MDT Documentation

Release 5.5

Dave Eramian, Ben Webb, Ursula Pieper

Jul 25, 2020

Contents

MDT is a module for protein structure analysis.

CHAPTER 1

Contents

1.1 Introduction

MDT prepares a raw frequency table, given information from MODELLER alignments and/or PDB files. It can also process the raw frequency table in several ways (e.g., normalization with `Table.normalize()`, smoothing with `Table.smooth()`, perform entropy calculations with `Table.entropy_full()`, and write out the data in various formats, including for plotting by ASGL (`Table.write_asgl()`) and use as restraints by MODELLER.

More precisely, MDT uses a sample of sequences, structures, and/or alignments to construct a table N(a,b,c,...,d) for features a, b, c, ..., d. The sample for generating the frequencies N is obtained depending on the type of features a, b, c, ..., d. The sample can contain individual proteins, pairs of proteins, pairs of residues in proteins, pairs of aligned residues, pairs of aligned pairs of residues, chemical bonds, angles, dihedral angles, and pairs of tuples of atoms. Some features work with triple alignments, too. All the needed features a, b, c, ..., d are calculated automatically from the sequences, alignments, and/or PDB files. The feature bins are defined by the user when each feature is created.

1.1.1 MDT features

A 'feature' in MDT is simply some binnable property of your input alignment. Example features include the `residue type`, `chi1` and `Phi` dihedral angles, `sequence identity` between two sequences, `X-ray resolution`, `atom-atom distances`, `atom type`, and `bond length`.

MDT understands that different features act on different sets of proteins, or parts of proteins, and will automatically scan over the correct range to collect necessary statistics (e.g. when you call `Table.add_alignment()`). For example, to collect statistics for the residue type feature, it is necessary to scan all residues in all proteins in the alignment. The X-ray resolution feature, on the other hand, only requires each protein in the alignment to be scanned, not each residue. The atom-atom distance feature requires scanning over all pairs of atoms in all proteins in the alignment, while the sequence identity feature requires scanning all pairs of proteins in the alignment. If you construct a table of multiple features, the most fine-grained of the features determines the scan - for example, a table of X-ray resolution against Φ dihedral would require a scan of all residues. See *the scan types table* for all of the scan types.

When choosing which proteins to scan, MDT also considers the features. It will scan each protein individually, all pairs of proteins, or all triples of proteins. The latter two scans only happen if you have features in your table that require multiple proteins (e.g. *protein pair* or *aligned residue* features) or you have single-protein features such as

protein or *residue* features but you have asked to evaluate them on the second or third protein (by setting the *protein* argument to 1 or 2 rather than the default 0).

MDT also knows that some *residue pair* or *atom pair* features are symmetric, and will perform a non-redundant scan in this case. If, however, any feature in the table is asymmetric, a full scan is performed. If in doubt, you can query `Table.symmetric` to see whether a symmetric scan will be performed for the current set of features. (Currently, any *tuple pair* feature in your table forces a full scan.)

The feature bins determine how to convert a feature value into a frequency table. For most feature types, you can specify how many bins to use, and their value ranges - see *Specification of bins* for more information. The last bin is always reserved as an 'undefined' bin, for values that don't fall into any other bin[1].

(Some features are predetermined by the setup of the system - for example, the `residue type` feature always has 22 bins - 20 for the standard amino acids, 1 for gaps in the alignment, and 1 for undefined.)

Type	Example feature
Protein	`features.XRayResolution`
Residue[2]	`features.Chi1Dihedral`
Residue pair[2][3]	`features.ResidueIndexDifference`
Atom	`features.AtomType`
Atom pair[3]	`features.AtomDistance`
Atom tuple	`features.TupleType`
Atom tuple pair	`features.TupleDistance`
Chemical bond	`features.BondType`
Chemical angle	`features.Angle`
Chemical dihedral angle	`features.Dihedral`

1.1.2 Dependent and independent features

An MDT `Table` object is simply a table of counts N(a,b,c,...,d) for features a, b, c, ..., d. However, this is often used to generate a conditional PDF, p(x,y,...,z | a,b,...,c) for independent features a, b, ..., c and dependent features x, y, ..., z. By convention in MDT the dependent features are the last or rightmost features in the table, and so methods which are designed to deal with PDFs such as `Table.smooth()`, `Table.super_smooth()`, `Table.normalize()`, `Table.offset_min()`, `Table.close()` expect the dependent features to be the last features. If necessary you can reorder the features using `Table.reshape()` or `Table.integrate()`.

1.1.3 Specification of bins

Most features take a *bins* argument when they are created, which specifies the bin ranges. This is simply a list of (start, end, symbol) triples, which specify the feature range for each bin, and the symbol to refer to it by. For example, the following creates an `X-ray resolution` feature, with 4 bins, the first for 0.51-1.4 , the second for 1.4-1.6 , and so on. Anything below 0.51 or 2.0 or above (or an undefined value) will be placed into a fifth 'undefined' bin.

[1] You can, however, remove the 'undefined' bin using `Table.reshape()` or by using the 'shape' argument when you create the `Table` object.

[2] Residue and residue pair scans are also used for 'one atom per residue' features, such as `features.ResidueDistance`, which is the distance between the 'special atom' in two residues. This special atom is usually Cα, but can be overridden by specifying the *distance_atoms* parameter when creating the `Library` object.

[3] When looking at pairs of atoms or residues, it is useful to extract information about the 'other' atom or residue in the pair. This other atom or residue is termed 'pos2' in MDT, and can be asked for when creating the feature. For example, when building a table of atom-atom distances (`features.AtomDistance`) it may be useful to tabulate it against the atom types of both the first and second atom. This is done by also using two copies of the `AtomType`, the second with *pos2=True*.

```
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 1.4, "<1.4"),
                                               (1.4,  1.6, "1.4-1.6"),
                                               (1.6,  1.8, "1.6-1.8"),
                                               (1.8,  2.0, "1.8-2.0")])
```

Note: Bin ranges in MDT are half-closed, i.e. a feature value must be greater than or equal to the lower value of the range, and less than the upper value, to be counted in the bin. For example, in the case above, 1.0 would be placed into the first bin, and 1.4 into the second. (If you define bins with overlapping ranges, values will be placed into the first bin that matches.)

