>
<tr>
<td>.var</td>
<td>sequence variability profile from multiple alignment</td>
</tr>
<tr>
<td>.asgl</td>
<td>data for plotting by [Asgl]</td>
</tr>
</tbody>
</table>

Table 5.1: List of file types.
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 Python keywords. 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 io_data() object, and is used by commands that need to read coordinate files, while edat should be an energy_data 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; 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 io_data object, which is used as the default by all routines which take an io argument (used for reading coordinate files). See Section 6.3.

6.2.3.environ.edat — default objective function parameters

This is an energy_data 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() 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() 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()` (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`

```python
# 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 energy_data class: objective function parameters

The energy_data 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 energy_data() — create a new set of objective function parameters

```python
energy_data(copy=None, **kwargs)
```

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

```python
edat = energy_data(contact_shell=7.0)
```

Alternatively, you can set parameters in an existing object:

```python
edat.contact_shell = 7.0
```

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

```python
env = environ()
env.edat.contact_shell = 7.0
```

6.3.2 energy_data.contact_shell — nonbond distance cutoff

This defines the maximal distance 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.

If undefined (−999), the default value is the maximum of:

- If energy_data.dynamic_sphere is True, twice the radius of the largest atom multiplied by energy_data.radius_factor (in the case of the all non-hydrogen atoms model, this is 3.2 Å).
- If energy_data.dynamic_lennard is True, energy_data.lennard_jones_switch[1].
- If energy_data.dynamic_coulomb is True, energy_data.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 energy_data.contact_shell must be found in combination with energy_data.update_dynamic (see also below). Good values are 4Å for energy_data.contact_shell and 0.39Å for energy_data.update_dynamic when no Lennard-Jones and Coulomb terms are used; if energy_data.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 energy_data.update_dynamic — nonbond recalculation threshold

This sets the cumulative maximal atomic shift during optimization that triggers recalculation of the list of atom–atom non-bonded pairs. It should be set in combination with energy_data.contact_shell.

For soft-sphere overlap, to be absolutely sure that no unaccounted contacts occur, energy_data.update_dynamic has to be equal to (energy_data.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 energy_data.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 energy_data.sphere_stdev — soft-sphere standard deviation

This sets the standard deviation of the lower bound harmonic potential used for the soft-sphere restraints. See energy_data.dynamic_sphere.

6.3.5 energy_data.dynamic_sphere — calculate soft-sphere overlap restraints

If set to True, 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.81), with standard deviation energy_data.sphere_stdev, 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 energy_data.dynamic_lennard — calculate Lennard-Jones restraints

If set to True, dynamic Lennard-Jones restraints are calculated, using equation A.89.

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

6.3.7 energy_data.dynamic_coulomb — calculate Coulomb restraints

If set to True, dynamic Coulomb (electrostatic) restraints are calculated, using equation A.86.

6.3.8 energy_data.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.

6.3.9 energy_data.excl_local — exclude certain local pairs of atoms

This specifies whether or not the atoms in a chemical bond, chemical angle, dihedral/improper 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 $C_\alpha - C_\alpha+2$ terms.

6.3.10 energy_data.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.

6.3.11 energy_data.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.89.
6.3.12 energy_data.coulomb_switch — Coulomb switching parameters

These are the parameters $f_1$ and $f_2$ to the Coulomb switching function, which smooths the potential down to zero; see equation A.86.

6.3.13 energy_data.relative_dielectric — relative dielectric

This sets the relative dielectric $\epsilon_r$, used in the calculation of the Coulomb energy (equation A.86).

6.3.14 energy_data.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). energy_data.covalent_cys = True is slow and only has an effect when certain statistical non-bonded potentials are used (i.e., energy_data.dynamic_modeller is True and the non-bonded library has been derived considering the disulfide effect). Thus, it should generally be set to False.

6.3.15 energy_data.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 energy_data.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 energy_data.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 $O(n^2)$ 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 energy_data.nlogn_use or when the box size for this algorithm would have to be larger than 8Å.

6.3.17 energy_data.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 io_data class: coordinate file input parameters

The io_data 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 io_data() — create a new input parameters object

io_data(copy=None, **kwargs)

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

io = io_data(hetatm=True)

Alternatively, you can set parameters in an existing object:

io.hetatm = True

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

env = environ()
env.io.hetatm = True

Please note that io_data.hetatm, io_data.hydrogen, and io_data.water are ignored when reading UHBD files. When reading CHARMM files, io_data.hetatm is ignored.

6.4.2 io_data.hetatm — whether to read HETATM records

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

6.4.3 io_data.hydrogen — whether to read hydrogen atoms

If set to True, then hydrogen atoms are read from PDB 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 io_data.water — whether to read water molecules

If set to True, then water molecules are read. (For PDB 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 io_data.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.)
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] (it also reads which subsets of atoms correspond to each `Topology.submodel` from library 'MODELS.LIB'). 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.

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 group_restraints class, in section 6.13 for more information.

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. 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().

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 6.16) 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 in a REMARK header.

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. 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 in a REMARK header (and is read back from Modeller-produced PDB files).

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.

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 group_restraints 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 energy_data.dynamic_modeller must also be set to True.
6.6. **THE MODEL CLASS: HANDLING OF ATOMIC COORDINATES, AND MODEL BUILDING**

Example: See [group_restraints()](#) 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 6.22 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 6.21.

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)

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.

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)

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.

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)

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

Example: See selection() command.

6.6.15 model.read() — read coordinates for MODEL

read(file, model_format='PDB', model_segment=('FIRST:@', 'LAST:@'), io=None)

This command reads the atomic coordinates, atom names, residue names, residue numbers, 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' format), or patches (such as disulfides); to make these assignments, which are necessary for almost all energy commands, use model.generate_topology(). All real and pseudo atoms are selected. 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.

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

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 files with alternate locations (characters in column 17 of ATOM or HETATM records), 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.)

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

Example: `examples/commands/read_model.py`

```python
# Example for: model.read(), model.write()

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

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` and `patch_default` can be used to adjust the topology; see `complete_pdb()` for more details. See also `alignment.append_sequence()`.

Example: `examples/commands/build_sequence.py`

```python
# 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("AFVTDNCIK/CKYTDCVEVC")
mdl.write(file='sequence.pdb')

# Build an alignment from one-letter codes
aln.append_sequence("AF---VVTDN---CIK------")
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)

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

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

'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 Macromolecular Crystallographic Information File (mmCIF) format.

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.

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, 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 "$\text{RESTYP}\_	ext{LIB}$" 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 the `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.

If only one chain is generated, it is given a blank chain ID. Otherwise, the chains are labeled 'A', 'B', 'C' and so on. 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.resnum_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` 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.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.

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.charmm_trajectory()`

Example: See `actions.charmm_trajectory()` command.

6.6.21 `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

```python
# 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:'],
        mdl.residues['22:']))

# 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()
```

6.6.22 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. 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

```python
# Example for: model.patch_ss_templates() and model.patch_ss()
```
# This will patch CYS-CYS disulfide bonds using disulfides in aligned templates:

```python
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.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()
```

6.6.23 model.patch_ss() — guess MODEL disulfides from model structure

```python
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.
CHAPTER 6. MODELLER COMMAND REFERENCE

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.

6.6.24  `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. 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, they are set to values close to those of the neighboring atoms. If even this fails, they are set randomly.

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 (±0.2 Å) 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.

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_STICK'`, 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`

```python
# 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')
```
6.6. THE MODEL CLASS: HANDLING OF ATOMIC COORDINATES, AND MODEL BUILDING

# Add PDB remarks for readability
mdl.remark = """REMARC 4 Extended-chain model of 1fas
REMARC 4 Built from internal coordinates only""

# Write the coordinates to a PDB file:
mdl.write(file='1fas.ini')

6.6.25 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 less than 0, the transferred coordinates and ($B_{iso}$) for a given target atom are the average of the coordinates of all the equivalent template atoms. Otherwise, the transferred coordinates 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`.

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`

```python
# 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')
```

### 6.6.26 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`

```python
# 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')

6.6.27 model.rename_segments() — rename MODEL segments

rename_segments(segment_ids, renumber_residues=[])

This command re-labels residue numbers in each chain (i.e., segment) so that they start with renumber_residues[iseg]. In addition, the single character PDB chain id's are also assigned: They are obtained from the corresponding elements of segment_ids. Thus, there should be as many elements in segment_ids and renumber_residues as there are chains in the current MODEL.

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 (here there are two 'chains': protein and the HETATM
    # water molecules).
    from modeller import *
    # Read the MODEL with all HETATM and water records (so there are two 'chains'):
    env = environ()
    env.io.atom_files_directory = ['../atom_files']
    env.io.hetatm = True
    env.io.water = True
    mdl = model(env, file='1fas')
    # Assign new segment names and write out the new model:
    mdl.rename_segments(segment_ids=('X', 'Y'))
    mdl.write(file='1fas.ini')

6.6.28 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 \[\text{IUPAC-IUB, 1970}\text{Kendrew et al., 1970}\]. These residues, pairs of atoms, and dihedral angles are:

- Phe, Tyr: (CD1, CD2), (CE1, CE2); \(\chi_2\);
• Asp: (OD1, OD2); \( \chi_2 \);
• Glu: (OE1, OE2); \( \chi_3 \);
• Arg: (NH1, NH2); \( \chi_4 \).

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`

```python
# 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')
```

6.6.29 `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`

```python
# 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')
```
6.6.30 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 nonspherical 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

```python
# 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: ", r.translation
```

6.6.31 model.write_data() — write derivative MODEL data

write_data(output, file, 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 writes the selected types of data about the MODEL to a corresponding file. The last such property (from the list below) is also assigned to the $B_{iso}$ field of the model (this can later be written out with model.write() to a PDB file, where it appears as the temperature factor). The root of the output filenames is given by the file variable.

The data to be calculated are specified by concatenating the corresponding keywords in the output variable:

- 'ALL': All types of data are written to the corresponding files.
- 'PSA': The atomic and residue solvent accessibilities are written to the .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). The single reference is [Sali & Overington, 1994]. Accessibilities are calculated with scaled radii from the $MODELS\_LIB$ library, as specified by \texttt{topology.submodel}. The radii are scaled by \texttt{energy.data.radii_factor} which should usually be set to 1. If \texttt{output} also contains ATOMIC\_SOL, atomic accessibilities in Å\(^2\) are assigned to \(B_{iso}\), otherwise residue accessibility of type \texttt{accessibility_type} (from 1 to 10, for the columns in the \texttt{.psa} file) is assigned. If \texttt{surfTyp} is 1, contact accessibility is calculated; if 2, surface accessibility is returned.

- **NGH**: Residue neighbors of each residue are listed to a \texttt{ngh} file. The \texttt{MODELLER} definition of a residue–residue contact used in restraints derivation is applied [Sali & Blundell, 1993]: Any pair of residues that has any pair of atoms within \texttt{neighbor_cutoff} Å of each other are in contact.

- **DIH**: All the dihedral angle types defined in the $RESDIH\_LIB$ library (mainchain, sidechain, and the virtual dihedral between four successive \(C_\alpha\) atoms, starting with the previous residue) are written to a \texttt{.dih} file. One column from this file, as selected by \texttt{accessibility_type}, is also assigned to \(B_{iso}\).

- **SSM**: Secondary structure assignments are written to a \texttt{.ssm} file. The algorithm for secondary structure assignment depends on the \(C_\alpha\) positions only and is based on the distance matrix idea described in [Richards & Kundrot, 1988]. For each secondary structure type, a ‘library’ \(C_\alpha\) 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 \(\alpha\)-helix (secondary structure type 2) and one for the \(\beta\)-strand (type 1). 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 \(C_\alpha\) 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 \(C_\alpha\) position;
   - Assign the current secondary structure type to all \(C_\alpha\)’s in all segments whose DRMS deviation and maximal distance difference are less than some cutoffs (\(P_1 < \text{cut}_1\), \(P_1 < \text{cut}_2\)) and are not already assigned to ‘earlier’ secondary structure types;

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 (\(P_1 > \text{cut}_3\), \(P_2 > \text{cut}_4\)). 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. The kink residue type is \(-2\).

3. If the current secondary structure type is \(\beta\)-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 (\(P_3 < \text{cut}_3\)). Currently, only one pass of this elimination is done, but could be repeated until self-consistency.

4. Remove those segments that are shorter than the cutoff (\(\text{cut}_b\)) 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 are assigned to the \(B_{iso}\) field. Local mainchain curvature at residue \(i\) is defined as the angle between the least-squares lines through \(C_\alpha\) atoms \(i - 3\) to \(i\) and \(i\) to \(i + 3\).

- **HBONDS**: Hydrogen bonds between amino acid residues are written to a \texttt{.hbnds} file. A list of donors and acceptors in the 20 naturally occurring amino acids, specified by \texttt{dnr_acpt_lib}, is utilized in the computation of H-bonds. Hydrogen bonds are reported if the donor–acceptor distance is between 2.5 and 3.5 Å and the donor-acceptor-acceptor antecedent angle is larger than 120 degrees. The model’s \(B_{iso}\) field is not changed by this property.

6.6.32 \texttt{model.make_region()} — define a random surface patch of atoms

\texttt{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 Å²). 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 `Biso` 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

```python
# 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 with appropriate probe_radius
myedat = energy_data()
myedat.radii_factor = 1.6
mdl.write_data(file='1fdn', 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')
```

**6.6.33 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 `Biso` (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`

```python
# 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')
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')
```

6.6.34 `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.

### 6.6.35 model.saxs_intens() — Calculate SAXS intensity from model

```python
saxs_intens(saxsd, filename, fitflag=False)
```

Calculate SAXS intensity from model. See Section 6.26.

### 6.6.36 model.saxs_pr() — Calculate $P(r)$ of model

```python
saxs_pr(saxsd, filename)
```

Calculate $P(r)$ from model. See Section 6.26.

### 6.6.37 model.saxs_chifun() — Calculate SAXS score chi from model

```python
saxs_chifun(transfer_is, edat=None)
```

Calculate SAXS score from model. See Section 6.26.

### 6.6.38 model.assess_ga341() — assess a model with the GA341 method

```python
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 [Melo et al., 2002, John & Sali, 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.seqid`.

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.