In most cases, a set of bins of equal width is desired, and it is tedious to specify these by hand. A utility function *uniform_bins()* is provided, which takes three arguments - the number of bins, the lower range of the first bin, and the width of each bin - and creates a set of bins; all bins are of the same size and follow after the first bin. For example, the following bins the *atom-atom distance* feature into 60 bins, each 0.5 wide, with the first bin starting at 0 . The first bin is thus 0-0.5 , the second 0.5-1.0 , and so on, up to bin 60 which is 29.5-30.0 . The additional 'undefined' bin thus counts anything below 0 , greater than or equal to 30.0 , or which could not be calculated for some reason.

```
atdist = mdt.features.AtomDistance(mlib, bins=mdt.uniform_bins(60, 0, 0.5))
```

1.1.4 Storage for bin data

By default, when a table is created in MDT it uses double precision floating point to store the counts. This allows large counts themselves to be accurately scored, and can also store floating point data such as PDFs. However, for very large tables, this may use a prohibitive amount of memory. Therefore, it is possible to change the data type used to store bin data, by specifying the *bin_type* parameter when creating a *Table* object. The same parameter can be given to *Table.copy()*, to make a copy of the table using a different data type for its storage. Note that other data types use less storage, but can also store a smaller range of counts. For example, the *UnsignedInt8* data type uses only a single byte for each bin, but can only store integer counts between 0 and 255 (floating point values, or values outside of this range, will be truncated). MDT uses double precision floating point for all internal operations, but any storage of bin values uses the user-selected bin type. Thus you should be careful not to use an inappropriate bin type - for example, don't use an integer bin type if you are planning to store PDFs or perform normalization, smoothing, etc.

1.2 Usage

MDT is simply a Python extension module, and as such can be used in combination with other Python modules, such as MODELLER or the Python standard library.

1.2.1 Running pre-built binaries

The easiest way to use MDT is to install the pre-built binary RPM for your variety of Linux (this will first require you to install the Modeller RPM). Then you should simply be able to run an MDT script `foo.py` just like any regular Python script with a command similar to:

```
python foo.py
```

In the Sali lab, MDT is built as part of the nightly build system at the same time as MODELLER. Thus you can set up your system to run MDT scripts by running:

```
module load modeller
```

1.2.2 Using with Anaconda Python

There is an MDT package available for Anaconda Python for Mac and Linux. To install it, simply run:

```
conda install -c salilab mdt
```

1.2.3 Using a Homebrew package

If you are using a Mac with homebrew you can get MDT by running in a terminal window:

```
brew tap salilab/salilab; brew install mdt
```

If you don't already have Modeller installed, you can get it by running *brew install modeller* before you install MDT.

Add *–with-python3* to the end of each *brew install* command if you also want to use Python 3.

1.2.4 Compilation from source code

The MDT source code can be downloaded from GitHub.

Install dependent packages needed for MDT: MODELLER, glib, SWIG, pkg-config, and HDF5:

- MODELLER 9.15 or later is required.
- glib 2.4 or later is required. It is available as pre-built packages for most modern Linux distributions; there is also a MacPorts package for Mac users.
- SWIG 1.3.39 or later is required.
- Unfortunately HDF5 only works if you use the exact same version that is used by MODELLER. See the MODELLER ChangeLog for the version to use.

To compile, run *scons* in the same directory (and optionally *scons test*) to build (and test) MDT. This will produce a script `bin/mdtpy.sh` which can be used to run an MDT Python script `foo.py`:

```
bin/mdtpy.sh python foo.py
```

Note: If you didn't use the RPM or Debian package to install Modeller then you will need to tell MDT where it can find Modeller. To do this, create a file called `config.py`, and in it set the *modeller* Python variable to the directory where you have MODELLER installed (on a Mac, this would look like *modeller="/Library/modeller-XXX"* where *XXX* is the Modeller version).

If you installed any of the prerequisites in non-standard locations (i.e. not */usr/include* for glib and HDF5, and not */usr/bin* for pkg-config or SWIG) you will also need to tell scons where to find them. Add similar lines to `config.py` to set *path* for pkg-config and SWIG and *includepath* for glib and HDF5 (e.g. *path="/opt/local/bin"* and *includepath="/opt/local/include"* on a Mac).

If you want to install MDT, run *scons install*. You can additionally specify a *prefix* option (or set it in `config.py`) to install in a different directory. For example, *scons prefix=/foo install* will install MDT in the `/foo` directory.

1.2.5 Example MDT script

Generally speaking, to use MDT, you should

1. Create a *Library* object.
2. Read any necessary additional files into the library, such as the definitions of chemical bonds (see *Chemical bonds* for an example), or atom tuples.
3. Define one or more features, which are classes in the *mdt.features* module.
4. Create one or more *Table* objects, using a selection of the features you added to the Library, to hold the frequency tables themselves.
5. Collect statistics into the table using methods such as *Table.add_alignment()*.
6. Post process (e.g. *smoothing*, *normalizing*), *plot the data*, or *write the table to a file*.

A simple example, which simply collects the distribution of residue types in a PDB file, is shown below:

```
import modeller
import mdt
import mdt.features

# Setup of Modeller and MDT system
env = modeller.environ()
mlib = mdt.Library(env)

# Creation of feature types
restyp = mdt.features.ResidueType(mlib)

# Create a 1D table of residue type
table = mdt.Table(mlib, features=restyp)

# Read in a PDB file and make an alignment of just this one structure
mdl = modeller.model(env, file='5fd1')
aln = modeller.alignment(env)
aln.append_model(mdl, align_codes='5fd1', atom_files='5fd1')

# Collect MDT statistics for this alignment
table.add_alignment(aln)

# Print out the MDT by treating it as a Python list
print "Distribution of residue types:"
print [bin for bin in table]
```

For more applied examples, see *Sample studies with MDT*.

1.3 Sample studies with MDT

1.3.1 Introduction

This section describes the use of MDT for updating many of the MODELLER restraint libraries, including stereochemical, non-bonded, and homology-derived restraints:

1. Stereochemical restraints
    - chemical bonds: *p(Bond | BondType)*

- chemical angles: *p(Angle | AngleType)*
- improper dihedral angles as defined in the CHARMM residue topology file: *p(Dihedral | DihedralType)*
- chemical angles: *p(Angle | AngleType)*
- the ω dihedral angle of the peptide bond: *p(ω | ResidueType+1)* where ResidueType+1 refers to the residue type following the residue with the ω dihedral angle
- the Φ and Ψ dihedral angles: *p(Φ | ResidueType)*, *p(Ψ | ResidueType)*
- the sidechain dihedral angles: *p($\chi 1$ | ResidueType)*, *p($\chi 2$ | ResidueType)*, *p($\chi 3$ | ResidueType)*, *p($\chi 4$ | ResidueType)*
- the mainchain conformation: *p(Φ, Ψ | ResidueType)*

2. Non-bonded restraints
   - the mainchain hydrogen bonding restraints: *p(h | d, a)*
   - the non-bonded pair of atom triplets: *p(d, $\alpha 1$, $\alpha 2$, $\theta 1$, $\theta 2$, $\theta 3$ | t1, t2)*
3. Homology-derived restraints
   - distance: *p(d | d')*

The following sections will outline the process of starting with the Protein Data Bank (PDB) and ending up with the MODELLER restraint library files. We will describe the rationale for the process, input data sets, programs and scripts used, and even the analysis of the results. All of the input files should be found in the MDT distribution, in the `constr2005` directory.

The overall approach is to construct appropriately accurate, smooth, and transformed surfaces based on the statistics in PDB for use as spatial restraints for model building. The restraints from the first iteration will be used to construct many models, which in turn will be used to re-derive the equivalent restraints from the models. These model-derived restraints will then be compared against the original PDB-derived restraints to find problems and get indications as to how to change the restraints so that models are statistically as similar to PDB structures as possible.

1.3.2 Stereochemical restraints

The sample

The starting point for deriving the restraints in this section consists of 9,365 chains that are representative of the 65,629 chains in the October 2005 version of PDB. The representative set was obtained by clustering all PDB chains with MODELLER, such that the representative chains are from 30 to 3000 residues in length and are sharing less than 60% sequence identity to each other (or are more than 30 residues different in length). This is the corresponding MODELLER script:

```
from modeller import *
import re

log.verbose()
env = environ()
sdb = sequence_db(env, seq_database_file='pdball.pir',
                  seq_database_format='PIR',
                  chains_list='ALL', minmax_db_seq_len=(30, 3000),
                  clean_sequences=True)
sdb.filter(rr_file='${LIB}/blosum62.sim.mat', gap_penalties_1d=(-500, -50),
           matrix_offset = -450, max_diff_res=30, seqid_cut=60,
           output_grp_file='pdb_60.grp', output_cod_file='pdb_60.cod')
```

(continues on next page)

(continued from previous page)

```
# Make pdb_60.pir file by copying every sequence listed in pdb_60.cod
# from pdball.pir:
out = file("pdb_60.pir", "w")
codes = [line.rstrip('\r\n') for line in file("pdb_60.cod")]
codes = dict.fromkeys(codes)

pirhead = re.compile(">P1;(.*)$")
printline = False
for line in file("pdball.pir"):
    m = pirhead.match(line)
    if m:
        printline = m.group(1) in codes
    if printline:
        out.write(line)
```

The actual chains for restraint derivation are in fact a subset of the 9,365 representative chains, corresponding to the 4,532 crystallographic structures determined at least at 2 resolution (the representative structure for each group is the highest resolution x-ray structure in the group). This decision was made by looking at the distribution of the $\chi 1$ dihedral angles as a function of resolution (see *Sidechain dihedral angle $\chi 1$*) and the distribution of resolutions themselves for all 9,365 representative chains, using this MDT script:

```
from modeller import *
import mdt
import mdt.features

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=mdt.uniform_bins(20, 0, 0.2))
m = mdt.Table(mlib, features=xray)

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt2.mdt')
```

This script creates a *Library* object and then adds an X-ray resolution feature. Values of this feature are placed into 20 bins of width 0.2, starting at 0. It then creates a *Table* object, which is the MDT table itself. This starts off as an empty 1D table of the X-ray resolution feature. It then uses a MODELLER alignment object to read the sequences from `pdb_60.pir` one by one, and for each one it updates the X-ray resolution feature in the MDT table by calling *Table.add_alignment()*. Finally, the table is written out to the file `mdt2.mdt` using *Table.write()*.

The resulting MDT table `mdt2.mdt` was then plotted with the script:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=mdt.uniform_bins(20, 0, 0.2))
```

(continues on next page)

(continued from previous page)

```
m = mdt.Table(mlib, file='mdt2.mdt')

text = """
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999, WORLD_WINDOW = -999 -999 -999 -999
"""
m.write_asgl(asglroot='asgl2-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=2, text=text, x_decimal=1)

os.system("asgl asgl2-a")
os.system("ps2pdf asgl2-a.ps")
```

where the *Table.write_asgl()* method writes out an ASGL script and the MDT data in a form suitable for plotting (which we then execute with ASGL using Python's `os.system()` method). This gives an `impact of resolution plot`.

Chemical bonds

The MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)

# read the bond definitions in terms of the constituting atom type pairs:
mlib.bond_classes.read('${LIB}/bndgrp.lib')

# define the features:
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
bond_type = mdt.features.BondType(mlib)
bond_length = mdt.features.BondLength(mlib,
                                      bins=mdt.uniform_bins(400, 1.0, 0.0025))

m = mdt.Table(mlib, features=(xray, bond_type, bond_length))

# make the MDT table using the pdb_60 sample chains:
a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

# write out the MDT table:
m.write('mdt.mdt')
```

In this case, we read the file `bndgrp.lib` which defines all chemical bonds, using the *BondClasses.read()* method. The MDT we then construct is a 3D table of X-ray resolution, bond type, and bond length. The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
mlib.bond_classes.read('${LIB}/bndgrp.lib')
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
bond_type = mdt.features.BondType(mlib)
bond_length = mdt.features.BondLength(mlib,
                                      bins=mdt.uniform_bins(400, 1.0, 0.0025))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, bond_type, bond_length),
              offset=(0,0,0), shape=(1,-1,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = 1. 0.0025
SET BAR_XSHIFT = 0.00125
ZOOM SCALE_WORLDX = 0.08
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=999, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving `a set of bond plots`. Notice that here we use the *`Table.reshape()`* method, which can reshape a table by reordering the features, and/or reducing the bin ranges (offset or shape) of these features. In this case we don't change the feature order, or the offset (leaving it at the default 0,0,0) but we do change the shape. The first feature is restricted to only one bin - because our X-ray resolution feature contains only two bins (for "less than 2 " and the undefined bin, which catches everything 2 or greater) this keeps only the good structures for our plot. The other two features have their bin ranges reduced by 1 (a negative value for shape means "reduce the size by this amount"), which effectively removes the final ("undefined") bin.