Example: `examples/commands/assess_ga341.py`

```python
# 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()
6.7 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.symmetry — all symmetry restraints

This is a list of all symmetry restraints on the model. See section 5.3.5 for more information.

6.7.5 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.

6.7.6 Restraints.make() — make restraints

make(atmsel, restraint_type, spline_on_site, residue_span_range=(0, 99999), residue_span_sign=True, residue_grouping=1, 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, distance_atoms=('CA', 'CA'), 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 Sali & Blundell, 1993 Sali & 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 ≤ |ir2 − ir1| ≤ r2 when residue_span_sign = False and r1 ≤ (ir2 − ir1) ≤ 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 restrain_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 restrain_selAtoms 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 mean values and force constants are obtained from the parameter library in memory. Only those dihedral angles are restrained that have all or at least restrain_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 improper angles are restrained that have all or at least restrain_sel_atoms in the selection atmsel.


- ‘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 ‘$RAD114LIB’ library. Only those non-bonded pairs are restrained that have all or at least [energy.data.nonbonded_sel_atoms](variable) 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 [energy.data.nonbonded_sel_atoms](variable) 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 [energy.data.nonbonded_sel_atoms](variable) 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 [energy.data.nonbonded_sel_atoms](variable) 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 intersegment 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
6.7. THE RESTRAINTS CLASS: STATIC RESTRAINTS

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 one of 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 energy_data.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 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 one of 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 energy_data.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 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 one of the residue_span_range & residue_span_sign criterion in the selection atmsel. The means and standard deviations for the dihedral Gaussian restraints are obtained from the $RESIDH_LIB and $MNCH_LIB libraries and their weights from the MDT tables, which are read in as specified by MDT_LIB in $LIB/1ibs.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 $RESIDH_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 $ 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.

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 residue_grouping variable defines the way in which residues are grouped, as defined in $LIB/resgrp.lib, and accessibility_type 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 = energy_data(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)
```

### 6.7.7 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)
```

**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 $\sigma$ is calculated from an analytic model specified by `distance_rsr_model`. Use model 5 for $C_{\alpha}-C_{\alpha}$ distances and model 6 for N–O distances. For models 1 through 6, this standard deviation is transformed by $\sigma' = a + b \times (\sigma + W_g)$ where $a$ and $b$ are given by `restraint_stdev` and $W_g$ is a gap weighting function of the form $W_g = 0.6 \times \max(0, 4 - g_{av})$. $g_{av}$ 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 $\sigma'' = \sigma' + \sum_i [d + e \times \max(0, f - g_i)]$ where the sum is over each of the atoms $i$ involved in the distance, $d e$.
and $f$ are given by $\text{restraint}_\text{stdev}^2$, 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}_\text{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 `energy.data.excl_local` (see `conjugate_gradients()`).

### 6.7.8 Restraints.unpick_all() — unselect all restraints

```python
unpick_all()
```

This unselects all of the current restraints.

### 6.7.9 Restraints.clear() — delete all restraints

```python
clear()
```

This deletes all of the current restraints, including pseudo atoms, excluded pairs, rigid bodies, and symmetry restraints.

### 6.7.10 Restraints.pick() — pick restraints for selected atoms

```python
pick(atmsel, residue_span_range=(0, 99999), restraint.sel_atoms=1, restraints.filter=physical.values(default=-999.000000))
```

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. `energy.data.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')

6.7.11  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 constraints on a dihedral angle or a pair of dihedral angles. Such constraints 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().

6.7.12  Restraints.remove_unpicked() — remove unselected restraints

remove_unpicked()

This command permanently removes all the unselected restraints from memory. See also Restraints.condense().
6.7.13  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.

6.7.14  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]

```python
# 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.atoms
 cispeptide(rsr, atom_ids1=(a['O:56'], a['C:56'], a['N:57'], a['CA:57']),
aatom_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.upper_bound(group=physical.xy_distance,
 feature=features.distance(a['CA:5'], a['CA:15']),
 mean=10., stdev=0.1))
```
6.7.15  **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.

6.7.16  **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](examples/commands/reindex_restraints.py)

```python
# 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')
```
6.7. THE RESTRAINTS CLASS: STATIC RESTRAINTS

# 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')

# 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)

6.7.17 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. For example, the poly-Gaussian range is from $m - \text{spline}\_\text{range} \times \sigma_m$ to $M + \text{spline}\_\text{range} \times \sigma_M$, where $m$ and $M$ are the minimal and maximal means of the basis pdfs, and $\sigma_m$ and $\sigma_M$ are their corresponding standard deviations.

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, spline\_dx is decreased 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.
6.7.18  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.

Example: See Restraints.make() command.

6.7.19  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.

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, Cα–Cα distances for \( i-j \in \{2-9\} \), Cα–O distances for \( i-j \in \{2-9\} \), and O–O distances for \( i-j \in \{2-6\} \). Note that this requires all heavy atoms to be present to work properly, so will not work with the Cα-only topology.

In many cases (e.g., most comparative modeling runs) you will already have binormal, Cα–Cα, 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

```python
# Example for model.build_sequence(), secondary_structure.alpha()

from modeller import *
from modeller.optimizers import conjugate_gradients

# 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 = conjugate_gradients()
cg.optimize(allatoms, max_iterations=100)
m.write(file='alpha-helix.pdb')
```

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 $\beta$-strand

strand(residues)

This makes restraints enforcing an extended $\beta$-strand conformation for the residue segment specified by residues (which can be created using the `model.residue_range()` function). This is achieved by applying $\Phi, \Psi$ binormal restraints only. These binormal restraints force the mainchain conformation into class “B”, except for the Pro residues which are restrained to class “P” [Sali & Blundell, 1993].

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 2.2.10 for an example.

6.8.3 sheet() — make a $\beta$-sheet

sheet(atom1, atom2, sheet_h_bonds)

This calculates H-bonding restraints for a pair of $\beta$-strands. atom1 and atom2 specify the first H-bond in the $\beta$-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 2.2.10 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.

To add objects to a selection, you can list them when you create the selection with 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 5Å of atom CA:1:A, OR currently within 3Å 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 5Å of atom CA:1:A, AND also currently within 3Å
# 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 5Å of atom CA:1:A, OR currently within 3Å 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'))

# 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:
mdl1 = 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(mdl1).only_mainchain()
atmsel.superpose(mdl2, aln)

# Pick and superpose sidechain atoms:
atmsel = selection(mdl1).only_sidechain()
atmsel.superpose(mdl2, aln)

# Pick and superpose CA and CB atoms:
atmsel = selection(mdl1).only_atom_types('CA CB')
atmsel.superpose(mdl2, aln)

# Pick and superpose all atoms:
atmsel = selection(mdl1)
atmsel.superpose(mdl2, aln)

# Pick and superpose CA and CB atoms in one segment only:
atmsel = selection(mdl1.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 = mdl1.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(mdl1.residue_range('2:A', '10:A')).select_sphere(6.0)
atmsel.superpose(mdl2, aln)

# Pick all atoms in the model
atmsel = selection(mdl1)

# Pick all atoms in all loops (ie residues within 2 positions # of 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

```
def 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 are also selected. The original selection is unchanged.

6.9.4 `selection.by_residue()` — make sure all residues are fully selected

```
def 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. The original selection is unchanged.

6.9.5 `selection.select_sphere()` — select all atoms within radius

```
def 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

```
def only_mainchain():
    ...
```

This returns a new selection containing only mainchain atoms (i.e., atom types O, OT1, OT2, C, CA, N) from the current selection.

6.9.7 `selection.only_sidechain()` — select only sidechain atoms

```
def 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

```
def only_atom_types(atom_types):
    ...
```

This returns a new selection containing only atoms from the current selection of the given 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 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)
CHAPTER 6. MODELLER COMMAND REFERENCE

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`

```python
# 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’]```
6.9. **THE SELECTION CLASS: HANDLING OF SETS OF ATOM COORDINATES**

```python
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_model.py](#)

```python
# 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 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. 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 *
6.9. THE SELECTION CLASS: HANDLING OF SETS OF ATOM COORDINATES

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.

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(). 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:
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 \( \text{rms} \) _cutoff_
- **r.cutoff_drms**: the DRMS after superposition of distances within \( \text{rms} \) _cutoff_
- **r.rotation**: the rotation matrix that was used to transform \( \text{mdl2} \) (applied first), if fit is True
- **r.translation**: the translation that was used to transform \( \text{mdl2} \) (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 \( \text{rms} \) _cutoff_
- **r.num_equiv_cutoff_dist**: the number of equivalent distances within \( \text{rms} \) _cutoff_

**Example:** examples/commands/superpose.py

```python
# 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`

```python
# 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`

```python
# 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 = aligment(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 $-\text{deviation}$ to $+\text{deviation}$ 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 $\text{RESDIH_LIB}$ library: alpha, phi, psi, omega, chi1, chi2, chi3, chi4, chi5. Dihedral angle 'alpha' is the virtual C$_\alpha$ dihedral angle defined by four consecutive C$_\alpha$ 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:
6.9. **THE SELECTION CLASS: HANDLING OF SETS OF ATOM COORDINATES**

```python
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

```python
unbuild()
```

This command undefines all of the Cartesian coordinates of the selected atoms.

### 6.9.25 selection.hot_atoms() — atoms violating restraints

```python
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, 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:**

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`

```python
# 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),
vio1_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 ‘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 $\chi_i$ restraints, ($\Phi, \Psi$) 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 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', '39'),
               ('3', '22'),
               ('53', '59'),
               ('41', '52')]:
        mdl.patch(residue_type='DISU', residues=[mdl.residues[r] for r in ids])

mdl = complete_pdb(env, "1fas", 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=energy_data(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

default.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.
6.9. **THE SELECTION CLASS: HANDLING OF SETS OF ATOM COORDINATES**

Example: `examples/scoring/debug_function.py`

```python
# 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 *  # 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

```python
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]. 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.scaled. See selection.energy() for more details. 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.) Currently the score is unnormalized with respect to the protein size and has an arbitrary scale, therefore scores from different proteins cannot be compared directly. A normalization scheme is under development.

Example: examples/commands/assess_dope.py

```python
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')

mdl = complete_pdb(env, '../atom_files/1fdx.B99990001.pdb')

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).

Example: See selection.assess_dope() command.
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.
<table>
<thead>
<tr>
<th>Python object</th>
<th>Index</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical.bond</td>
<td>1</td>
<td>Bond length potential</td>
</tr>
<tr>
<td>physical.angle</td>
<td>2</td>
<td>Bond angle potential</td>
</tr>
<tr>
<td>physical.dihedral</td>
<td>3</td>
<td>Stereochemical cosine dihedral potential</td>
</tr>
<tr>
<td>physical.improper</td>
<td>4</td>
<td>Stereochemical improper dihedral potential</td>
</tr>
<tr>
<td>physical.soft_sphere</td>
<td>5</td>
<td>soft-sphere overlap restraints</td>
</tr>
<tr>
<td>physical.lennard_jones</td>
<td>6</td>
<td>Lennard-Jones 6–12 potential</td>
</tr>
<tr>
<td>physical.coulomb</td>
<td>7</td>
<td>Coulomb point-point electrostatic potential</td>
</tr>
<tr>
<td>physical.h_bond</td>
<td>8</td>
<td>H-bonding potential</td>
</tr>
<tr>
<td>physical.ca_distance</td>
<td>9</td>
<td>Distance restraints 1 (C₁–C₁)</td>
</tr>
<tr>
<td>physical.n_o_distance</td>
<td>10</td>
<td>Distance restraints 2 (N–O)</td>
</tr>
<tr>
<td>physical.phi_dihedral</td>
<td>11</td>
<td>Mainchain Φ dihedral restraints</td>
</tr>
<tr>
<td>physical.psi_dihedral</td>
<td>12</td>
<td>Mainchain Ψ dihedral restraints</td>
</tr>
<tr>
<td>physical.omega_dihedral</td>
<td>13</td>
<td>Mainchain ω dihedral restraints</td>
</tr>
<tr>
<td>physical.chi1_dihedral</td>
<td>14</td>
<td>Sidechain χ₁ dihedral restraints</td>
</tr>
<tr>
<td>physical.chi2_dihedral</td>
<td>15</td>
<td>Sidechain χ₂ dihedral restraints</td>
</tr>
<tr>
<td>physical.chi3_dihedral</td>
<td>16</td>
<td>Sidechain χ₃ dihedral restraints</td>
</tr>
<tr>
<td>physical.chi4_dihedral</td>
<td>17</td>
<td>Sidechain χ₄ dihedral restraints</td>
</tr>
<tr>
<td>physical.disulfide_distance</td>
<td>18</td>
<td>Disulfide distance restraints</td>
</tr>
<tr>
<td>physical.disulfide_angle</td>
<td>19</td>
<td>Disulfide angle restraints</td>
</tr>
<tr>
<td>physical.disulfide_dihedral</td>
<td>20</td>
<td>Disulfide dihedral angle restraints</td>
</tr>
<tr>
<td>physical.lower_distance</td>
<td>21</td>
<td>X lower bound distance restraints</td>
</tr>
<tr>
<td>physical.upper_distance</td>
<td>22</td>
<td>X upper bound distance restraints</td>
</tr>
<tr>
<td>physical.sd_mn_distance</td>
<td>23</td>
<td>Distance restraints 3 (SDCH–MNCH)</td>
</tr>
<tr>
<td>physical.chi5_dihedral</td>
<td>24</td>
<td>Sidechain χ₅ dihedral restraints</td>
</tr>
<tr>
<td>physical.phi_psi_dihedral</td>
<td>25</td>
<td>(Φ, Ψ) binomial dihedral restraints</td>
</tr>
<tr>
<td>physical.sd_sd_distance</td>
<td>26</td>
<td>Distance restraints 4 (SDCH–SDCH)</td>
</tr>
<tr>
<td>physical.xy_distance</td>
<td>27</td>
<td>Distance restraints 5 (X–Y)</td>
</tr>
<tr>
<td>physical.nmr_distance</td>
<td>28</td>
<td>NMR distance restraints 6 (X–Y)</td>
</tr>
<tr>
<td>physical.nmr_distance2</td>
<td>29</td>
<td>NMR distance restraints 7 (X–Y)</td>
</tr>
<tr>
<td>physical.min_distance</td>
<td>30</td>
<td>Minimal distance restraints</td>
</tr>
<tr>
<td>physical.nonbond_spline</td>
<td>31</td>
<td>Non-bonded spline restraints</td>
</tr>
<tr>
<td>physical.accessibility</td>
<td>32</td>
<td>Atomic accessibility restraints</td>
</tr>
<tr>
<td>physical.density</td>
<td>33</td>
<td>Atom density restraints</td>
</tr>
<tr>
<td>physical.absposition</td>
<td>34</td>
<td>Absolute position restraints</td>
</tr>
<tr>
<td>physical.dihedral_diff</td>
<td>35</td>
<td>Dihedral angle difference restraints</td>
</tr>
<tr>
<td>physical.gbssa</td>
<td>36</td>
<td>GBSA implicit solvent potential</td>
</tr>
<tr>
<td>physical.em_density</td>
<td>37</td>
<td>EM density fitting potential</td>
</tr>
<tr>
<td>physical.saxs</td>
<td>38</td>
<td>SAXS restraints</td>
</tr>
<tr>
<td>physical.symmetry</td>
<td>39</td>
<td>Symmetry restraints</td>
</tr>
</tbody>
</table>

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 conjugate_gradients() — optimize atoms given restraints, with CG

conjugate_gradients(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 energy_data 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.write_structure() 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 energy_data.contact_shell to something large (e.g., 8Å) and energy_data.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 $\text{energy.data.nlogn\_use} = 9999$).

Example: `examples/scoring/optimize.py`

```python
# Example for: conjugate_gradients(), molecular_dynamics(), 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 conjugate_gradients, molecular_dynamics, 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 = conjugate_gradients(output='REPORT')
md = molecular_dynamics(output='REPORT')

# Open a file to get basic stats on each optimization
trcfil = file(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.write_structure(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 quasi_newton() — optimize atoms with quasi-Newton minimization

quasi_newton(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 conjugate_gradients() 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 conjugate_gradients() plus one additional max_atom_shift argument. This is used to limit the maximum size of an optimization move.

6.11.3 molecular_dynamics() — optimize atoms given restraints, with MD

molecular_dynamics(output='NO_REPORT', cap_atom_shift=0.2, md_time_step=4.0, init_velocities=True, temperature=293.0, md_return='FINAL', equilibrate=99999, 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. This value should be smaller than energy_data.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 conjugate_gradients() for a description of the other parameters and the edat and actions optional keyword arguments.

Example: See conjugate_gradients() command.

6.11.4 actions.write_structure() — write out the model coordinates

write_structure(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., `conjugate_gradients()` or `molecular_dynamics()`).

`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 `conjugate_gradients()` command.

6.11.5 actions.trace() — write out optimization energies, etc

```python
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 `conjugate_gradients()` command.

6.11.6 actions.charmm_trajectory() — write out a CHARMM trajectory

```python
charmm_trajectory(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.write_structure()` 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](http://www.cgl.ucsf.edu/chimera) or [VMD](http://www.ks.uiuc.edu/Research/vmd/). 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., `conjugate_gradients()` or `molecular_dynamics()`).

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`

```python
# Example for PSF and binary trajectory output

from modeller import *
from modeller.scripts import complete_pdb
from modeller.optimizers import molecular_dynamics, actions
```

Note that binary trajectory files are machine dependent; it is up to the visualization software to do any necessary byte-swapping.
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)

# 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 = molecular_dynamics(output='REPORT')
md.optimize(atmsel, temperature=300, max_iterations=100,
actions=actions.charmm_trajectory(5, filename=code+'.dcd'))

6.11. User-defined optimizers

The optimizers module also provides a state_optimizer 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

```python
from modeller.optimizers import state_optimizer

class SteepestDescent(state_optimizer):
    """""""Very simple steepest descent optimizer, in Python""""

    # Add options for our optimizer
    _ok_keys = state_optimizer._ok_keys + ('min_atom_shift', 'min_e_diff',
        'step_size', 'max_iterations')
```
```python
def __init__(self, step_size=0.0001, min_atom_shift=0.01, min_e_diff=1.0, max_iterations=None, **vars):
    state_optimizer.__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
    state_optimizer.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`

```python
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 6.7.6); (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 6.7.6) 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
equiv.libs.topology.read(file='$(LIB)/top_heav.lib')
equiv.libs.parameters.read(file='$(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.conjugate_gradients
MD = optimizers.molecular_dynamics
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, fh.

Example: See schedule() command.
6.13 The group_restraints class: restraints on atom groups

The group_restraints 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 energy_data.dynamic_modeller is set to True.