Inspection of the plots shows that all distributions are mono-modal, but most are distinctly non-Gaussian. However, at this point, Gaussian restraints are still favored because the ranges are very narrow and because the non-Gaussian shape of the histograms may result from the application of all the other restraints (this supposition will be tested by deriving the corresponding distributions from the models, not PDB structures). Also, although many distributions are quite symmetrical, not all of them are. Therefore, there is the question of how best to fit a restraint to the data. There are at least three possibilities, in principle: (i) calculating the average and standard deviation from all (subset) of the data, (ii) least-squares fitting of the Gaussian model to the data, and (iii) using cubic splines of the data. The first option was adopted here: the mean and standard deviation will be the parameters of the analytically defined bond restraint for MODELLER.

The final MODELLER MDT library is produced with:

```
from modeller import *
import mdt
import mdt.features
```

(continues on next page)

(continued from previous page)

```
env = environ()
mlib = mdt.Library(env)
mlib.bond_classes.read('${LIB}/bndgrp.lib')
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
bond_type = mdt.features.BondType(mlib)
bond_length = mdt.features.BondLength(mlib,
                                      bins=mdt.uniform_bins(400, 1.0, 0.0025))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, bond_type, bond_length),
              offset=(0,0,0), shape=(0,-1,-1))

m = m.integrate(features=(bond_type, bond_length))

mdt.write_bondlib(file('bonds.py', 'w'), m, density_cutoff=0.1)
```

Here we use the *Table.integrate()* method, which removes a feature from the table by integrating the remaining features over all of that feature's bins, and the *write_bondlib()* function to write out a MODELLER script which builds restraints using the MDT-derived distributions.

Chemical angles

As for the bonds above, the MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
mlib.angle_classes.read('${LIB}/anggrp.lib')
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
angle_type = mdt.features.AngleType(mlib)
angle = mdt.features.Angle(mlib, bins=mdt.uniform_bins(720, 0, 0.25))

m = mdt.Table(mlib, features=(xray, angle_type, angle))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
```

(continues on next page)

(continued from previous page)

```
import mdt.features

env = environ()
mlib = mdt.Library(env)
mlib.angle_classes.read('${LIB}/anggrp.lib')
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
angle_type = mdt.features.AngleType(mlib)
angle = mdt.features.Angle(mlib, bins=mdt.uniform_bins(720, 0, 0.25))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, angle_type, angle),
              offset=(0,0,0), shape=(1,-1,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = 0. 0.25
SET BAR_XSHIFT = 0.125
ZOOM SCALE_WORLDX = 0.08
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=999, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving `a set of angle plots`.

The situation is similar to that for the chemical bonds, except that there are also four cases of bi-modal (as opposed to mono-modal) distributions: Asp:CB:CG:OD2, Asp:OD2:CG,OD1, Pro:CB:CG:CD, and Pro:CD:N:CA angles. The Asp bi-modal distribution may result from crystallographers mislabeling carboxyl oxygens for the protonated state of the sidechain (which is interesting because the corresponding angles might be used as a means to assign the protonation state). The mean values for these angles were edited by hand. Otherwise exactly the same considerations as for bonds apply here.

The final MODELLER MDT library is produced with:

```
from modeller import *
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
mlib.angle_classes.read('${LIB}/anggrp.lib')
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
angle_type = mdt.features.AngleType(mlib)
angle = mdt.features.Angle(mlib, bins=mdt.uniform_bins(720, 0, 0.25))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, angle_type, angle),
              offset=(0,0,0), shape=(0,-1,-1))

m = m.integrate(features=(angle_type, angle))
```

(continues on next page)

(continued from previous page)

```
mdt.write_anglelib(file('angles.py', 'w'), m, density_cutoff=0.1)
```

Improper dihedral angles

Exactly the same situation applies as for the chemical bonds. The MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
mlib.dihedral_classes.read('${LIB}/impgrp.lib')
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
impr_type = mdt.features.DihedralType(mlib)
improper = mdt.features.Dihedral(mlib, bins=mdt.uniform_bins(400, 1.0, 0.0025))

m = mdt.Table(mlib, features=(xray, impr_type, improper))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
mlib.dihedral_classes.read('${LIB}/impgrp.lib')
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
impr_type = mdt.features.DihedralType(mlib)
improper = mdt.features.Dihedral(mlib, bins=mdt.uniform_bins(400, 1.0, 0.0025))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, impr_type, improper),
              offset=(0,0,0), shape=(1,-1,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
```

(continues on next page)

(continued from previous page)

```
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 0.5
SET BAR_XSHIFT = 0.25
ZOOM SCALE_WORLDX = 0.08
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=999, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving `a set of improper plots`.

The final MODELLER MDT library is produced with:

```
from modeller import *
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
mlib.dihedral_classes.read('${LIB}/impgrp.lib')
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
impr_type = mdt.features.DihedralType(mlib)
improper = mdt.features.Dihedral(mlib, bins=mdt.uniform_bins(400, 1.0, 0.0025))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, impr_type, improper),
              offset=(0,0,0), shape=(1,-1,-1))

m = m.integrate(features=(impr_type, improper))

mdt.write_improperlib(file('impropers.py', 'w'), m, density_cutoff=0.1)
```

Sidechain dihedral angle $\chi 1$

The first question asked was "What is the impact of resolution on the distributions of residue $\chi 1$?". The answer was obtained by constructing and inspecting *p($\chi 1$ | R, resolution)* with:

```
from modeller import *
import mdt
import mdt.features

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 1.4, "under 1.4"),
                                               (1.4,  1.6, "1.4-1.6"),
                                               (1.6,  1.8, "1.6-1.8"),
                                               (1.8,  2.001, "1.8-2.0")])
restyp = mdt.features.ResidueType(mlib)
```

(continues on next page)

(continued from previous page)

```
chi1 = mdt.features.Chi1Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, features=(xray, restyp, chi1))

a = alignment(env)
f = modfile.File('../../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

and

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 1.4, "under 1.4"),
                                               (1.4,  1.6, "1.4-1.6"),
                                               (1.6,  1.8, "1.6-1.8"),
                                               (1.8,  2.001, "1.8-2.0")])
restyp = mdt.features.ResidueType(mlib)
chi1 = mdt.features.Chi1Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))


m = mdt.Table(mlib, file='mdt.mdt')
# Remove undefined bins (and gap residue type)
m = m.reshape(features=(xray, restyp, chi1), offset=m.offset, shape=(0,-2,-1))

text = """
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999, WORLD_WINDOW = -999 -999 -999 -999
"""
m.write_asgl(asglroot='asgl2-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=1)

os.system("asgl asgl2-a")
os.system("ps2pdf asgl2-a.ps")
```

giving `this output` which clearly shows that X-ray structures at resolution of at least 2.0 are just fine. X-ray structures above that resolution and NMR structures (whose resolution is set artificially to 0.45 for the purposes of MDT tabulation) do not appear to be suitable for deriving restraints for modeling, as the peaks are significantly wider and there is a significant population at ~120°. Also, the peaks appear Gaussian. Thus, a weighted sum of three Gaussians (except for Pro, which has two) was judged to be an appropriate model for these data. Again, the following script was used to construct the MDT table:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
```

(continues on next page)

(continued from previous page)