6.13.1 group_restraints() — create a new set of group restraints

```python
group_restraints(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 group_restraints.append().

Example: examples/commands/group_restraints.py

```python
# Example for: group_restraints()
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 = group_restraints(env, classes="$(LIB)/atmcls-melo.lib",
parameters="$(LIB)/melo1-dist.lib")

# Read DOPE loop modeling potential
# gprsr = group_restraints(env, classes="$(LIB)/atmcls-mf.lib",
# parameters="$(LIB)/dist-mf.lib")

# Read DOPE-HR loop modeling potential
# gprsr = group_restraints(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
atmsel = selection(mdl.residue_range('1', '10'))
atmsel.energy()
```

6.13.2 group_restraints.append() — read group restraint parameters

```python
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.
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 `dope_loopmodel()` 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

```python
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 `energy_data.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 `energy_data.contact_shell`. The electrostatic component of GB/SA is also switched using the value of `energy_data.coulomb_switch`.

Example: `examples/scoring/gbsa.py`

```python
# 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 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.

6.15.1 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.

6.15.2 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).

6.15.3 alignment.append() — read sequences and/or their alignment

append(file, align_codes='all', atom_files=None, remove_gaps=True, alignment_format='PIR', io=None)

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.

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 '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.

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 INSIGHT 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 \texttt{remove\_gaps = True}, positions with gaps in all selected sequences are removed from the alignment.
The \texttt{io} argument is required since PIR files can contain empty sequences; in this case, the sequence is read from the corresponding PDB file.
For 'PIR' and 'FASTA' files, the \texttt{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 \texttt{FileFormatError} if the alignment file format is invalid.

Example: \texttt{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()
```

### 6.15.4 \texttt{alignment.clear()} — delete all sequences from the alignment

\texttt{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.

### 6.15.5 \texttt{alignment.read\_one()} — read sequences one by one from a file

\texttt{read\_one(file, remove\_gaps=False, alignment\_format='PIR', io=None)}

Output: \texttt{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, \texttt{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, \texttt{False} is returned.

Arguments are as for \texttt{alignment.append()} Note that only 'PIR' or 'FASTA' format files can be read with this command. \texttt{file} should be an open file handle (see \texttt{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.

Example: examples/commands/alignment_read_one.py

```python
# 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()
```

6.15.6 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\textalpha\ 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\textalpha\ 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.
6.15.7 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 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.

6.15.8 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()

6.15.9 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

```python
# 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)
```

6.15.10 alignment.append_model() — copy model sequence and coordinates to alignment

```python
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`.)

Example: examples/commands/aln_append_model.py

```python
# 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')

6.15.11  `alignment.append_sequence()` — add a sequence from one-letter codes

```python
append_sequence(sequence)
```

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.

Example: See `model.build_sequence()` command.

6.15.12  `alignment.append_profile()` — add profile sequences to the alignment

```python
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()`.

6.15.13  `alignment.write()` — write sequences and/or their alignment

```python
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 `modfile.File()` object open in write mode (in which case the alignment is appended to the 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 at each position, 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.

6.15.14 alignment.edit() — edit overhangs in alignment

edit(overhang, edit_align_codes, base_align_codes, min_base_entries, io=None)

This command edits the overhangs in the alignment.

eidt_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.

Example: examples/commands/edit_alignment.py

```python
# 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()
environ().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')
```

6.15.15 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

```python
# Example for: alignment.describe()

# Describe the sequences and structures in the alignment.

from modeller import *

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

aln = alignment(env, file='toxin.ali', align_codes=('2ctx', '2abx'))
aln.describe()
```
6.15.16 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** 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

```python
# 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, matrix_file='$LIB/as1.sim.mat', max_gaps_match=1,
                       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:
environ.dendrogram(matrix_file='toxin.mat', cluster_cut=-1.0)
```

6.15.17 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 \( x' = -x + x_{\text{max}} \).

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 \( B_{\text{iso}} \) field of the model mdl, which must correspond to the first sequence in the alignment.

**Example:** See `alignment.id_table()` command.

### 6.15.18 alignment.align() — align two (blocks of) sequences

```python
align(off_diagonal=100, local_alignment=False, matrix_offset=0.0, gap_penalties_1d=(-900.0, -50.0), n_subopt=1, 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)
```

**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 \( |i - j| > \text{off_diagonal} \) 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 \texttt{align\_what}\ is set to \texttt{‘ALIGNMENT’}, in which case the two alignments defined by \texttt{align\_block}\ are aligned; \textit{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 \texttt{alignment\_compare\_with()}\ and \texttt{alignment\_write()}\ for evaluation of various alignment parameters and methods. The second exception is when \texttt{align\_what}\ is set to \texttt{‘LAST’}, in which case only the last sequences in the two blocks are used to get the scores. In \texttt{‘BLOCK’}, \texttt{‘ALIGNMENT’}, and \texttt{‘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.

For the time being, this and the other alignment commands \texttt{alignment.malign()}\, \texttt{alignment.align2d()}\ and \texttt{alignment.malign3d()}\ remove chain break information from the CALN\ array, which means that chain breaks are not retained when the alignment is written to a file after executing these commands.

Example: \texttt{examples/commands/align.py}

```python
# 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')
```

6.15.19 \texttt{alignment.align2d()} \textit{—} align sequences with structures

\texttt{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=(-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),
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=1, subopt\_offset=0.0, input\_profile\_file=None,
output\_profile\_file=None, weigh\_sequences=False, smooth\_prof\_weight=10, weights\_type='SIMILAR',
io=None)}

\textbf{IMPORTANT NOTE:} This command is obsolete, and is no longer maintained. It is strongly recommended that you use \texttt{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 \texttt{alignment.align()}\ command except that a variable gap \textbf{opening} penalty is used. This gap
6.15. THE ALIGNMENT CLASS: COMPARISON OF SEQUENCES AND STRUCTURES 143

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 Cα 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 \( \omega_H, \omega_S, \omega_B, \omega_C, \omega_D, \delta_\alpha, \gamma, t \) 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 \( \beta \)-strand positions, two consecutive buried positions, or two consecutive positions where the mainchain is locally straight. This function is 

\[ f_1 = 1 + \omega_H H_i H_{i+1} + \omega_S S_i S_{i+1} + \omega_B B_i B_{i+1} + \omega_C C_i C_{i+1} + \omega_S C S C_i S_{i+1} \]

\( H_i \) is the fraction of helical residues at position \( i \) in block 1, \( S_i \) is the fraction of \( \beta \)-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 \( S_{C_i} \) is the structural conservedness at position \( i \) in block 1. See Section 6.6.31 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 \( \beta \)-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 know 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 \( \beta \)-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 

\[ f_2 = 1 + [\omega_H H_i + \omega_S S_i + \omega_B B_i + \omega_C C_i + \omega_D \sqrt{d - d_\alpha} + \omega_S C S C_i] \]

\( d \) is the distance between the two Cα atoms spanning the gap, averaged over all structures in block 1 and \( d_\alpha \) is the distance that is small enough to correspond to no increase in the opening gap penalty (e.g., 8.6 Å).

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.

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)
mdl2.write(file='2nbt.fit.pdb')

6.15.20 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.salign() 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.salign() for more details.
Example: `examples/commands/malign.py`

```python
# 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')
```

6.15.21 `alignment.consensus()` — consensus sequence alignment

```python
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`

```python
# 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')
```

6.15.22 `alignment.compare_structures()` — compare 3D structures given alignment

```python
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,
```
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 ± 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, \( \alpha \), \( \Phi \), \( \Psi \), \( \chi_1 \), \( \chi_2 \), \( \chi_3 \), \( \chi_4 \), and \( \chi_5 \) (there are 5 dihedrals in a disulfide bridge), where \( \alpha \) is the virtual \( C_\alpha \) dihedral angle between four consecutive \( C_\alpha \) 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.

### 6.15.23 `alignment.align3d()` — align two structures

```python
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.
6.15. THE ALIGNMENT CLASS: COMPARISON OF SEQUENCES AND STRUCTURES

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

```python
# 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:
mdl1 = model(env, file='1fas')
atmsel = selection(mdl1).only_atom_types('CA')
mdl2 = model(env, file='2ctx')
atmsel.superpose(mdl2, aln)
```
6.15.24  alignment.malign3d() — align two or more structures

alignment.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', writewholepdb=True, current_directory=True, write_fit=False, editfile_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.atomfile` 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 `editfile_ext[0]` is removed (if present), and then the extension in `editfile_ext[1]` is added (if not already present). By default this creates files with a `.fit` extension.
6.15. THE ALIGNMENT CLASS: COMPARISON OF SEQUENCES AND STRUCTURES

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.

If write_whole_pdb is True, the whole PDB files are written out; 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

```python
# 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 3.5Å/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))
```

6.15.25 alignment.salign() — align two or more sequences/structures of proteins

```python
alignment.salign(residue_type2='REGULAR', no_term=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,
0.0), rms_cutoff=3.5, fit=True, surf_type=1, fit_on_first=False, gap_function=False,
align_block=0, max_gap_length=99999999, align_what='BLOCK', input_weights_file=None,
output_weights_file=None, weigh_sequences=False, smooth_prof_weight=0.0, fix_offsets=(0.0,
-1.0, -2.0, -3.0, -4.0), substitution=False, comparison_type='MAT', matrix_comparision='CC',
alignment_type='PROGRESSIVE', edit_file_ext=('pdb', '_fit.pdb'), weights_type='SIMILAR',
similarity_flag=False, bkgrnd_prlty_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=1, subopt_offset=0.0, align3d_trf=False,
normalize_scores=False, gap_gap_score=0.0, gap_residue_score=0.0, nsegm=2,
matrix_offset_3d=-0.1, io=None)
```

Output: SalignData object

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 [Sali & 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.

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() or an environment dependent gap penalty function (as used in alignment.align2d()). The choice of gap penalty types can be regulated by switching the variable gap_function on or off. 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.qscore pct: 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. \( W_{ij}^1 \) is obtained from the residue type – residue type dissimilarity matrix, specified in the file rr_file. \( W_{ij}^1 \) 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.

**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\(_\alpha\), although we should also allow for C\(_\beta\) 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. \( W_{ij}^2 \) 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 \texttt{gap\_penalties\_3d} (a reasonable setting is \((0, 3)\)). \texttt{gap\_penalties\_3d[0]} is a gap creation penalty (usually 0), and \texttt{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 \texttt{gap\_penalties\_3d[0]} angstroms apart in the current superposition, as described for the \texttt{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 \texttt{improve\_alignment} and \texttt{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 \texttt{write\_fit = True}. If \texttt{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 \texttt{feature\_weight[2] > 0} if you wish to write out the structures superposed according to the tree-following procedure; also, \texttt{fit\_on\_first} must be \texttt{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 \texttt{alignment\_salign()} If you wish such a superposition, you can use \texttt{alignment\_malign3d()} with \texttt{fit = False} and \texttt{write\_fit = True} (the meaning of \texttt{fit} is different between \texttt{alignment\_salign()} and \texttt{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 \texttt{alignment\_malign()} or by \texttt{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 \texttt{fit\_on\_first} is \texttt{True}, all the coordinate sets are superposed on the first structure, using the final multi-feature multiple alignment. If \texttt{fit\_on\_first} is \texttt{False}, and position feature was used, and \texttt{fit} was \texttt{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 \texttt{improve\_alignment} is \texttt{False}, it does not make much sense to have \texttt{fit = True} (use \texttt{fit\_on\_first = True}).

For local alignments, the matrix offset variable is \texttt{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, 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 (\texttt{fit\_atoms}) from the segments of \((2^\text{nsegm} + 1)\) 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 \texttt{modlib} directory for examples) has to be specified using \texttt{input\_weights\_file}. The user can input either a similarity matrix (\texttt{weights\_type = SIMILAR}) or a distance matrix (\texttt{weights\_type = DISTANCE}).
Alignment of protein sequences

- **Alignment of two sequence blocks**
  
  For all alignments that are to be performed using only sequence information all features but the first should be made 0. Pairs of sequence blocks are aligned using SALIGN the same way in which `alignment.align()` aligned sequence blocks. The example in the next section illustrates how SALIGN accomplishes this. `gap_function` should be kept `False` to align sequence blocks locally or globally using affine gap penalties.

- **Alignment of protein sequences by their profiles**
  
  Two blocks of sequences can be aligned using the information contained within each of the multiple sequence blocks [Martí-Renom *et al.*, 2004]. `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. Also, since this kind of alignment is effected only between two blocks, `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 the 20 amino acids at each position (`alignment_type` is set to `PSSM`).
  2. Comparing the residue substitution matrices implied at each position of the two blocks (`alignment_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`).

  **Example:** `examples/salign/salign_profile_profile.py`

  ```python
  # 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
    similarity_flag=True,  # The score matrix is not rescaled
    substitution=True,  # The BLOSUM62 substitution values are
    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',
    alignment_codes=('12asA', '1b8aA'))
  aln.write(file='salign_pair.ali', alignment_format='PIR')
  aln.write(file='salign_pair.pap', alignment_format='PAP')
  ```

- **Multiple Protein sequence alignment**
  
  Multiple alignments of sequences is similar to multiply aligning structures, the difference being that for
sequence alignments only the first feature is non-zero. The example in the structure alignment section can illustrate multiple sequence alignments as well.

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 = (4.35, 1.2, 0.9, 1.2, 0.6, 8.6, 1.2, 0., 0.) (default)
# 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., 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
#write_weights=True, 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 not been optimized for distance matrices.

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.

• 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 \(\text{gap\_function}\) is set to \text{False} or \text{True} (arguments to the \text{alignment.align()} and \text{alignment.align2d()} commands apply).

- **Progressive Multiple Structure Alignments**
  If \(\text{alignment\_type}\) is set to \text{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, \text{auto\_overhang} can be set to \text{True}. This will ensure that the value of \text{overhang} changes as \text{overhang\_factor} times the numerical difference in the residues of the pair. Further, this is only effected if the difference is greater than \text{overhang\_auto\_limit}.

The dendrogram can be written out in a separate file by specifying the file name to \text{dendrogram\_file}.

**Example:** examples/salign/salign_multiple.py

```python
# 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 'progressive', 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 output suboptimal alignments are written into the file ‘suboptimal_alignments.out’ and is of the format that specifies the serial residue numbers of the two sequences.

Example: examples/salign/salign_subopt.py

```python
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)
```

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

```python
# 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](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 penalties for the ext. matrix
        #feature_weights=(1., 0., 0., 0., 0., 0.), gap_penalties_1d=(30, 26),
        write_weights=True, 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.
6.15. **THE ALIGNMENT CLASS: COMPARISON OF SEQUENCES AND STRUCTURES**

**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 \[ g_{i,j} = \frac{n_{rg}}{(n_1 n_2)} r + \frac{n_{rg}}{(n_1 n_2)} g, \] 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.

**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 \( W_{i,j} = \sum_f \frac{\omega_f}{\omega_f} W_{f,i,j} + g_{i,j} \), 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 \( \omega_f \) (specified in `feature_weights`) is non-zero. The matrices \( W \) 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.

If `write_whole_pdb` is `True`, the whole PDB files are written out; 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.

### 6.15.26 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

```python
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')
```

### 6.15.27 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    KETAAKFERQHMDSSTSAASSSNYCNQMMKSRNLTKRCKPVWTFTVHESLADVQAVCSQKNVAC-KN
edn     ---KPPQFTWAQWFETQHMDSSTSAASSSNYCNQMMKSRNLTKRCKPNTFLTTAFANVNVQCNPHMTCP
 templ    -HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
_consrvd *          *          *          *          *          ****        ****          *          *          *          *          *

_aln.p 70 80 90 100 110 120 130
7rsa   -GQTNCYQSYSTMSITDCRETGSS--KYPNCAKYTTQANKHIVACEGN---------PYYPVHF
edn    KTRKNCNCHSSQSQFPLTHCNLTPFPQINSRNYAQTAPNMVYACNDPRDRDPPQYPVPV
 templ   GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
_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_[_i_−1] to _segment_shift_[_i_]. 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_[_i_−1] to _segment_growth_n_[_i_]. 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_[_i_−1] to _segment_growth_c_[_i_]. 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_idnnnn0000file_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`

```python
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')
```
6.16 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 6.15).

For alignment template structures (i.e., sequences for which a structure is also available) see the Structure class in Section 6.17.

Example: examples/python/alignsequence.py

```python
# 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" \n    % (seq.code, seq.source, len(seq))
```

6.16.1 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().

6.16.2 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.

6.16.3 Sequence.atom_file — PDB file name

This gives the name of the PDB file containing the associated 3D structure for the sequence, if available.

6.16.4 Sequence.source — source organism

This gives the name of the organism from which the sequence was obtained, if available.

6.16.5 Sequence.name — protein name

This gives the full name of the protein, if available.

6.16.6 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.

6.16.7 Sequence.resolution — structure resolution

The resolution of the associated X-ray structure, or -1.0 if unknown or not applicable.
6.16.8 **Sequence.rfactor** — R factor

The R factor of the associated X-ray structure, or -1.0 if unknown or not applicable.

6.16.9 **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_#:CHAIN_ID`, where 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 '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 6.19 for more information about Residue objects. See also `model.residue_range()` for getting a contiguous range of residues in a model.

6.16.10 **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 6.18 for more information about Chain objects.

Example: See `selection.assess_dope()` command.

6.16.11 **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.

6.16.12 **Sequence.get_num_equiv()** — get number of equivalences

`get_num_equiv(seq)`

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.

6.16.13 **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.
6.17 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 6.16 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.

6.17.1 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).

6.17.2 Structure.reread() — reread coordinates from 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 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.

Example: examples/python/chains.py

```python
# 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: ", [(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)
```

6.18.1 Chain.name — chain ID

This is the name (ID) of the chain, traditionally a single letter.

6.18.2 Chain.residues — all residues in the chain

This is a list of all residues in the chain, as Residue objects (see Section 6.19).

6.18.3 Chain.atoms — all atoms in the chain

This is a list of all atoms in the chain, as Atom objects (see Section 6.22). (Not available for alignment sequences.)

6.18.4 Chain.filter() — check if this chain passes all criteria

```python
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

```python
# 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 *

dl = 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)
```

6.18.5 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.