```
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi1 = mdt.features.Chi1Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, features=(xray, restyp, chi1))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

and the contents then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi1 = mdt.features.Chi1Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp, chi1), offset=(0,0,0), shape=(1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 2.5
SET BAR_XSHIFT = 1.25
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving a set of $\chi 1$ plots.

The weights, means, and standard deviations of the Gaussians were obtained by least-squares fitting with ASGL (with the script below) and are manually added to the MODELLER MDT library.

```
SET TICK_FONT = 13
SET BAR_GRAYNESS = 1.00
SET CAPTION_FONT = 12
```

(continues on next page)

(continued from previous page)

```
# The parameters are initial guesses
# (number of points, (weight, mean, standard deviation)_i; last weight missing),
# to help ASGL a little, but not important; just check the fitted curves
# against the data in fit.ps:
SET FIT_PARAM_INITIAL = 30000   0.3 175 10   0.3 -65 10   60  10
CALL ROUTINE = 'gauss3', FILE = 'c.dat', POSITION = 1 0, CAPTION_TEXT = 'C'
SET FIT_PARAM_INITIAL = 118000   0.3 175 10   0.3 -65 10   60  10
CALL ROUTINE = 'gauss3', FILE = 'd.dat', POSITION = 2 0, CAPTION_TEXT = 'D'
CALL ROUTINE = 'gauss3', FILE = 'e.dat', POSITION = 3 0, CAPTION_TEXT = 'E'
CALL ROUTINE = 'gauss3', FILE = 'f.dat', POSITION = 4 0, CAPTION_TEXT = 'F'
CALL ROUTINE = 'gauss3', FILE = 'h.dat', POSITION = 5 0, CAPTION_TEXT = 'H'
CALL ROUTINE = 'gauss3', FILE = 'i.dat', POSITION = 6 0, CAPTION_TEXT = 'I'
CALL ROUTINE = 'gauss3', FILE = 'k.dat', POSITION = 7 0, CAPTION_TEXT = 'K'
CALL ROUTINE = 'gauss3', FILE = 'l.dat', POSITION = 8 0, CAPTION_TEXT = 'L'
NEW_PAGE

SET FIT_PARAM_INITIAL =  45000   0.3 175 10   0.3 -65 10   60  10
CALL ROUTINE = 'gauss3', FILE = 'm.dat', POSITION = 1 0, CAPTION_TEXT = 'M'
SET FIT_PARAM_INITIAL = 88000   0.3 175 10   0.3 -65 10   60  10
CALL ROUTINE = 'gauss3', FILE = 'n.dat', POSITION = 2 0, CAPTION_TEXT = 'N'
# Pro has two peaks only, "gauss3' will still work as is:
SET FIT_PARAM_INITIAL = 95000   0.4 -30 7   0.4 40 7   0  5
CALL ROUTINE = 'gauss3', FILE = 'p.dat', POSITION = 3 0, CAPTION_TEXT = 'P'
SET FIT_PARAM_INITIAL = 76000   0.3 175 10   0.6 -65 20   62  10
CALL ROUTINE = 'gauss3', FILE = 'q.dat', POSITION = 4 0, CAPTION_TEXT = 'Q'
SET FIT_PARAM_INITIAL = 104000   0.3 175 10   0.6 -65 20   62  10
CALL ROUTINE = 'gauss3', FILE = 'r.dat', POSITION = 5 0, CAPTION_TEXT = 'R'
SET FIT_PARAM_INITIAL = 124000   0.3 175 10   0.6 -65 20   62  10
CALL ROUTINE = 'gauss3', FILE = 's.dat', POSITION = 6 0, CAPTION_TEXT = 'S'
SET FIT_PARAM_INITIAL = 112000   0.1 -175 10   0.5 -65 10   65  10
CALL ROUTINE = 'gauss3', FILE = 't.dat', POSITION = 7 0, CAPTION_TEXT = 'T'
SET FIT_PARAM_INITIAL = 147000   0.7 180 10   0.1 -65 10   65  10
CALL ROUTINE = 'gauss3', FILE = 'v.dat', POSITION = 8 0, CAPTION_TEXT = 'V'
NEW_PAGE

SET FIT_PARAM_INITIAL = 28000    0.2 175 10   0.5 -65 10   60  10
CALL ROUTINE = 'gauss3', FILE = 'w.dat', POSITION = 1 0, CAPTION_TEXT = 'W'
SET FIT_PARAM_INITIAL = 72000   0.2 175 10   0.7 -65 10   60  10
CALL ROUTINE = 'gauss3', FILE = 'y.dat', POSITION = 2 0, CAPTION_TEXT = 'Y'


SUBROUTINE ROUTINE = 'gauss3'

   READ_TABLE
   SET X_TICK = -180   10  180, X_TICK_LABEL =    1    6
   SET Y_TICK = -999 -999 -999, Y_TICK_LABEL = -999 -999
   SET XY_COLUMNS = 0 1
   # only to get 1, 2, 3, 4, 5, ... in column 2
   WORLD
   # get the right X-axis from -180 to +180:
   TRANSFORM NO_XY_SCOLUMNS = 1 0, XY_SCOLUMNS = 2, ;
             TRF_TYPE = 'LINEAR', TRF_PARAMETERS = -181.25 2.5
   WORLD WORLD_WINDOW = -190 0 190 -999
   AXES2D
   RESET_CAPTIONS
   CAPTION CAPTION_POSITION 1
```

(continues on next page)

(continued from previous page)

```
   CAPTION CAPTION_POSITION 2, CAPTION_TEXT '@c@_1_'
   CAPTION CAPTION_POSITION 3, CAPTION_TEXT 'FREQUENCY'
   HIST2D

   SET ERROR_COLUMN = 0
   SET FIT_MODEL = POLYGAUSS360
   # SET FIT_PARAM_INITIAL = 1639   0.3 175 10   0.3 -65 10   60  10
   SET FIT_PARAM_INDICES =  1 2 3 4 5 6 7 8 9
   FIT

   SMOOTH_TABLE SMOOTH_TYPE = 'SPLINE'
   PLOT2D PLOT2D_LINE_TYPE = 1, PLOT2D_SYMBOL_TYPE = 0

RETURN
END_SUBROUTINE
```

Sidechain dihedral angle $\chi 2$

The situation is very similar to that for $\chi 1$, except that the shapes of histograms are not Gaussian in most cases. Therefore, 1D cubic splines are used to represent the restraints.

The MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi2 = mdt.features.Chi2Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, features=(xray, restyp, chi2))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
```

(continues on next page)

(continued from previous page)

```
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi2 = mdt.features.Chi2Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp, chi2), offset=(0,0,0), shape=(1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 2.5
SET BAR_XSHIFT = 1.25
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving a set of $\chi 2$ plots.

The final MODELLER MDT library is produced with:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi2 = mdt.features.Chi2Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, file='mdt.mdt')

# remove the bins corresponding to undefined values for each of the 3 variables:
m = m.reshape(features=(xray, restyp, chi2), offset=(0,0,0), shape=(1,-2,-1))

# Let's get rid of the resolution variable from the output MDT table:
m = m.integrate(features=(restyp, chi2))

# Process the raw histograms to get appropriate pdf 1D splines for restraints:

# Start by smoothing with a uniform prior (equal weight when 10 points per bin),
# producing a normalized distribution that sums to 1 (not a pdf when dx != 1):
m = m.smooth(dimensions=1, weight=10)

# Normalize it to get the true pdf (Integral p(x) dx = 1):
# (the scaling actually does not matter, because I am eventually taking the
#  log and subtracting the smallest element of the final pdf, so this command
#  could be omitted without impact):
```

(continues on next page)

(continued from previous page)

```
m = m.normalize(to_pdf=True, dimensions=1, dx_dy=2.5, to_zero=True)

# Take the logarithm of the smoothed frequencies
# (this is safe: none of bins is 0 because of mdt.smooth()):
m = m.log_transform(offset=0., multiplier=1.)

# Reverse the sign:
m = m.linear_transform(offset=0., multiplier=-1.)

# Offset the final distribution so that the lowest value is at 0:
m = m.offset_min(dimensions=1)

mdt.write_splinelib(file("chi2.py", "w"), m, "chi2", density_cutoff=0.1)

text = """
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
"""
m.write_asgl(asglroot='modlib-a', plot_type='PLOT2D', every_x_numbered=20,
             text=text, dimensions=1, plot_position=1, plots_per_page=8)
os.system('asgl modlib-a')
```

This script also uses *Table.smooth()* to smooth the raw frequencies and *Table.normalize()* to convert the distribution into a PDF. It is then converted into a statistical potential by taking the negative log of the values (using the *Table.log_transform()*, *Table.linear_transform()*, and *Table.offset_min()* methods). The smoothing parameter *weight* of 10 was selected by trial and error, inspecting the resulting restraints in `modlib-a.ps`, also produced by the script above.

Sidechain dihedral angle $\chi 3$

Exactly the same considerations apply as to $\chi 2$. The MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi3 = mdt.features.Chi3Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, features=(xray, restyp, chi3))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)
```

(continues on next page)

(continued from previous page)

```
m.write('mdt.mdt')
```

The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi3 = mdt.features.Chi3Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp, chi3), offset=(0,0,0), shape=(1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 2.5
SET BAR_XSHIFT = 1.25
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving a set of $\chi 3$ plots. The final MODELLER MDT library is produced with:

```
from modeller import *
import os
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi3 = mdt.features.Chi3Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, file='mdt.mdt')

# remove the bins corresponding to undefined values for each of the 3 variables:
m = m.reshape(features=(xray, restyp, chi3), offset=(0,0,0), shape=(1,-2,-1))

# Let's get rid of the resolution variable from the output MDT table:
m = m.integrate(features=(restyp, chi3))

# Process the raw histograms to get appropriate pdf 1D splines for restraints:

# Start by smoothing with a uniform prior (equal weight when 10 points per bin),
# producing a normalized distribution that sums to 1 (not a pdf when dx != 1):
```

(continues on next page)

(continued from previous page)

```
m = m.smooth(dimensions=1, weight=10)

# Normalize it to get the true pdf (Integral p(x) dx = 1):
# (the scaling actually does not matter, because I am eventually taking the
#  log and subtracting the smallest element of the final pdf, so this command
#  could be omitted without impact):
m = m.normalize(to_pdf=True, dimensions=1, dx_dy=2.5, to_zero=True)

# Take the logarithm of the smoothed frequencies
# (this is safe: none of bins is 0 because of mdt.smooth()):
m = m.log_transform(offset=0., multiplier=1.)

# Reverse the sign:
m = m.linear_transform(offset=0., multiplier=-1.)

# Offset the final distribution so that the lowest value is at 0:
m = m.offset_min(dimensions=1)

mdt.write_splinelib(file("chi3.py", "w"), m, "chi3", density_cutoff=0.1)

text = """
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
"""
m.write_asgl(asglroot='modlib-a', plot_type='PLOT2D', every_x_numbered=20,
             text=text, dimensions=1, plot_position=1, plots_per_page=8)
os.system('asgl modlib-a')
```

The resulting restraints are plotted in `modlib-a.ps`, also produced by the script above.

Sidechain dihedral angle $\chi 4$

Exactly the same considerations apply as to $\chi 2$ and $\chi 3$. The MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi4 = mdt.features.Chi4Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, features=(xray, restyp, chi4))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
```

(continues on next page)

(continued from previous page)

```
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi4 = mdt.features.Chi4Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp, chi4), offset=(0,0,0), shape=(1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 2.5
SET BAR_XSHIFT = 1.25
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving a set of $\chi 4$ plots. The final MODELLER MDT library is produced with:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
chi4 = mdt.features.Chi4Dihedral(mlib, bins=mdt.uniform_bins(144, -180, 2.5))

m = mdt.Table(mlib, file='mdt.mdt')

# remove the bins corresponding to undefined values for each of the 3 variables:
m = m.reshape(features=(xray, restyp, chi4), offset=(0,0,0), shape=(1,-2,-1))

# Let's get rid of the resolution variable from the output MDT table:
m = m.integrate(features=(restyp, chi4))
```

(continues on next page)

(continued from previous page)

```
# Process the raw histograms to get appropriate pdf 1D splines for restraints:

# Start by smoothing with a uniform prior (equal weight when 10 points per bin),
# producing a normalized distribution that sums to 1 (not a pdf when dx != 1):
m = m.smooth(dimensions=1, weight=10)

# Normalize it to get the true pdf (Integral p(x) dx = 1):
# (the scaling actually does not matter, because I am eventually taking the
#  log and subtracting the smallest element of the final pdf, so this command
#  could be omitted without impact):
m = m.normalize(to_pdf=True, dimensions=1, dx_dy=2.5, to_zero=True)

# Take the logarithm of the smoothed frequencies
# (this is safe: none of bins is 0 because of mdt.smooth()):
m = m.log_transform(offset=0., multiplier=1.)