6.18.6 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. **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`).

**Example:** [examples/python/residues.py](#)

```python
# 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': ", mdl.residues['10:'].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:', '10:'))
analyze_seq("Aligned sequence %s" % firstseq.code, firstseq.residues)
```

### 6.19.1 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.

### 6.19.2 Residue.pdb_name — PDB (IUPAC) type name

This is the 3-letter name of the residue, as used in PDB.
6.19.3 Residue.code — One-letter residue type code

This is the one-letter residue type code, as used in alignment files.

6.19.4 Residue.hetatm — HETATM indicator

If True, this residue is marked in PDB as a HETATM residue rather than an ATOM residue.

6.19.5 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

```python
# This demonstrates relating PDB residue numbers with residue indices.
from modeller import *
log.verbose()
env = environ()
environ.atom_files_directory = ['../atom_files']
mdl = model(env, file='2abx')

print "Mapping from residue indices to PDB residue and chainnames:",
for r in mdl.residues:
    print "%6d %3s:%s %s" % (r.index, r.num, r.chain.name, r.pdb_name)
```

6.19.6 Residue.atoms — all atoms in the residue

This is a list of all atoms in the residue, as Atom objects (see Section 6.22). (Not available for alignment sequences.)

6.19.7 Residue.chain — chain object

This is the Chain object to which the residue belongs. See Section 6.18

6.19.8 Residue.phi — \( \phi \) dihedral angle

This is a Dihedral object, with information about the residue’s \( \phi \) dihedral. (If no \( \phi \) dihedral is defined for this residue, it is the special Python value None instead.) See Section 6.20 for more information on Dihedral objects. Not available for alignment sequence residues.

6.19.9 Residue.psi — \( \psi \) dihedral angle

This is a Dihedral object, with information about the residue’s \( \psi \) dihedral; see Residue.phi for more information. Not available for alignment sequence residues.

6.19.10 Residue.omega — \( \omega \) dihedral angle

This is a Dihedral object, with information about the residue’s \( \omega \) dihedral; see Residue.phi for more information. Not available for alignment sequence residues.
6.19. **THE RESIDUE CLASS: A SINGLE RESIDUE IN A MODEL OR ALIGNMENT**

6.19.11 **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 Cα atoms, starting with the previous residue); see `Residue.phi` for more information. Not available for alignment sequence residues.

6.19.12 **Residue.chi1 — χ1 dihedral angle**

This is a Dihedral object, with information about the residue’s χ1 dihedral; see `Residue.phi` for more information. Not available for alignment sequence residues.

6.19.13 **Residue.chi2 — χ2 dihedral angle**

This is a Dihedral object, with information about the residue’s χ2 dihedral; see `Residue.phi` for more information. Not available for alignment sequence residues.

6.19.14 **Residue.chi3 — χ3 dihedral angle**

This is a Dihedral object, with information about the residue’s χ3 dihedral; see `Residue.phi` for more information. Not available for alignment sequence residues.

6.19.15 **Residue.chi4 — χ4 dihedral angle**

This is a Dihedral object, with information about the residue’s χ4 dihedral; see `Residue.phi` for more information. Not available for alignment sequence residues.

6.19.16 **Residue.chi5 — χ5 dihedral angle**

This is a Dihedral object, with information about the residue’s χ5 dihedral; see `Residue.phi` for more information. Not available for alignment sequence residues.

6.19.17 **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.

6.19.18 **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.

6.19.19 **Residue.add_trailing_gaps() — add gap(s) after this residue**

`add_trailing_gaps(ngap=1)`
This adds $\text{ngap}$ 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.

6.19.20  Residue.remove_leading_gaps() — remove gap(s) before this residue

$\text{remove\_leading\_gaps(ngap=1)}$

This removes $\text{ngap}$ gaps from the alignment, immediately before this residue. Not available for model sequences.

6.19.21  Residue.remove_trailing_gaps() — remove gap(s) after this residue

$\text{remove\_trailing\_gaps(ngap=1)}$

This removes $\text{ngap}$ 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.

6.19.22  Residue.get_leading_gaps() — get number of gaps before this residue

$\text{get\_leading\_gaps()}$

This returns the number of gaps in the alignment immediately preceding this residue. Not available for model sequences.

6.19.23  Residue.get_trailing_gaps() — get number of gaps after this residue

$\text{get\_trailing\_gaps()}$

This returns the number of gaps in the alignment immediately following this residue. Not available for model sequences.
6.20 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.ph] for more information on accessing these objects.

6.20.1 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.

6.20.2 Dihedral.atoms — atoms defining the angle

This is the list of the four atoms which define the dihedral angle.

6.20.3 Dihedral.dihclass — integer dihedral class

This is the current value of the dihedral class, as defined in 'modlib/resdih.lib'.
6.21 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()`.  

6.21.1 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.

6.21.2 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.
6.22 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 6.21.

Example: `examples/python/atoms.py`

```python
# Example for 'atom' objects
rom modeller import *
from modeller.scripts import complete_pdb

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

mdl = complete_pdb(e, '1fas')

# 'mdl.atoms' is a list of all atoms in the model
print "Name of C-alpha atom in residue 4: ", mdl.atoms['CA:4'].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'].atoms[0]
print "Biso for first atom in residue 10: %.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
```

6.22.1 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 dyv and dvz. (Only available for models, not template structures.)

6.22.2 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.)

6.22.3 Atom.biso — isotropic temperature factor

This is the isotropic temperature factor ($B_{iso}$), which can be set by `selection.energy()` or `model.write_data()`.

6.22.4 Atom.occ — occupancy

This is the crystallographic occupancy of the atom.
6.22.5 Atom.charge — electrostatic charge

This is the electrostatic charge of the atom. (Only available for models, not template structures.)

6.22.6 Atom.mass — mass

This is the mass of the atom. (Only available for models, not template structures.)

6.22.7 Atom.name — PDB name

This is the name used to refer to the atom in PDB.

6.22.8 Atom.residue — residue object

This returns the Residue object to which this atom belongs. See Section 6.19.
6.23 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()`.

6.23.1 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.

6.23.2 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).

Example: `examples/commands/read_profile.py`

```python
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')
```

6.23.3 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.

Example: `examples/commands/alu_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')

6.23.4 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')
6.23.5 profile.scan() — Compare a target profile against a database of profiles

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 \texttt{profile.read()} command. All the profiles listed in \texttt{profile_list_file} should be in a format that is understood by \texttt{profile.read()}. The \texttt{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 \texttt{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 \texttt{profile.read()} for help on \texttt{profile.format}.

\texttt{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.

\texttt{gap_penalties_1d} are the gap penalties to use for the dynamic programming. \texttt{matrix_offset} is the value to be used to offset the substitution matrix (used in PSSM calculation). \texttt{ccmatrix_offset} is used to offset the scoring matrix during dynamic programing. The most optimal values for these parameters are:

- \texttt{matrix_offset} = -450 (for the BLOSUM62 matrix)
- \texttt{ccmatrix_offset} = -100
- \texttt{gap_penalties_1d} = (-700, -70)

\texttt{max_aln_evalue} sets the threshold for the E-values. Alignments with e-values better than the threshold will be written out.

\texttt{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

\texttt{score_statistics} is a flag that triggers the calculation of e-values. If set to \texttt{False}, the significance estimates for the alignments will not be calculated. The calculation of alignment significance is similar to that used for \texttt{profile.build()}. This option can be useful when there are only a very small number of template profiles in \texttt{profile_list_file}, insufficient to calculate reliable statistics. Also see \texttt{profile.build()}

\texttt{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 \texttt{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

\texttt{summary_file} is the name of a file into which to output a summary of all the significant alignments. (If it is set to \texttt{None}, no such output is written.) The format of the summary file is the following:

1. File name of target profile
2. Length of target profile
CHAPTER 6. MODELLER COMMAND REFERENCE

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
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

```python
# 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',
```
6.23. THE PROFILE CLASS: USING SEQUENCE PROFILES

```python
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')
```

6.23.6 profile.build() — Build a profile for a given sequence or alignment

```python
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)
```

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 'on', 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).

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 (or may even fail) if the database does not have sequences that represent the entire range of lengths possible.

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.

Example: `examples/commands/build_profile.py`

```python
from modeller import *
log.verbose()
env = environ()

#-- Prepare the input files

#-- Read in the sequence database
sdb = sequence_db(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')
```
6.23.7  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.

6.23.8  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.
6.24 The sequence_db class: using sequence databases

The sequence_db 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 sequence_db.search() used to build sequence profiles with profile.build() or filtered by given criteria with sequence_db.filter().

6.24.1 sequence_db() — create a new sequence database

sequence_db(env, **vars)

This creates a new, empty, sequence database. If you give any arguments to this constructor, they are passed to sequence_db.read() to read in an initial database. See the sequence_db.filter() example.

6.24.2 sequence_db.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 seq_database_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 sequence_db.filter().

None of the options above apply to the BINARY format, which, in return, is very fast (i.e., 3 seconds for 300 MB of 800,000 sequences in the TrEMBL database). Binary files are standard HDF5 files (see Section B.4).

Note that 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).

If you are intending to read in a sequence database simply to write it out again in binary format, you should consider using the sequence_db.convert() function instead, as this does not need to keep the whole database in memory.

Example: See profile.build() command.
6.24.3 sequence_db.write() — write a database of sequences

```
write(chains_list, seq_database_file, seq_database_format)
```

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.

Example: See `profile.build()` command.

6.24.4 sequence_db.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 `sequence_db.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 `sequence_db.read()` followed by `sequence_db.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

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

sdb = sequence_db(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')
```

6.24.5 sequence_db.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, 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 `sequence_db.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 array. These sequences are taken from seq_database_file, which is often (but not always) the same file previously provided to sequence_db.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.

Example: examples/commands/sequence_search.py

# Example for: sequence_db.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 = sequence_db(env, seq_database_file='pdball.pir',
# Read in the sequences of all PDB structures
try:
sdb = sequence_db(env, seq_database_file='pdball.pir',
6.24. THE SEQUENCE_DB CLASS: USING SEQUENCE DATABASES

```python
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 (http://salilab.org/modeller/supplemental.html).

Note: it is recommended to use profile.build() rather than sequence_db.search().
See step 1 of the Modeller basic tutorial at
http://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')

6.24.6 sequence_db.filter() — cluster sequences by sequence-identity

```python
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)
```

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 sequence_db.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 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 $\text{max}_\text{unaligned}_\text{res}$.

3. Sequence Identity: Finally, if the sequence identity calculated from the alignment is greater than $\text{seqid}_\text{cut}$, 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.

Example: `examples/commands/seqfilter.py`

```python
from modeller import *

log.verbose()
env = environ()

sdb = sequence_db(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')
```
6.25 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 \texttt{density\_grid\_search()}) to improve their quality.

6.25.1 density() — create a new density map

density(env, **vars)

This creates a new, empty, density map. If you give any arguments to the \texttt{density()} constructor, they are passed to \texttt{density\_read()} to read in an initial density. See the \texttt{density\_grid\_search()} example.

6.25.2 density.read() — read an EM (electron microscopy) density map file

\begin{verbatim}
read(file, em_map_size, voxel_size, resolution, em_density_format='XPLOR', filter_type='NONE', filter_values=(0.0, 0.0), density_type='SPHERE', px=0, py=0, pz=0, ccf_func_type='CCF')
\end{verbatim}

This command reads a density map from file, which should be provided as a cubic grid of intensities, in the X-PLOR ([Bru
gger, 1992]) or MRC format. The size of the cubic grid is given by \texttt{em\_map\_size}. The density map resolution and the map voxel size are given by \texttt{resolution} and \texttt{voxel\_size}, respectively.

When fitting the probe into the EM grid, the probe structure is converted first into probe density, by using the function indicated in the \texttt{density\_type} variable. Each atom can be represented by one of several atomic density functions, including, the uniform sphere model (\texttt{SPHERE}), the Gaussian function (\texttt{GAUSS}), a normalized Gaussian function (\texttt{NORM}), a hybrid Gaussian/sphere model (\texttt{HYBRID}), and an interpolation to the closest point on the grid (\texttt{TRACE}). The recommended function is \texttt{SPHERE}.

When calculating the cross-correlation coefficient between a probe model and the density map, \texttt{ccf\_func\_type} specifies if you want the normalized cross-correlation coefficient (\texttt{CCF}), or the local cross-correlation coefficient (\texttt{LCCF}).

Example: See \texttt{density\_grid\_search()} command.

6.25.3 density.grid_search() — dock a structure into an EM (electron microscopy) density map

\begin{verbatim}
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')
\end{verbatim}

Requirements: PDB files

This command docks a structure of a protein/domain (probe) into a given EM density map. See original paper for the description of the method and the most recommended protocols [Topf et al., 2005].

The probe is specified by the variable \texttt{em\_pdb\_name}. Before starting the protocol, the probe is positioned on the EM density grid based on the \texttt{start\_type} variable:

- \texttt{CENTER} will translate the center of mass of the probe to the center of mass of the grid.
- \texttt{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.
- \texttt{SPECIFIC} will not perform any translation and will just 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 \( (\phi, \theta, \psi) \) 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 \( (i.e., 3 \times 3 \times 3 - 1) \) 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 'CENTER', this protocol can be used for a translational and rotational refinement of the initial superposition of the centers of mass. 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:**

```python
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')
6.26 The saxsdata class: using small-angle X-ray (SAXS) data

The saxsdata class is used to store small-angle X-ray (SAXS) data.

6.26.1 saxsdata() — create a new saxsdata structure

saxsdata(env, **vars)

This creates a new saxsdata object. Use saxsdata.ini() to initialize it.

6.26.2 saxsdata.ini() — Initialization of SAXS data

ini.saxs(atmsel, filename='$\text{LIB}/formfactors-intertab.solvation.lib', s_min=0.0, s_max=2.0, maxs=100, nmesh=100, natomtyp=15, represtype='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. represtype 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\(^{-}\text{Å}^{-3}\). (The default 0.334 corresponds to H\(_2\)O.) 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.

6.26.3 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(). The file is text format, containing 3 columns: spatial frequency s in Å\(^{-1}\), Intensity, and experimental error (if determined). Comments start with '#'.

6.26.4 saxsdata.saxs_pr_read() — Read in P(r) data

saxs_pr_read(filename)
6.26. **THE SAXSDATA CLASS: USING SMALL-ANGLE X-RAY (SAXS) DATA**

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()`
6.27 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.

6.27.1 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.

6.27.2 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.)

6.27.3 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.

6.27.4 info.exe_type — the executable type of this binary

This is a string identifying the machine and FORTRAN compiler type of this MODELLER binary.

6.27.5 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.

6.27.6 info.bindir — MODELLER binary directory

This is the directory in which MODELLER binaries are installed.

6.27.7 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.

6.27.8 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.
6.28 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.)

6.28.1 log.level() — Set all log output levels

```python
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.

6.28.2 log.none() — display no log output

```python
none()
```

This instructs MODELLER to display no log output.

6.28.3 log.minimal() — display minimal log output

```python
minimal()
```

This instructs MODELLER to only display important outputs, and errors.

6.28.4 log.verbose() — display verbose log output

```python
verbose()
```

This instructs MODELLER to display all log output.

6.28.5 log.very_verbose() — display verbose log output, and dynamic memory information

```python
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.)
6.29 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.

6.29.1 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}'.

6.29.2 modfile.delete() — delete a file

delete(file)

This command deletes the named file.

6.29.3 modfile.inquire() — check if file exists

inquire(file)

Output: file_exists

This command returns 1 if the specified file exists, or 0 otherwise.

6.29.4 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(). 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.

6.30 The scripts module: utility scripts

The scripts module contains utility scripts for some common tasks.
6.31. PARALLEL JOB SUPPORT

6.30.1 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.

6.30.2 complete_pdb() — read a PDB file, and fill in any missing residues

complete_pdb(env, filename, special_patches=None, transfer_res_num=False, model_segment=None, patch_default=True)

Output: mdl

This reads in a PDB file specified by filename, and fills in any missing atoms using internal coordinates. If you want to read in a PDB 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 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 is retained (by default, residues are renumbered from 1).

If model_segment is set, it is used to read a subset of the residues from the PDB 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()" conjugate_gradients() command.

6.31 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 ‘slave’, while the process which starts and controls the parallel job is termed the ‘master’.

Two methods for accessing the slaves from the master are provided; the first is a task-based interface (job.queue_task() and job.run_all_tasks()) which is largely transparent in use and fault-tolerant, while the second is a lower-level slave-master message-passing interface (job.start() Communicator.send_data() Communicator.get_data() and slave.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 2.3 if it is not already present on your system. It is also experimental, and is not yet supported fully on Windows systems.

6.31.1 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 slave processes. It is initially empty, but acts just like an ordinary Python list, so you can add or remove slave objects (see below) using ordinary list operations (e.g., append, del). Also, if you provide a list of suitable slave objects to job(), they will automatically be added.

Each slave runs a MODELLER process. This will expect to find ‘mod9v4’ in your PATH; if this is not the case, specify the binary location and name with the modeller_path variable. (By specifying the ‘bin/modslave.py’ script you can start a slave which uses the system Python rather than that included in the MODELLER interpreter, which is useful if you’re using the system Python for your master process as well.)

Each slave, when started, tries to connect back over the network to the master node. By default, they try to use the fully qualified domain name of the machine on which you create the job object (the master). If this name is incorrect (e.g., on multi-homed hosts) then specify the true hostname with the host parameter.

Each slave will run in the same directory as the master, so will probably fail if you do not have a shared filesystem on all nodes. The output from each slave is written to a logfile called ‘${JOB}.slaveN’ where ‘${JOB}’ is the name of your master script file (or ‘stdout’ if you are reading from standard input) and ‘N’ is the number of the slave, starting from zero.

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().

To use the message-passing interface, first start all slaves with job.start() and then use Communicator.send_data(), Communicator.get_data() and slave.run_cmd() to pass messages and commands.

Example: See job.start() job.run_all_tasks() command.

6.31.2  sge_pe_job() — create a job using all Sun GridEngine (SGE) slave processes

sge_pe_job(seq=(), modeller_path=None, host=None)

This functions identically to job() above, but automatically adds slaves 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 sge_pe_slave object (see below) for each processor. Other slaves 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 master (first) node in an SGE parallel environment job.

6.31.3  sge_qsub_job() — create a job which can be expanded with Sun GridEngine ’qsub’

sge_qsub_job(options, maxslave, seq=(), modeller_path=None, host=None)

This functions identically to job() above, but it automatically grows by adding new sge_qsub_slave slaves (up to a maximum of maxslave) if you submit more tasks to the job than there are available slaves. (These are grouped into a single SGE array job.) options specifies options for the new sge_qsub_slave objects. New slaves are not automatically added when using the message-passing interface; you should manually add new sge_qsub_slave 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 slave nodes).

6.31.4  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()`.

The task should be an instance of a class derived from task, which provides a ‘run’ method. This method will be run on the slave node; any arguments to this method are given on the master when the object is created, and are automatically passed for you to the slave. Anything you return from this method is automatically passed back to the master. (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.

### 6.31.5 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 slave. The tasks are run in the same order they were submitted, and this function returns a list of all the return values from the tasks, in that order.

Tasks are run in a simple round-robin fashion on the available slaves. If a slave fails while running a task, that task is automatically resubmitted to another slave. If you submit more tasks than available slaves, new slaves are automatically added to the job if the job supports this functionality (e.g., `sge_qsub_job()`).

**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 slave, 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 *

# 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 slave processes running on the local machine
j = job()
j.append(local_slave())
j.append(local_slave())
```
# 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: ", results

### 6.31.6 job.start() — start all slaves for message-passing

This starts all non-running slaves 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 slaves are automatically started when required.)

Example: examples/python/parallel-msg.py

```python
from modeller import *
from modeller.parallel import *

j = job()
j.append(local_slave())

j.start()

for slave in j:
    slave.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)
    ''
    slave.send_data('1fdn')
    data = slave.get_data()
    print slave, "returned model resolution: ", data
```

### 6.31.7 Communicator.send_data() — send data

```
send_data(data)
```
6.31. PARALLEL JOB SUPPORT

This sends the given data to the communicator. For sending from the master to slaves, this communicator is simply the 'slave' object itself. For sending from the slaves back to the master, a communicator called 'master' is provided for you.

Any data that can be processed by the Python 'pickle' module can be sent to and from slaves. 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 not passed in these objects, so if, for example, you were to pass a model object, it would contain no atoms when it reached the slave. 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 master to send data to a slave using this function, unless the slave is already waiting for data (i.e., by itself calling Communicator.get_data()). Generally this means you should call slave.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 slave, for example. Any other errors (e.g., a syntax error in your script) will raise a RemoteError exception.

6.31.8 Communicator.get_data() — get data

get_data(allow_heartbeat=False)

This gets the next piece of data available from the slave or master. 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().

6.31.9 slave.run_cmd() — run a command on the slave

run_cmd(cmd)

This runs a command (or several commands, separated by line breaks) on the slave. (It is not possible for the slave to run commands on the master.) Errors are as for Communicator.send_data().

6.31.10 local_slave() — create a slave running on the local machine

local_slave()

This creates a new slave process, which will run on the same machine as the master. This is useful if the machine has multiple CPUs, or if the master process is going to be largely idle. It should be added to a job object to be useful.

6.31.11 sge_pe_slave() — create a slave running on a Sun GridEngine parallel environment slave node

sge_pe_slave(nodename)

This creates a new slave process, which runs on the slave node given by nodename as part of a Sun GridEngine (SGE) parallel job. The process is started using 'qsh -inherit -V', so your SGE setup should be correctly configured to allow this. Generally you would use an sge_pe_job object instead, above, to automatically create slave processes for all SGE nodes.
6.31.12  sge_qsub_slave() — create a 'qsub' slave running on a Sun GridEngine node

sge_qsub_slave(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 master script and acts as a new slave process. Generally you would use an sge_qsub_job object instead, above, to automatically create slave processes when required.

6.31.13  ssh_slave() — create a slave on a remote host accessed via ssh

ssh_slave(nodename, ssh_command='ssh')

This creates a new slave 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 Pyrex, use the Psyco optimizer, 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

```python
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)
sr = mdl.restraints
at = mdl.atoms
sr.add(forms.gaussian(group=physical.bond,
                       feature=MyDist(at['CA:1'], at['C:1'])),
       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.restraint_form. You should then override the following functions: __init__, eval, vmin, rvmin, min_mean, vheavy, rvheavy, and heavy_mean. 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 restraint_form.__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 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`

```python
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.restraint_form):
    """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.restraint_form.__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]

    # 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'], at['CA:1']),
                mean=1.5380, stdev=0.0364))
```
7.1.3 User-defined energy terms

To create a new energy term, derive a new class from the base `terms.energy_term`. 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 `energy_data.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`

```python
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.energy_term):
    """Custom energy term, which tries to force all atoms to one side of the x=10.0Å 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.energy_term.__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](https://www.python.org/) 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 $MODINSTALL9v4/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](https://www.python.org/) interface is also built from these header files, using the [SWIG](http://www.swig.org/) package. All of the files used to build this interface can be found in the $MODINSTALL9v4/src/swig directory. You can use these to build an interface for a different version of [Python](https://www.python.org/) see the 'README' file in this directory for instructions.

If you run `mod9v4 --cflags`, it will output 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.