# Reverse the sign:
m = m.linear_transform(offset=0., multiplier=-1.)

# Offset the final distribution so that the lowest value is at 0:
m = m.offset_min(dimensions=1)

mdt.write_splinelib(file("chi4.py", "w"), m, "chi4", density_cutoff=0.1)

text = """
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
"""
m.write_asgl(asglroot='modlib-a', plot_type='PLOT2D', every_x_numbered=20,
             text=text, dimensions=1, plot_position=1, plots_per_page=8)
os.system('asgl modlib-a')
```

The resulting restraints are plotted in `modlib-a.ps`, also produced by the script above.

Mainchain dihedral angle Φ

Exactly the same considerations apply as to $\chi 2$, $\chi 3$, and $\chi 4$. The MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
phi = mdt.features.PhiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))
```

(continues on next page)

(continued from previous page)

```
m = mdt.Table(mlib, features=(xray, restyp, phi))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
phi = mdt.features.PhiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp, phi), offset=(0,0,0), shape=(-1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 2.5
SET BAR_XSHIFT = 1.25
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving a set of Φ plots. The final MODELLER MDT library is produced with:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
phi = mdt.features.PhiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))

m = mdt.Table(mlib, file='mdt.mdt')
```

(continues on next page)

(continued from previous page)

```
# remove the bins corresponding to undefined values for each of the 3 variables:
m = m.reshape(features=(xray, restyp, phi), offset=(0,0,0), shape=(-1,-2,-1))

# Let's get rid of the resolution variable from the output MDT table:
m = m.integrate(features=(restyp, phi))

# Process the raw histograms to get appropriate pdf 1D splines for restraints:

# Start by smoothing with a uniform prior (equal weight when 10 points per bin),
# producing a normalized distribution that sums to 1 (not a pdf when dx != 1):
m = m.smooth(dimensions=1, weight=10)

# Normalize it to get the true pdf (Integral p(x) dx = 1):
# (the scaling actually does not matter, because I am eventually taking the
#  log and subtracting the smallest element of the final pdf, so this command
#  could be omitted without impact):
m = m.normalize(to_pdf=True, dimensions=1, dx_dy=2.5, to_zero=True)

# Take the logarithm of the smoothed frequencies
# (this is safe: none of bins is 0 because of mdt.smooth()):
m = m.log_transform(offset=0., multiplier=1.)

# Reverse the sign:
m = m.linear_transform(offset=0., multiplier=-1.)

# Offset the final distribution so that the lowest value is at 0:
m = m.offset_min(dimensions=1)

mdt.write_splinelib(file("phi.py", "w"), m, "phi", density_cutoff=0.1)

text = """
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
"""
m.write_asgl(asglroot='modlib-a', plot_type='PLOT2D', every_x_numbered=20,
             text=text, dimensions=1, plot_position=1, plots_per_page=8)
os.system('asgl modlib-a')
```

The resulting restraints are plotted in `modlib-a.ps`, also produced by the script above.

Mainchain dihedral angle Ψ

Exactly the same considerations apply as to $\chi 2$, $\chi 3$, $\chi 4$, and Φ. The MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']
```

(continues on next page)

(continued from previous page)

```
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
psi = mdt.features.PsiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))

m = mdt.Table(mlib, features=(xray, restyp, psi))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
psi = mdt.features.PsiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp, psi), offset=(0,0,0), shape=(-1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 2.5
SET BAR_XSHIFT = 1.25
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving a set of Ψ plots. The final MODELLER MDT library is produced with:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
```

(continues on next page)

(continued from previous page)

```
restyp = mdt.features.ResidueType(mlib)
psi = mdt.features.PsiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))

m = mdt.Table(mlib, file='mdt.mdt')

# remove the bins corresponding to undefined values for each of the 3 variables:
m = m.reshape(features=(xray, restyp, psi), offset=(0,0,0), shape=(-1,-2,-1))

# Let's get rid of the resolution variable from the output MDT table:
m = m.integrate(features=(restyp, psi))

# Process the raw histograms to get appropriate pdf 1D splines for restraints:

# Start by smoothing with a uniform prior (equal weight when 10 points per bin),
# producing a normalized distribution that sums to 1 (not a pdf when dx != 1):
m = m.smooth(dimensions=1, weight=10)

# Normalize it to get the true pdf (Integral p(x) dx = 1):
# (the scaling actually does not matter, because I am eventually taking the
#  log and subtracting the smallest element of the final pdf, so this command
#  could be omitted without impact):
m = m.normalize(to_pdf=True, dimensions=1, dx_dy=2.5, to_zero=True)

# Take the logarithm of the smoothed frequencies
# (this is safe: none of bins is 0 because of mdt.smooth()):
m = m.log_transform(offset=0., multiplier=1.)