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](https://www.python.org/) scripts, and can be much faster than implementing the code in [Python](https://www.python.org/). 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

```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
*/

,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_write_header();

    mod_log_set(2, 1);
    libs = mod_libraries_new(NULL);
    mod_libraries_read_libs(libs, "${LIB}/restyp.lib", &ierr);
    handle_error(ierr);
    mod_libraries_rand_seed_set(libs, -8123);
    mdl = mod_model_new(NULL);
    io = mod_io_data_new();
    mod_model_read(mdl, io, libs, "../atom_files/2nbt.pdb", "PDB",
                   "FIRST:@LAST: ", 7, &ierr);
    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);
}
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 [Sali & 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 [Sali & 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 $W$ 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 (i.e. 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 $D$ are:

$$ D_{i,j} = \min\left\{ \begin{array}{l}
  P_{i,j} \\
  D_{i-1,j-1} + W_{i,j} \\
  Q_{i,j}
\end{array} \right. $$

$$ P_{i,j} = \min\left\{ \begin{array}{l}
  D_{i-1,j} + g(1) \\
  P_{i-1,j} + v
\end{array} \right. $$

$$ Q_{i,j} = \min\left\{ \begin{array}{l}
  D_{i,j-1} + g(1) \\
  Q_{i,j-1} + v
\end{array} \right. $$

where $g(l)$ is a linear gap penalty function:

$$ g(l) = u + v \cdot l. $$

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 $D$, $P$ and $Q$ are initialized as follows:

$$ D_{i,0} = \begin{cases}
  0, & i \leq e \\
  g(i - e), & e < i \leq N
\end{cases} $$

$$ D_{0,j} = \begin{cases}
  0, & j \leq e \\
  g(j - e), & e < j \leq N
\end{cases} $$

$$ P_{i,0} = \begin{cases}
  \infty, & i = 1, 2, \ldots, M
\end{cases} $$

$$ P_{0,j} = \begin{cases}
  \infty, & j = 1, 2, \ldots, N
\end{cases} $$

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 $P$ and $Q$ 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

$$ d_{M,N} = \min(D_{i,N}, D_{M,j}) $$

A.1.2 Variable gap penalty

This work is still in progress and is not described here.

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.2. OPTIMIZATION OF THE OBJECTIVE FUNCTION BY MODELLER

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, i.e. 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 CHARM [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):

\[
F = F(R) = F_{\text{symm}} + \sum_i c_i(f_i, p_i)
\]  

(A.5)

where \(F_{\text{symm}}\) is an optional symmetry term defined in Eq. [A.98] \(R\) are Cartesian coordinates of all atoms, \(c\) is a restraint \(i\), \(f\) is a geometric feature of a molecule, and \(p\) 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 \(Å^2\), 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 \(\Phi\) and \(\Psi\). Each of the restraints also depends on a few parameters \(p_i\) 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]

\[
F = F_{\text{symm}} + F_s + F_d
\]  

(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 beneficial 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., \texttt{selection.energy()}, \texttt{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.

\section*{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 & Go, 1985] which is implemented with the \texttt{automodel} class (Section A.4). The molecular dynamics procedure can be used in a simulated annealing protocol that is also implemented with the \texttt{automodel} class.

\subsection*{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\textsuperscript{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]:

$$\dot{r}_i \left( t + \frac{\delta t}{2} \right) = \dot{r}_i \left( t - \frac{\delta t}{2} \right) - \frac{\partial F}{\partial r_i(t)} \frac{\delta t}{m_i}$$

$$r_i(t + \delta t) = r_i(t) + \dot{r}_i \left( t + \frac{\delta t}{2} \right) \delta t$$

where $r_i$ 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 capped at a maximum value. The velocities can be equilibrated every equilibrate steps to stabilize temperature. This is achieved by scaling the velocities with a factor $f$:

$$f = \sqrt{T/E_{kin}}$$

$$E_{kin} = \frac{1}{2} \sum_i m_i \dot{r}_i^2$$

where $E_{kin}$ is the current kinetic energy of the system.

Langevin dynamics

Langevin dynamics (LD) are implemented as in [Loncharich et al., 1992]. The equations of motion (Equation A.9) are modified as follows:

$$\dot{r}_i \left( t + \frac{\delta t}{2} \right) = \dot{r}_i \left( t - \frac{\delta t}{2} \right) \left( 1 - \frac{1}{2} \gamma \delta t \right) + \left( R_i - \frac{\partial F}{\partial r_i(t)} \right) \frac{\delta t}{m_i} \left( 1 + \frac{1}{2} \gamma \delta t \right)$$

where $\gamma$ is a friction factor (in $fs^{-1}$) and $R_i$ a random force, chosen to have zero mean and standard deviation

$$\sigma(R_i) = \sqrt{\frac{2 \gamma m_i k_B T}{\delta t}}$$

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:

$$g_i(t) = \left( 1 - \frac{\delta t}{t_i} \right) g_i(t - \delta t) + \frac{\delta t}{t_i} \left( \lambda g_i(t - \delta t) - \frac{\partial F}{\partial r_i(t)} \right)$$

$$\dot{r}_i \left( t + \frac{\delta t}{2} \right) = \dot{r}_i \left( t - \frac{\delta t}{2} \right) + \left( \lambda g_i(t) - \frac{\partial F}{\partial r_i(t)} \right) \frac{\delta t}{m_i}$$

where $\lambda$ is the guiding factor (the same for all atoms), $t_i$ the guide time in femtoseconds, and $g_i$ a guiding force, set to zero at the start of the simulation. (Position $r_i$ 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.13):

$$g_i(t) = \left( 1 - \frac{\delta t}{t_i} \right) g_i(t - \delta t) + \frac{\delta t}{t_i} \gamma m_i \dot{r}_i \left( t - \frac{\delta t}{2} \right)$$

A scaling parameter $\chi$ is then determined by first making an unconstrained half step:

$$\dot{r}_i'(t) = \dot{r}_i \left( t - \frac{\delta t}{2} \right) + \frac{1}{2} \left( \lambda g_i(t) + R_i - \frac{\delta F}{\partial r_i(t)} \right) \frac{\delta t}{m_i}$$

$$\zeta = \left( 1 + \frac{\gamma \delta t}{2} \right) \frac{\sum_i N \lambda g_i(t) r_i'(t)}{\sum_i m_i r_i'^2(t)}$$

$$\chi = \left( 1 + \frac{(\gamma + \zeta) \delta t}{2} \right)^{-1}$$
Finally, the velocities are advanced using the scaling factor:

\[ \dot{r}_i \left( t + \frac{\delta t}{2} \right) = (2\chi - 1)\dot{r}_i \left( t - \frac{\delta t}{2} \right) + \left( \lambda g_i(t) + R_i - \frac{\partial F}{\partial r_i(t)} \right) \frac{\delta t}{m_i} \]  
(A.21)

**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, \( r_{COM} \), and an orientation quaternion, \( \hat{q} \) \cite{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:

\[ \frac{\partial F}{\partial r_{COM}} = \sum_i \frac{\partial F}{\partial r_i} \]  
(A.22)

where the sum \( i \) operates over all atoms in the rigid body, and \( r_i \) 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 \cite{Rapaport, 1997}:

\[ \ddot{q} = \frac{1}{2} W^T \begin{pmatrix} \dot{\omega}'_x \\ \dot{\omega}'_y \\ \dot{\omega}'_z \\ -2 \sum_m q^2_m \end{pmatrix} \]  
(A.23)

where \( W \) is the orthogonal matrix

\[ W = \begin{pmatrix} q_4 & q_3 & -q_2 & -q_1 \\ -q_3 & q_4 & q_1 & -q_2 \\ q_2 & -q_1 & q_3 & -q_4 \\ q_1 & q_2 & q_3 & q_4 \end{pmatrix} \]  
(A.24)

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:

\[ \dot{\omega}'_x = \frac{T_x + (I_y - I_z)\omega'_z \omega'_y}{I_x} \]  
(A.25)

(Similar equations exist for the \( y \) and \( z \) components.) The angular velocities \( \omega' \) are obtained from the quaternion velocities:

\[ \begin{pmatrix} \omega'_x \\ \omega'_y \\ \omega'_z \\ 0 \end{pmatrix} = 2 W \dot{q} \]  
(A.26)

The torque, \( T \), in the body-fixed frame, is calculated as

\[ T = A \sum_i (r_i - r_{COM}) \times -\frac{\partial F}{\partial r_i} \]  
(A.27)

and \( A \) is the rotation matrix to convert from world space to body space

\[ A = 2 \begin{pmatrix} q_1^2 + q_4^2 - \frac{1}{2} & q_1 q_2 + q_3 q_4 & q_1 q_3 - q_2 q_4 \\ q_1 q_2 - q_3 q_4 & q_2^2 + q_4^2 - \frac{1}{2} & q_2 q_3 + q_1 q_4 \\ q_1 q_3 + q_2 q_4 & q_2 q_3 - q_1 q_4 & q_3^2 + q_4^2 - \frac{1}{2} \end{pmatrix} \]  
(A.28)
and finally the $x$ component of the inertia tensor, $I_x$, is given by

$$I_x = \sum_i m_i (r_i'^2 + r_i'^2)$$

(A.29)

where $r_i'$ 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:

$$E_{kin}^{body} = \frac{1}{2} \left( \sum_i m_i \dot{r}_{COM}^2 + \frac{1}{2} (I_x \omega_x'^2 + I_y \omega_y'^2 + I_z \omega_z'^2) \right)$$

(A.30)

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, $r_{COM}$, and the rotations in radians about the body-fixed axes: $\theta_x$, $\theta_y$, and $\theta_z$. The first derivative of the objective function $F$ with respect to the center of mass is obtained from Equation [A.22] and those with respect to the angles from:

$$\frac{\partial F}{\partial \theta_k} = M_k r_i' \cdot \frac{\partial F}{\partial r_i}$$

(A.31)

The transformation matrices $M_k$ are given as:

$$M_k = \begin{bmatrix} 0 & -\sin \theta_z \sin \theta_x - \cos \theta_z \sin \theta_y \cos \theta_x & \sin \theta_z \cos \theta_x - \cos \theta_z \sin \theta_y \sin \theta_x \\ 0 & -\cos \theta_z \sin \theta_x + \sin \theta_z \sin \theta_y \cos \theta_x & \cos \theta_z \cos \theta_x + \sin \theta_z \sin \theta_y \sin \theta_x \\ 0 & -\cos \theta_y \cos \theta_x & -\cos \theta_y \sin \theta_x \end{bmatrix}$$

(A.32)

$$M_y = \begin{bmatrix} -\cos \theta_z \sin \theta_y & -\cos \theta_z \cos \theta_y \sin \theta_x & \cos \theta_z \cos \theta_y \cos \theta_x \\ \sin \theta_z \sin \theta_y & \sin \theta_z \cos \theta_y \sin \theta_x & -\sin \theta_z \cos \theta_y \cos \theta_x \\ -\sin \theta_y & \sin \theta_x \sin \theta_y & -\sin \theta_x \cos \theta_y \end{bmatrix}$$

(A.33)

$$M_z = \begin{bmatrix} -\sin \theta_z \cos \theta_y & \cos \theta_z \cos \theta_y + \sin \theta_z \sin \theta_y \sin \theta_x & \cos \theta_z \sin \theta_x - \sin \theta_z \sin \theta_y \cos \theta_x \\ -\cos \theta_z \cos \theta_y & -\sin \theta_z \cos \theta_y + \sin \theta_z \sin \theta_y \sin \theta_x & \cos \theta_z \sin \theta_x - \sin \theta_z \sin \theta_y \cos \theta_x \\ 0 & \cos \theta_z \sin \theta_x & \cos \theta_z \cos \theta_x \end{bmatrix}$$

(A.34)

The atomic positions $r_i$ are reconstructed when necessary from the body’s orientation by means of the following relation:

$$r_i = M r_i'^{COM}$$

(A.35)

where $M$ is the rotation matrix

$$M = \begin{bmatrix} \cos \theta_z \cos \theta_y & \sin \theta_z \cos \theta_x - \cos \theta_z \sin \theta_y \sin \theta_x & \sin \theta_z \sin \theta_x + \cos \theta_z \sin \theta_y \cos \theta_x \\ -\sin \theta_z \cos \theta_y & \cos \theta_z \cos \theta_x + \sin \theta_z \sin \theta_y \sin \theta_x & \cos \theta_z \sin \theta_x - \sin \theta_z \sin \theta_y \cos \theta_x \\ -\sin \theta_y & \cos \theta_z \sin \theta_x & \cos \theta_y \cos \theta_x \end{bmatrix}$$

(A.36)

**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 $j$:

$$d = \sqrt{r_{ij} \cdot r_{ij}} = |r_{ij}| = r_{ij}$$

(A.37)
where
\[ r_{ij} = r_i - r_j . \] (A.38)

The first derivatives of \( d \) with respect to Cartesian coordinates are:
\[ \frac{\partial d}{\partial r_i} = \frac{r_{ij}}{|r_{ij}|} \] (A.39)
\[ \frac{\partial d}{\partial r_j} = \frac{\partial d}{\partial r_i} \] (A.40)

**Angle**

Angle is defined by points \( i, j, \) and \( k \), and spanned by vectors \( ij \) and \( kj \):
\[ \alpha = \arccos \frac{r_{ij} \cdot r_{kj}}{|r_{ij}||r_{kj}|} . \] (A.41)

It lies in the interval from 0 to 180°. Internal MODELLER units are radians.

The first derivatives of \( \alpha \) with respect to Cartesian coordinates are:
\[ \frac{\partial \alpha}{\partial r_i} = \frac{\partial \alpha}{\partial \cos \alpha} \frac{\partial \cos \alpha}{\partial r_i} = \frac{1}{\sqrt{1 - \cos^2 \alpha}} \frac{1}{r_{ij}} \left( \frac{r_{ij} \cos \alpha - r_{kj}}{r_{ij}} \right) \] (A.42)
\[ \frac{\partial \alpha}{\partial r_k} = \frac{\partial \alpha}{\partial \cos \alpha} \frac{\partial \cos \alpha}{\partial r_k} = \frac{1}{\sqrt{1 - \cos^2 \alpha}} \frac{1}{r_{kj}} \left( \frac{r_{kj} \cos \alpha - r_{ij}}{r_{ij}} \right) \] (A.43)
\[ \frac{\partial \alpha}{\partial r_j} = -\frac{\partial \alpha}{\partial r_i} - \frac{\partial \alpha}{\partial r_k} \] (A.44)

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 \( l \) (\( ijkl \)):
\[ \chi = \text{sign}(\chi) \arccos \left( \frac{(r_{ij} \times r_{kj}) \cdot (r_{kj} \times r_{kl})}{|r_{ij} \times r_{kj}| |r_{kj} \times r_{kl}|} \right) \] (A.45)

where
\[ \text{sign}(\chi) = \text{sign} \left[ r_{kj} \cdot (r_{ij} \times r_{kj}) \times (r_{kj} \times r_{kl}) \right] . \] (A.46)

The first derivatives of \( \chi \) with respect to Cartesian coordinates are:
\[ \frac{d\chi}{dr} = \frac{d\chi}{d\cos \chi} \frac{d\cos \chi}{dr} \] (A.47)
where
\[ \frac{d\chi}{d\cos \chi} = \left( \frac{d\cos \chi}{d\chi} \right)^{-1} = \frac{1}{\sin \chi} \] (A.48)
A.3. EQUATIONS USED IN THE DERIVATION OF THE MOLECULAR PDF

and

\[
\frac{\partial \cos \chi}{\partial r_i} = r_{kj} \times a \tag{A.49}
\]

\[
\frac{\partial \cos \chi}{\partial r_j} = r_{ik} \times a - r_{kl} \times b \tag{A.50}
\]

\[
\frac{\partial \cos \chi}{\partial r_k} = r_{jl} \times b - r_{ij} \times a \tag{A.51}
\]

\[
\frac{\partial \cos \chi}{\partial r_l} = r_{ij} \times b \tag{A.52}
\]

\[
a = \frac{1}{|r_{ij} \times r_{kj}|} \left( \frac{r_{kj} \times r_{kl}}{|r_{kj} \times r_{kl}|} - \cos \chi \frac{r_{ij} \times r_{kj}}{|r_{ij} \times r_{kj}|} \right) \tag{A.53}
\]

\[
b = \frac{1}{|r_{kj} \times r_{kl}|} \left( \frac{r_{ij} \times r_{kj}}{|r_{ij} \times r_{kj}|} - \cos \chi \frac{r_{kj} \times r_{kl}}{|r_{kj} \times r_{kl}|} \right). \tag{A.54}
\]

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]:

\[
r_{mj} = r_{ij} \times r_{kj} \tag{A.55}
\]

\[
r_{nk} = r_{kj} \times r_{kl} \tag{A.56}
\]

\[
\frac{\partial \chi}{\partial r_i} = \frac{r_{kj}}{v_{mj}^2} r_{mj} \tag{A.57}
\]

\[
\frac{\partial \chi}{\partial r_l} = -\frac{r_{kj}}{v_{nk}^2} r_{nk} \tag{A.58}
\]

\[
\frac{\partial \chi}{\partial r_j} = \left( \frac{r_{ij} \cdot r_{kj}}{v_{kj}^2} - 1 \right) \frac{\partial \chi}{\partial r_i} - \frac{r_{kl} \cdot r_{kj}}{v_{kj}^2} \frac{\partial \chi}{\partial r_l} \tag{A.59}
\]

\[
\frac{\partial \chi}{\partial r_k} = \left( \frac{r_{kl} \cdot r_{kj}}{v_{kj}^2} - 1 \right) \frac{\partial \chi}{\partial r_l} - \frac{r_{ij} \cdot r_{kj}}{v_{kj}^2} \frac{\partial \chi}{\partial r_i} \tag{A.60}
\]

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

xx

Atomic density

Atomic density for a given atom is simply calculated as the number of atoms within a distance \textbf{energy.data.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

\[
p = \frac{1}{\sigma \sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{f - \bar{f}}{\sigma} \right)^2 \right].
\]

(A.61)

A corresponding restraint \( c \) in the sum that defines the objective function \( F \) is

\[
c = -\ln p = \frac{1}{2} \left( \frac{f - \bar{f}}{\sigma} \right)^2 - \ln \frac{1}{\sigma \sqrt{2\pi}}
\]

(A.62)

(Note that since the second term is constant for a given restraint, it is ignored.)

The first derivatives with respect to feature \( f \) are:

\[
\frac{dc}{df} = \frac{f - \bar{f}}{\sigma} \cdot \frac{1}{\sigma}.
\]

(A.63)

The relative heavy violation with respect to \( f \) is given as:

\[
\frac{f - \bar{f}}{\sigma}
\]

(A.64)

**Multiple Gaussian restraint**

The polymodal pdf for a geometric feature \( f \) (e.g., distance, angle, dihedral angle) is

\[
p = \sum_{i=1}^{n} \omega_i p_i = \sum_{i=1}^{n} \omega_i \cdot \frac{1}{\sigma_i \sqrt{2\pi}} \exp \left[ -\frac{1}{2} \left( \frac{f - \bar{f}_i}{\sigma_i} \right)^2 \right].
\]

(A.65)

A corresponding restraint \( c \) in the sum that defines the objective function \( F \) is

\[
c = -\ln p = -\ln \sum_{i=1}^{n} \omega_i p_i
\]

(A.66)

The first derivatives with respect to feature \( f \) are:

\[
\frac{dc}{df} = \frac{1}{p} \sum_{i=1}^{n} \omega_i p_i \cdot \left[ \frac{f - \bar{f}_i}{\sigma_i} \right].
\]

(A.67)

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 \texttt{rgauss1} (10 for double precision). This scheme works up to \texttt{rgauss2} (200 for double precision); violations larger than that are ignored. This trick is equivalent to increasing the standard deviation \( \sigma_i \). A slight disadvantage is that there is a discontinuity in the first derivatives at \texttt{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).
A.3. EQUATIONS USED IN THE DERIVATION OF THE MOLECULAR PDF

\[ M = 37 \quad ; \quad M^2/2 \text{ has to be smaller than the largest argument to } \exp \] (A.68)

\[ A = \frac{\text{rgauss2} - M}{M \left( \text{rgauss2} - \text{rgauss1} \right)} \] (A.69)

\[ B = \frac{\text{rgauss2} - \text{rgauss1}}{M \left( \text{rgauss2} - \text{rgauss1} \right)} \] (A.70)

\[ v = \frac{f - \bar{f}_i}{\sigma_i} \] (A.71)

\[ F = A \left| v \right| + B \] (A.72)

\[ v' = \frac{v}{F} \] (A.73)

Now, Eqs. [A.65] [A.67] are used with \( v' \) instead of \( v \). For single precision, \( M = 12, \text{rgauss1} = 4, \text{rgauss2} = 100 \).

The relative heavy violation with respect to \( f \) is given as:

\[ \max_{\omega_i} (f - \bar{f}_i)/\sigma_i \] (A.74)

Multiple binormal restraint

The polymodal pdf for a geometric feature \((f_1, f_2)\) (e.g., a pair of dihedral angles) is

\[
p = \sum_{i=1}^{n} \omega_i p_i = \sum_{i=1}^{n} \omega_i \frac{1}{2\pi \sigma_{1i} \sigma_{2i} \sqrt{(1 - \rho_i^2)}} \cdot 
\exp \left\{ -\frac{1}{2(1 - \rho_i^2)} \left[ \left( \frac{f_1 - \bar{f}_{1i}}{\sigma_{1i}} \right)^2 - 2 \rho_i \frac{f_1 - \bar{f}_{1i}}{\sigma_{1i}} \frac{f_2 - \bar{f}_{2i}}{\sigma_{2i}} + \left( \frac{f_2 - \bar{f}_{2i}}{\sigma_{2i}} \right)^2 \right] \right\} . \] (A.75)

where \( \rho < 1 \). \( \rho \) is the correlation coefficient between \( f_1 \) and \( f_2 \). MODELLER actually uses the following series expansion to calculate \( p \):

\[
p = \sum_{i=1}^{n} \frac{1}{2\pi \sigma_{1i} \sigma_{2i} \sqrt{(1 - \rho_i^2)}} \cdot 
\exp \left\{ -\frac{1}{1 - \rho_i^2} \left[ 1 - \cos(f_1 - \bar{f}_{1i}) - \rho_i \frac{\sin(f_1 - \bar{f}_{1i}) \sin(f_2 - \bar{f}_{2i})}{\sigma_{1i}} + \frac{1 - \cos(f_2 - \bar{f}_{2i})}{\sigma_{2i}} \right] \right\} . \] (A.76)

A corresponding restraint \( c \) in the sum that defines the objective function \( F \) is

\[ c = -\ln p = -\ln \sum_{i=1}^{n} \omega_i p_i \] (A.77)

The first derivatives with respect to features \( f_1 \) and \( f_2 \) are:

\[ \frac{\partial c}{\partial f_1} = \frac{1}{p} \sum_{i=1}^{n} \omega_i p_i \cdot \frac{1}{\sigma_{1i}(1 - \rho_i^2)} \left( \frac{\sin(f_1 - \bar{f}_{1i})}{\sigma_{1i}} - \rho_i \frac{\cos(f_1 - \bar{f}_{1i}) \sin(f_2 - \bar{f}_{2i})}{\sigma_{2i}} \right) \] (A.78)

\[ \frac{\partial c}{\partial f_2} = \frac{1}{p} \sum_{i=1}^{n} \omega_i p_i \cdot \frac{1}{\sigma_{2i}(1 - \rho_i^2)} \left( \frac{\sin(f_2 - \bar{f}_{2i})}{\sigma_{2i}} - \rho_i \frac{\cos(f_2 - \bar{f}_{2i}) \sin(f_1 - \bar{f}_{1i})}{\sigma_{1i}} \right) \] (A.79)

The relative heavy violation with respect to \( f \) is given as:

\[ \max_{\omega_i} \sqrt{\frac{1}{2(1 - \rho_i^2)} \left[ \left( \frac{f_1 - \bar{f}_{1i}}{\sigma_{1i}} \right)^2 - 2 \rho_i \frac{f_1 - \bar{f}_{1i}}{\sigma_{1i}} \frac{f_2 - \bar{f}_{2i}}{\sigma_{2i}} + \left( \frac{f_2 - \bar{f}_{2i}}{\sigma_{2i}} \right)^2 \right] \] (A.80)
Lower bound

This is like the left half of a single Gaussian restraint:

\[
p = \begin{cases} 
p_{\text{gauss}} : & f < \tilde{f} \\
0 : & f \geq \tilde{f}
\end{cases}
\]  

(A.81)

where \( \tilde{f} \) is a lower bound and \( p_{\text{gauss}} \) is given in Eq. [A.61]. 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:

\[
p = \begin{cases} 
p_{\text{gauss}} : & f > \bar{f} \\
0 : & f \leq \bar{f}
\end{cases}
\]  

(A.82)

where \( \bar{f} \) is an upper bound and \( p_{\text{gauss}} \) is given in Eq. [A.61]. 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 \( f \):

\[
c = |b| - b \cos(nf + a)
\]  

(A.83)

where \( b \) is CHARMN 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 CHARMN phase value from the CHARMN 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.

\[
\frac{dc}{df} = bn \sin(nf + a)
\]  

(A.84)

Coulomb restraint

\[
c = \frac{1}{\epsilon_r} \frac{q_i q_j}{f} s(f, f_1, f_2)
\]  

(A.85)

\[
s(f, f_1, f_2) = \begin{cases} 
1 ; & f \leq f_1 \\
\frac{(f_2 - f)^2(f_2 + 2f_1 - 3f)}{(f_2 - f_1)^3} ; & f_1 < f \leq f_2 \\
0 ; & f > f_2
\end{cases}
\]  

(A.86)

where \( q_i \) and \( q_j \) are the atomic charges of atoms \( i \) and \( j \), obtained from the CHARMN topology file, that are at a distance \( f \). \( \epsilon_r \) is the relative dielectric, controlled by the [energy_data_relative_dielectric] variable. Function \( s(f, f_1, f_2) \) is a switching function that smooths 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:

\[
\frac{dc}{df} = -\frac{c}{f} + \frac{c}{s} \frac{ds}{df}
\]  

(A.87)

\[
\frac{ds}{df} = \begin{cases} 
0 ; & f \leq f_1 \\
\frac{6(f_2 - f)(f_1 - f)}{(f_2 - f_1)^3} ; & f_1 < f \leq f_2 \\
0 ; & f > f_2
\end{cases}
\]  

(A.88)

The violations of this restraint are always reported as zero.
A.3. EQUATIONS USED IN THE DERIVATION OF THE MOLECULAR PDF

Lennard-Jones restraint

Usually used for non-bonded distances:

\[ c = \left( \frac{A}{f} \right)^{12} - \left( \frac{B}{f} \right)^{6} \]

s(f, f_1, f_2) \quad (A.89)

The parameters \( f_1 \) and \( f_2 \) of the switching function can be different from those in Eq. (A.86). 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 \( E_{ij}(f) = -4\sqrt{E_i E_j} [(\rho_{ij}/f)^{12} - (\rho_{ij}/f)^{6}] \) in kcal/mole for \( f \) in angstroms and \( \rho = (r_i + r_j)/2^{1/6} \); the minimum of \( E \) is \(-\sqrt{E_i E_j} \) 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:

\[ \frac{dc}{df} = \frac{Cs}{f} - C \frac{ds}{df} \quad (A.90) \]

\[ C = -12 \left( \frac{A}{f} \right)^{12} + 6 \left( \frac{B}{f} \right)^{6} \quad (A.91) \]

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]:

\[ c = Ac_j + Bc_{j+1} + Cc''_j + Dc''_{j+1} \quad (A.92) \]

\[ A = \frac{f_{j+1} - f}{f_{j+1} - f_j} \quad (A.93) \]

\[ B = 1 - A \quad (A.94) \]

\[ C = \frac{1}{6}(A^3 - A)(f_{j+1} - f_j)^2 \quad (A.95) \]

\[ D = \frac{1}{6}(B^3 - B)(f_{j+1} - f_j)^2 \quad (A.96) \]

where \( f_j \leq f \leq f_{j+1} \).

The first derivatives are:

\[ \frac{dc}{df} = \frac{c_{j+1} - c_j}{f_{j+1} - f_j} - \frac{3A^2 - 1}{6} (f_{j+1} - f_j) c''_j - \frac{3B^2 - 1}{6} (f_{j+1} - f_j) c''_{j+1} \quad (A.97) \]

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 \( \sigma \). These are then used in Eq. (A.64).
Symmetry restraint

The asymmetry penalty added to the objective function is defined as

\[ F_{\text{symm}} = \sum_{i<j} \omega_i \omega_j (d_{ij} - d'_{ij})^2 \]  

(A.98)

where the sum runs over all pairs of equivalent atoms \(ij\), \(\omega_i\) 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:

\[ \frac{\partial c}{\partial d_{ij}} = 2 \omega_i \omega_j \frac{d_{ij} - d'_{ij}}{d_{ij}} \]  

(A.99)

\[ \frac{\partial c}{\partial d'_{ij}} = -2 \omega_i \omega_j \frac{d_{ij} - d'_{ij}}{d'_{ij}} \]  

(A.100)

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
   \[ \text{alignment.append()} \] and \[ \text{alignment.check()} \].

2. Calculate restraints on the target from its alignment with the templates:
   (a) Generate molecular topology for the target sequence \[ \text{model.generate_toplogy()} \]. Disulfides in the target are assigned here from the equivalent disulfides in the templates \[ \text{model.patch_ss_templates()} \]. Any user defined patches are also done here (as defined in the \[ \text{automodel.special_patches()} \] routine).
   (b) Calculate coordinates for atoms that have equivalent atoms in the templates as an average over all templates \[ \text{model.transfer_xyz()} \] (alternatively, read the initial coordinates from a file).
   (c) Build the remaining unknown coordinates using internal coordinates from the CHARM topology library \[ \text{model.build()} \].
   (d) Write the initial model to a file with extension \[ .ini \] \[ \text{model.write()} \].
   (e) Generate stereochemical, homology-derived, and special restraints \[ \text{Restraints.make()} \] (alternatively, skip this and assume the restraints file already exists):
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 mainchain carbonyl oxygen and amide nitrogen atoms of the N- and C-terminal anchor regions respectively, 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 ±5Å 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
AFVVTDICVCKYTDCCVCPDVCFYEQPNFLVIHPDECIDCALCEPECQAIFSEDEVVPEDQEFILMAELA
EVWPNITEKDPLPDAEDVDCGKGLQHLER*

>P1;1fdx
sequence:1fdx:1 : :54 : :ferredoxin;Peptococcus aerogenes: 2.00:-1.00
AYVINDSC--IACGACKPECPVNIIQS--IYADADSCIDCVGASVCPGAPNED-----------------
-----------------------------------------------------------------

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.)

The second line of each entry contains information necessary to extract atomic coordinates of the segment from the original PDB 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 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 PDB code in this field, which is used to get structural data, does not have to be unique. It is a good idea to use the PDB code with an optional chain identifier as the protein code. The PDB code corresponds to the Sequence.atom file variable and can also contain the full atom filename, directory included.

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, and a chain identifier the 1-letter PDB chain code. 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 '@:0' 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 or even missing entirely. 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

```
structure:pdb_file:......
```

This entry will result in reading from PDB file pdb_file the structure segment corresponding to the sequence in the subsequent lines of the alignment entry.

The fields that do not exist are assigned blank values. Thus,

```
structure:pdb_file
```
B.2 RESTRAINTS FILE

is equivalent to


which will achieve what was probably intended (read in the structure segment from file pdb_file that corresponds to the sequence in the subsequent lines of the alignment entry) only if the chain id is a blank character.

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 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, 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.
APPENDIX B. FILE FORMATS

<table>
<thead>
<tr>
<th>Numeric form</th>
<th>Form type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>forms.lower_bound</td>
</tr>
<tr>
<td>2</td>
<td>forms.upper_bound</td>
</tr>
<tr>
<td>3</td>
<td>forms.gaussian</td>
</tr>
<tr>
<td>4</td>
<td>forms.multi_gaussian</td>
</tr>
<tr>
<td>5</td>
<td>forms.lennard_jones</td>
</tr>
<tr>
<td>6</td>
<td>forms.coulomb</td>
</tr>
<tr>
<td>7</td>
<td>forms.cosine</td>
</tr>
<tr>
<td>8</td>
<td>forms.factor</td>
</tr>
<tr>
<td>9</td>
<td>forms.multi_binormal</td>
</tr>
<tr>
<td>10</td>
<td>forms.spline or forms.nd_spline</td>
</tr>
<tr>
<td>50+</td>
<td>user-defined restraint forms</td>
</tr>
</tbody>
</table>

Table B.1: Numerical restraint forms

<table>
<thead>
<tr>
<th>Numeric feature</th>
<th>Feature type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>features.distance</td>
</tr>
<tr>
<td>2</td>
<td>features.angle</td>
</tr>
<tr>
<td>3</td>
<td>features.dihedral</td>
</tr>
<tr>
<td>6</td>
<td>features.minimal_distance</td>
</tr>
<tr>
<td>7</td>
<td>features.solvent_access</td>
</tr>
<tr>
<td>8</td>
<td>features.density</td>
</tr>
<tr>
<td>9</td>
<td>features.x_coordinate</td>
</tr>
<tr>
<td>10</td>
<td>features.y_coordinate</td>
</tr>
<tr>
<td>11</td>
<td>features.z_coordinate</td>
</tr>
<tr>
<td>12</td>
<td>features.dihedral_diff</td>
</tr>
<tr>
<td>50+</td>
<td>user-defined feature types</td>
</tr>
</tbody>
</table>

Table B.2: Numerical feature types

B.2.2 Excluded pairs

An 'E' line should look like:

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.

<table>
<thead>
<tr>
<th>Numeric pseudo atom type</th>
<th>Pseudo atom type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pseudo_atom.gravity_center</td>
</tr>
<tr>
<td>2</td>
<td>virtual_atom.ch1</td>
</tr>
<tr>
<td>3</td>
<td>virtual_atom.ch1a</td>
</tr>
<tr>
<td>4</td>
<td>pseudo_atom.ch2</td>
</tr>
<tr>
<td>5</td>
<td>virtual_atom.ch2</td>
</tr>
<tr>
<td>6</td>
<td>pseudo_atom.ch31</td>
</tr>
<tr>
<td>7</td>
<td>pseudo_atom.ch32</td>
</tr>
</tbody>
</table>

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
```

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 `sequence_db.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 2.3 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:
APPENDIX C. CONVERTING TOP SCRIPTS FROM OLD MODELLER VERSIONS

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:

ev = 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).
### C.2. CONVERTING TOP SCRIPTS TO PYTHON

<table>
<thead>
<tr>
<th>Top command</th>
<th>Python equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_RESTR AI NT</td>
<td>Restraints.add()</td>
</tr>
<tr>
<td>ALIGN</td>
<td>alignment.align()</td>
</tr>
<tr>
<td>ALIGN2D</td>
<td>alignment.align2d()</td>
</tr>
<tr>
<td>ALIGN3D</td>
<td>alignment.align3d()</td>
</tr>
<tr>
<td>ALIGN_CONSENSUS</td>
<td>alignment.consensus()</td>
</tr>
<tr>
<td>ALN_TO_PROF</td>
<td>alignment.to_profile()</td>
</tr>
<tr>
<td>BUILD_MODEL</td>
<td>model.build()</td>
</tr>
<tr>
<td>BUILD_PROFILE</td>
<td>profile.build()</td>
</tr>
<tr>
<td>CALL</td>
<td>use Python subroutines</td>
</tr>
<tr>
<td>CHECK_ALIGNMENT</td>
<td>alignment.check()</td>
</tr>
<tr>
<td>CLOSE</td>
<td>use Python file object</td>
</tr>
<tr>
<td>COLOR_ALN_MODEL</td>
<td>model.color()</td>
</tr>
<tr>
<td>COMPARE</td>
<td>alignment.compare_structures()</td>
</tr>
<tr>
<td>COMPARE_ALIGNMENTS</td>
<td>alignment.compare_with()</td>
</tr>
<tr>
<td>CONDENSE_RESTR AI NT S</td>
<td>Restraints.condense()</td>
</tr>
<tr>
<td>DEBUG_FUNCTION</td>
<td>selection.debug_function()</td>
</tr>
<tr>
<td>DEFINE_INTEGER</td>
<td>use Python 'int' variables</td>
</tr>
<tr>
<td>DEFINE_LOGICAL</td>
<td>use Python 'bool' variables</td>
</tr>
<tr>
<td>DEFINE_REAL</td>
<td>use Python 'float' variables</td>
</tr>
<tr>
<td>DEFINE_STRING</td>
<td>use Python 'str' variables</td>
</tr>
<tr>
<td>DEFINE_SYMMETRY</td>
<td>Restraints.add()</td>
</tr>
<tr>
<td>DELETE_ALIGNMENT</td>
<td>del(alignment)</td>
</tr>
<tr>
<td>DELETE_FILE</td>
<td>modfile.delete()</td>
</tr>
<tr>
<td>DELETE_RESTR AI NT</td>
<td>Restraints.unpick()</td>
</tr>
<tr>
<td>DENDROGRAM</td>
<td>environ.dendrogram()</td>
</tr>
<tr>
<td>DESCRIBE</td>
<td>alignment.describe()</td>
</tr>
<tr>
<td>DO</td>
<td>use Python while or for loops</td>
</tr>
<tr>
<td>EDIT_ALIGNMENT</td>
<td>alignment.edit()</td>
</tr>
<tr>
<td>EM_GRID_SEARCH</td>
<td>density.grid_search()</td>
</tr>
<tr>
<td>END_SUBROUTINE</td>
<td>use Python subroutines</td>
</tr>
<tr>
<td>ENERGY</td>
<td>selection.energy()</td>
</tr>
<tr>
<td>EXIT</td>
<td>use Python while or for loops</td>
</tr>
<tr>
<td>EXPAND_ALIGNMENT</td>
<td>use alignment.append_model() in a loop</td>
</tr>
<tr>
<td>GENERATE_TOPOLOGY</td>
<td>model.generate_topology()</td>
</tr>
<tr>
<td>ID_TABLE</td>
<td>alignment.id_table()</td>
</tr>
<tr>
<td>IF</td>
<td>use Python if statement</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>use Python import statement</td>
</tr>
<tr>
<td>INQUIRE</td>
<td>modfile.inquire()</td>
</tr>
<tr>
<td>IUPAC_MODEL</td>
<td>model.to_iupac()</td>
</tr>
<tr>
<td>MAKE CHAINS</td>
<td>model.make_chains()</td>
</tr>
<tr>
<td>MAKE_REGION</td>
<td>model.make_region()</td>
</tr>
<tr>
<td>MAKE_RESTR AI NT S</td>
<td>Restraints.make()</td>
</tr>
<tr>
<td>MAKE_SCHEDULE</td>
<td>schedule.make_for_model()</td>
</tr>
<tr>
<td>MAKE_TOPOLOGY_MODEL</td>
<td>Topology.make()</td>
</tr>
<tr>
<td>MALIGN</td>
<td>alignment.malign()</td>
</tr>
<tr>
<td>MALIGN3D</td>
<td>alignment.malign3d()</td>
</tr>
<tr>
<td>MUTATE_MODEL</td>
<td>selection.mutate()</td>
</tr>
<tr>
<td>OPEN</td>
<td>use Python file object</td>
</tr>
<tr>
<td>OPERATE</td>
<td>use Python arithmetic</td>
</tr>
<tr>
<td>OPTIMIZE</td>
<td>conjugate_gradients() or molecular_dynamics() or &lt;schedule&gt; objects</td>
</tr>
<tr>
<td>ORIENT_MODEL</td>
<td>model.orient()</td>
</tr>
<tr>
<td>PATCH</td>
<td>model.patch()</td>
</tr>
<tr>
<td>PATCH SS_MODEL</td>
<td>model.patch_ss()</td>
</tr>
<tr>
<td>PATCH_SSTEMPLATES</td>
<td>model.patch_ss_templates()</td>
</tr>
</tbody>
</table>
 appendix c. converting top scripts from old modeller versions

pick_atoms
pick_hot_atoms
pick_restraints
principal_components
profile_profile_scan
prof_to_aln
randomize_xyz
read
read_alignment
read_alignment2
read_atom_classes
read_model
read_model2
read_parameters
read_profile
read_restraints
read Restyp_Lib
read_schedule
read_sequence_db
read_topo
read_index_restraints
rename_segments
reorder_atoms
reset
return
rotate_dihedrals
rotate_model
align
segment_matching
seqfilter
sequence_comparision
sequence_search
sequence_to_al
set
spline_restraints
stop
string_if
string_operate
subsection
superspose
switch_trace
system
time_mark
transfer_res_num
transfer_xyz
unbuild_model
write
write_alignment
write_data
write_model
write_model2
write_pdb_xref
write_profile
write_restraints
write_schedule
write_sequence_db

pick atoms
pick <selection> objects

use Python file object

use Python subroutines
### Table C.1: Correspondence between Top and Python commands.

<table>
<thead>
<tr>
<th>Top variable</th>
<th>Python equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN_CODES</td>
<td><code>Sequence.code</code></td>
</tr>
<tr>
<td>ATOM_FILES</td>
<td><code>Sequence.atom_file</code></td>
</tr>
<tr>
<td>ATOM_FILES_DIRECTORY</td>
<td><code>io_data.atom_files_directory</code></td>
</tr>
<tr>
<td>CONTACT_SHELL</td>
<td><code>energy_data.contact_shell</code></td>
</tr>
<tr>
<td>COULOMB_SWITCH</td>
<td><code>energy_data.coulomb_switch</code></td>
</tr>
<tr>
<td>COVALENT_CYS</td>
<td><code>energy_data.covalent_cys</code></td>
</tr>
<tr>
<td>DYNAMIC_ACCESS</td>
<td>do not use</td>
</tr>
<tr>
<td>DYNAMIC_COULOMB</td>
<td><code>energy_data.dynamic_coulomb</code></td>
</tr>
<tr>
<td>DYNAMIC_LENNARD</td>
<td><code>energy_data.dynamic_lennard</code></td>
</tr>
<tr>
<td>DYNAMIC_MODELLER</td>
<td><code>energy_data.dynamic_modeeller</code></td>
</tr>
<tr>
<td>DYNAMIC_PAIRS</td>
<td>set automatically; do not use</td>
</tr>
<tr>
<td>DYNAMIC_SPHERE</td>
<td><code>energy_data.dynamic_sphere</code></td>
</tr>
<tr>
<td>EXCL_LOCAL</td>
<td><code>energy_data.excl_local</code></td>
</tr>
<tr>
<td>HETATM_IO</td>
<td><code>io_data.hetatm</code></td>
</tr>
<tr>
<td>HYDROGEN_IO</td>
<td><code>io_data.hydrogen</code></td>
</tr>
<tr>
<td>LENNARD_JONES_SWITCH</td>
<td><code>energy_data.lennard_jones_switch</code></td>
</tr>
<tr>
<td>MOLPDF</td>
<td>return value from <code>selection.energy()</code></td>
</tr>
<tr>
<td>NLOGN_USE</td>
<td><code>energy_data.nlogn_use</code></td>
</tr>
<tr>
<td>NONBONDED_SEL_ATOMS</td>
<td><code>energy_data.nonbonded_sel_atoms</code></td>
</tr>
<tr>
<td>NUMB_OF SEQUENCES</td>
<td><code>len(alignment)</code></td>
</tr>
<tr>
<td>N_SCHEDULE</td>
<td><code>len(schedule)</code></td>
</tr>
<tr>
<td>OUTPUT_CONTROL</td>
<td>use <code>log.level()</code></td>
</tr>
<tr>
<td>RADI_FACtor</td>
<td><code>energy_data.radii_factor</code></td>
</tr>
<tr>
<td>RELATIVE_DIELECTRIC</td>
<td><code>energy_data.relative_dielectric</code></td>
</tr>
<tr>
<td>SCHEDULE_STEP</td>
<td>do not use</td>
</tr>
<tr>
<td>SPHERE_STDV</td>
<td><code>energy_data.sphere_stdv</code></td>
</tr>
<tr>
<td>TOPOLOGY_MODEL</td>
<td><code>Topology.submodel</code></td>
</tr>
<tr>
<td>UPDATE_DYNAMIC</td>
<td><code>energy_data.update_dynamic</code></td>
</tr>
<tr>
<td>WATER_IO</td>
<td><code>io_data.water</code></td>
</tr>
</tbody>
</table>

### Table C.2: Correspondence between Top and Python variables.
APPENDIX C. CONVERTING TOP SCRIPTS FROM OLD MODELLER VERSIONS
Bibliography


**Index**

A, 49
accessibility_type, 82, 89
actions, 119 121 122
actions.charmm_trajectory(), 39 73 122
actions.trace(), 39 119 122 236
actions.write_structure(), 119 121 122
add, 93 101
add_leading_gaps, 169
add.trailing_gaps, 169
align, 141
align2d, 142
align3d, 146
align3d_repeat, 147
align3d_trf, 147 150
align.alignment, 138
align_block, 138 141 142 144 152 159
align_code, 166 166
align_codes, 132 136
align what, 142 152
alignment, 58
alignment(), 132
alignment_class, 132
alignment files
  format, 225
  reading, 132
  writing, 137
alignment.Structure class, 163
alignment(), 132
alignment.align(), 35 138 141 142 144 146 150 152 154 235
alignment.align2d(), 46 138 142 143 150 153 154
alignment.align3d(), 107 142 146 148 150 151 235
alignment.append(), 34 132 133 139 138 139 179 222 236
alignment.append_model(), 34 35 105 136 223
alignment.append_profile(), 37 236
alignment.append_sequence(), 71 137
alignment.check(), 9 22 32 34 135 155 163 222 235
alignment.check_sequence_structure(), 34 135
alignment.check_structure_structure(), 34 134 135
alignment.clear(), 133
alignment.comments, 132
alignment.compare_sequences(), 58 140 230
alignment.compare_structures(), 46 58 143 146 235
alignment.compare_with(), 34 136 142 235
alignment.consensus(), 35 138 145 235
alignment.describe(), 139 235
alignment.edit(), 138 235
alignment.id_table(), 68 59 140 141 235
alignment.malign(), 142 144 148 150 151 154 235
alignment.malign3d(), 16 77 142 146 148 150 151 235
alignment.read_one(), 34 133 194
alignment.salign(), 34 141 142 144 146 148 151 152 154 158 157 236
alignment.segment_matching(), 158 236
alignment.to_profile, 59
alignment.to_profile(), 158 173 176 179 230 235
alignment.write(), 34 105 137 142 143 146 148
alignment_features, 138
alignment_format, 138
alignment_type, 152 154
allhmodel, 42
allhmodel class, 41
allhmodel(), 42
aln, 40 70 74 77 78 83 88 89 107 133 181
aln_base_filename, 177
alnfile, 37 12
alnseq, 72
alpha, 97
angular_step_size, 188
append, 65 96 129 132
append_model, 136
append_profile, 137
append_sequence, 137
asgl_output, 112 146
assess_dope, 115
assess_dopehr, 116
assess_ga341, 85
assess_methods, 14 37
atmsel, 87 89 91
atmsel1, 90
atmsel2, 90
Atom class, 173
atom classes, 224
Atom.biso, 173
Atom.charge, 174
features.minimal_distance, 48
features.solvent_access, 48
features.x_coordinate, 49
features.y_coordinate, 49
features.z_coordinate, 49
fh, 128
File, 194
file, 65, 66, 72, 81, 130, 133, 159, 163, 165, 187
file naming, 45
file types, 46
filename, 190, 195
filepattern, 122
filter, 164, 185
first, 122
fit, 107, 108, 138, 143, 146, 147, 149, 151, 158
fit_atoms, 146, 148, 150, 151
fit_on_first, 151, 158
fit_pdbnam, 151
fix_offsets, 155
form, 95
format, 165
forms.cosine, 49
forms.coulomb, 49
forms.factor, 49
forms.gaussian, 49
forms.lennard_jones, 49
forms.lower_bound, 49
forms.multi_bilnormal, 49
forms.multi_gaussian, 49
forms.nd_spline, 50
forms.spline, 50
forms.upper_bound, 49
frequently asked questions, 27

get_optimize_actions, 39
generate_topology, 72
gap_function, 150, 152, 154
gap_gap_score, 157
gap_penalties_1d, 143, 144, 150
get_aligned_residue, 169
get_data, 199
get_deletions, 70
get_insertions, 70
get_leading_gaps, 170
get_loop_model_filename, 48
get_model_filename, 41
get_num_equiv, 162

get_refine_actions, 39
get_sequence_identity, 162
get_trailing_gaps, 170
grid_search, 187
group, 95

high, 59
highderiv, 59
host, 196
hot_atoms, 111

id1, 194
id2, 194
id_table, 140
improve_alignment, 151, 158

input_profile_file, 143
input_weights_file, 151

inquire, 194
install, 5
intersegment, 88, 89
io, 57, 58, 63, 133, 139
io_data, 64
io_data_class, 64
io_data(), 67, 64
io_data.atom_files_directory, 12, 44, 46, 48, 237
io_data.hetatm, 67, 64, 237
io_data.hydrogen, 64, 237
io_data.water, 64, 237

job, 195
job_class, 195
job(), 195, 196
job.queue_task(), 195, 196
job.run_all_tasks(), 195, 197
job.start(), 195, 196, 198

knowns, 15, 37, 42, 134
INDEX

Restraints.unpick.all(), 51
Restraints.unpick_redundant(), 59, 92, 93
Restraints.write(), 48, 95, 223, 236
restraints_filter, 35, 91
restyp_lib_file, 57
rewind_file, 54
rho_solv, 190
rigid bodies, 52
rigid_body, 52
rms_cutoff, 107, 108, 157
rms_cutoffs, 146
root_name, 159, 194
rotate_dihedrals, 104
rotate_mass_center, 104
rotate_origin, 104
rr_file, 59, 111, 149, 144, 150, 157, 159, 177, 183, 185, 230
run_all_tasks, 197
run_cmd, 199
s_hi, 190
s_hybrid, 190
s_low, 190
s_max, 190
s_min, 190
salign, 149
SalignData, 149, 150
saxs_chifun, 85
saxs_intens, 85
saxs_pr, 85
saxs_pr_read, 190
saxs_read, 190
saxsdata, 190
saxsdata_class, 190
saxsdata(), 190
saxsdata.ini_saxs(), 190
saxsdata.saxs_pr_read(), 190
saxsdata.saxs_read(), 190
scale_factor, 52
scan, 177
schedule, 126
schedule_class, 126
schedule(), 126, 128
schedule.make_for_model(), 126, 127, 235
schedule.write(), 128, 236
schedule_scale, 111, 112, 113, 116, 119
score, 55
score_statistics, 177
Scorer, 131
script_file, 9
search, 183
search_group_list, 184
search_randomizations, 184
search_sort, 184
search_top_list, 184
segment_cutoff, 159
segment_growth_c, 159
segment_growth_n, 159
segment_ids, 79
segment_matching, 158
segment_report, 161
segment_shift, 159
select_atoms, 39
select_loop_atoms, 43
select_sphere, 102, 172
selection, 99
selection_class, 99
selection(), 16, 69, 70, 99, 102, 104, 172
selection.add(), 99, 101
selection_assess_dope(), 37, 111, 115, 116, 112
selection_assess_dopehr(), 37, 111, 116
selection_by_residue(), 99, 102
selection.debug_function(), 35, 114, 117, 235
selection_energy(), 17, 32, 55, 92, 73, 112, 114, 116
selection_extend_by_residue(), 102, 111
selection_hot_atoms(), 73, 111, 212, 235
selection_mutate(), 105, 235
selection_only_atom_types(), 99, 102
selection_only_defined(), 94, 99, 103
selection_only_het_residues(), 99, 103
selection_only_mainchain(), 99, 102
selection_only_no_topology(), 99, 103
selection_only_residue_types(), 99, 103
selection_only_sidechain(), 99, 102
selection_only_std_residues(), 99, 103
selection_only_water_residues(), 94, 99, 103
selection_randomize_xyz(), 106, 126, 223, 236
selection_rotate_dihedrals(), 110, 236
selection_rotate_mass_center(), 104
selection_rotate_origin(), 104, 236
selection_select_sphere(), 99, 102, 172
selection_supersose(), 35, 107, 146, 236
selection_transform(), 104, 236
selection_translate(), 104, 236
selection_unbuild(), 111, 236
selection_write(), 54, 72, 103
selection_segment, 220
send_data, 198
seq, 102
seq_database_file, 182, 184
seq_database_format, 185
seqid_cut, 185, 186
sequence, 32, 42, 111
Sequence_class, 161
Sequence_atom_file, 173, 136, 148, 191, 163, 225, 237
Sequence_chains, 68, 69, 99, 162, 164
Sequence_code, 136, 161, 225, 230, 237
Sequence_get_num_equiv(), 162
Sequence_get_sequence_identity(), 162
Sequence_name, 161
Sequence_prottyp, 161
INDEX

Sequence.range, 161
Sequence.residues, 38 88 69 99 102 107
Sequence.resolution, 161
Sequence.rfactor, 162
Sequence.source, 161
Sequence.transfer_res_prop(), 138 162
sequence_db, 182
sequence_db class, 182
sequence_db(), 182
sequence_db.convert(), 182, 183
sequence_db.filter(), 182 185 236
sequence_db.read(), 182 183 236
sequence_db.search(), 182 183 236
sequence_db.write(), 183 236
sge_pe_job, 196
sge_pe_job(), 196
sge_pe_slave, 199
sge_pe_slave(), 199
sge_qsub_job, 196
sge_qsub_job(), 196 197
sge_qsub_slave, 200
sge_qsub_slave(), 200
sheet, 98
sheet(), 98
sheet_hbonds, 98
signif_cutoff, 184
similarity_flag, 150 153 157
skip, 122
slave.run_cmd(), 195 196 199
smoothing_window, 113
soft-sphere, 30 31 44 48 61 62 88 118 126 212
solvation_model, 131
spaceflag, 190
special_patches, 41
special_patches, 71 195
special_restraints, 40
spline, 95
spline_dx, 89 95
spline_min_points, 89 95
spline_on_site, 89
spline_range, 89 95
ssh_command, 200
ssh_slave, 200
ssh_slave(), 200
start, 198
start, 69 122
start_type, 187 188
statistical_potential, 224
stdev, 49
stdevs, 49
steps, 126
strand, 98
strand(), 98
Structure.reread(), 103
Structure.write(), 84 103
structure_types, 163
submodel, 66
subopt_offset, 155
summary_file, 177
superpose, 107
superpose_refine, 107
SupersposeData, 107 108
surf_typ, 82
swap_atoms_in_res, 107
symmetry, 53
symmetry_restraints, 53
system, 59
temperature, 121 158
terms, 112 113
time_mark, 192
to_alignment, 176
to_iupac, 79
to_profile, 158
Topology.append(), 65 110 236
Topology.clear(), 65
Topology.make(), 65 67 235
Topology.read(), 37 65 67 236
Topology.submodel, 30 82 83 87 73 76 82 237
Topology.write(), 67 237
trace, 122
transfer_res_num, 195
transfer_res_prop, 162
transfer_xyz, 77
transform, 104
translate, 104
translate_type, 188
tutorial, 8
unbuild, 111
unpick, 94
unpick_all, 91
unpick_redundant, 92
use_conv, 190
use_lookup, 190
use_offset, 190
use_parallel_job, 41
use_rolloff, 190
user_after_single_loop_model, 83
user_after_single_model, 41
values, 117
values, 50
var_atom, 146
variability_file, 111
variable_target_function_method, 13
verbose, 193
very_fast, 40
very_very_verbose, 193
viol_report_cut, 111 113
viol_report_cut2, 113
virtual_atom.ch1, 51
virtual_atom.ch1a, 51
virtual_atom.ch2, 51
voxel_size, 187
VTFM, see variable target function method

weights, 19
weights_type, 151
write, 67, 72, 96, 103, 128, 137, 153, 155, 175, 183
write_all_atoms, 122
write_data, 51
write_fit, 148, 149, 151, 157
write_psf, 73
write_structure, 121
write_whole_pdb, 149, 158
wswitch, 190

z_comb, 85
z_pair, 85
z_surf, 85