# Reverse the sign:
m = m.linear_transform(offset=0., multiplier=-1.)

# Offset the final distribution so that the lowest value is at 0:
m = m.offset_min(dimensions=1)

mdt.write_splinelib(file("psi.py", "w"), m, "psi", density_cutoff=0.1)

text = """
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
"""
m.write_asgl(asglroot='modlib-a', plot_type='PLOT2D', every_x_numbered=20,
             text=text, dimensions=1, plot_position=1, plots_per_page=8)
os.system('asgl modlib-a')
```

The resulting restraints are plotted in `modlib-a.ps`, also produced by the script above.

Mainchain dihedral angle ω

This dihedral angle is a little different from all others explored thus far because it depends more strongly on the type of the subsequent residue than the type of the residue whose dihedral angle is studied; that is, the ω of the residue preceding Pro, not the Pro ω, is impacted by the Pro residue. These dependencies are explored with MDT tables in directory `constr2005/omega/run1/`. The bottom line is that we need to set *delta* to 1 when creating our *residue type* feature (rather than the default value 0), which will make it refer to the type of the residue after the residue with the dihedral angle ω.

The next step is to obtain the *p(ω | R+1)* distributions with finer sampling of 0.5°:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp_1 = mdt.features.ResidueType(mlib, delta=1)
omega = mdt.features.OmegaDihedral(mlib, bins=mdt.uniform_bins(720, -180, 0.5))

# This table uses the subsequent residue type, relative to the omega
m = mdt.Table(mlib, features=(xray, restyp_1, omega))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

The `distribution in raw form` is then plotted with:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp_1 = mdt.features.ResidueType(mlib, delta=1)
omega = mdt.features.OmegaDihedral(mlib, bins=mdt.uniform_bins(720, -180, 0.5))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp_1, omega), offset=(0,0,0), shape=(1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 0.5
SET BAR_XSHIFT = 0.25
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

and `in logarithmic form` with:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp_1 = mdt.features.ResidueType(mlib, delta=1)
omega = mdt.features.OmegaDihedral(mlib, bins=mdt.uniform_bins(720, -180, 0.5))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp_1, omega), offset=(0,0,0), shape=(1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = -180. 0.5
SET BAR_XSHIFT = 0.25
TRANSFORM TRF_TYPE = LOGARITHMIC4, ;
          TRF_PARAMETERS = 1 1, NO_XY_SCOLUMNS = 0 1, XY_SCOLUMNS = 1
"""
m.write_asgl(asglroot='asgl2-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl2-a")
os.system("ps2pdf asgl2-a.ps")
```

Clearly, the peaks are sharp and will best be modeled by Gaussian distributions.

Similarly to $\chi 1$, two Gaussian distributions are fit to the histograms with the following ASGL script:

```
SET TICK_FONT = 13
SET BAR_GRAYNESS = 1.00
SET CAPTION_FONT = 12

SET FIT_PARAM_INITIAL = 87000   0.95 179 5   0 5
CALL ROUTINE = 'gauss2', FILE = 'a.dat', POSITION = 1 0, CAPTION_TEXT = 'A'
CALL ROUTINE = 'gauss2', FILE = 'c.dat', POSITION = 2 0, CAPTION_TEXT = 'C'
CALL ROUTINE = 'gauss2', FILE = 'd.dat', POSITION = 3 0, CAPTION_TEXT = 'D'
CALL ROUTINE = 'gauss2', FILE = 'e.dat', POSITION = 4 0, CAPTION_TEXT = 'E'
CALL ROUTINE = 'gauss2', FILE = 'f.dat', POSITION = 5 0, CAPTION_TEXT = 'F'
CALL ROUTINE = 'gauss2', FILE = 'g.dat', POSITION = 6 0, CAPTION_TEXT = 'G'
CALL ROUTINE = 'gauss2', FILE = 'h.dat', POSITION = 7 0, CAPTION_TEXT = 'H'
CALL ROUTINE = 'gauss2', FILE = 'i.dat', POSITION = 8 0, CAPTION_TEXT = 'I'
NEW_PAGE

CALL ROUTINE = 'gauss2', FILE = 'k.dat', POSITION = 1 0, CAPTION_TEXT = 'K'
CALL ROUTINE = 'gauss2', FILE = 'l.dat', POSITION = 2 0, CAPTION_TEXT = 'L'
CALL ROUTINE = 'gauss2', FILE = 'm.dat', POSITION = 3 0, CAPTION_TEXT = 'M'
CALL ROUTINE = 'gauss2', FILE = 'n.dat', POSITION = 4 0, CAPTION_TEXT = 'N'
CALL ROUTINE = 'gauss2', FILE = 'p.dat', POSITION = 5 0, CAPTION_TEXT = 'P'
CALL ROUTINE = 'gauss2', FILE = 'q.dat', POSITION = 6 0, CAPTION_TEXT = 'Q'
CALL ROUTINE = 'gauss2', FILE = 'r.dat', POSITION = 7 0, CAPTION_TEXT = 'R'
```

(continues on next page)

(continued from previous page)

```
CALL ROUTINE = 'gauss2', FILE = 's.dat', POSITION = 8 0, CAPTION_TEXT = 'S'
NEW_PAGE

CALL ROUTINE = 'gauss2', FILE = 't.dat', POSITION = 1 0, CAPTION_TEXT = 'T'
CALL ROUTINE = 'gauss2', FILE = 'v.dat', POSITION = 2 0, CAPTION_TEXT = 'V'
CALL ROUTINE = 'gauss2', FILE = 'w.dat', POSITION = 3 0, CAPTION_TEXT = 'W'
CALL ROUTINE = 'gauss2', FILE = 'y.dat', POSITION = 4 0, CAPTION_TEXT = 'Y'


SUBROUTINE ROUTINE = 'gauss2'

   READ_TABLE
   SET X_TICK = -180   10  180, X_TICK_LABEL =    1    6
   SET Y_TICK = -999 -999 -999, Y_TICK_LABEL = -999 -999
   SET XY_COLUMNS = 0 1
   # only to get 1, 2, 3, 4, 5, ... in column 2
   WORLD
   # get the right X-axis from -180 to +180:
   TRANSFORM NO_XY_SCOLUMNS = 1 0, XY_SCOLUMNS = 2, ;
             TRF_TYPE = 'LINEAR', TRF_PARAMETERS = -180.25 0.5
   WORLD WORLD_WINDOW = -190 0 190 -999
   AXES2D
   RESET_CAPTIONS
   CAPTION CAPTION_POSITION 1
   CAPTION CAPTION_POSITION 2, CAPTION_TEXT '@w@'
   CAPTION CAPTION_POSITION 3, CAPTION_TEXT 'FREQUENCY'
   HIST2D

   SET ERROR_COLUMN = 0
   SET FIT_MODEL = POLYGAUSS360
   # SET FIT_PARAM_INITIAL = 1639   0.3 175 10   0.3 -65 10   60  10
   SET FIT_PARAM_INDICES =  1 2 3 4 5 6
   FIT

   SMOOTH_TABLE SMOOTH_TYPE = 'SPLINE'
   PLOT2D PLOT2D_LINE_TYPE = 1, PLOT2D_SYMBOL_TYPE = 0

RETURN
END_SUBROUTINE
```

The means and standard deviations for each residue type are extracted from `fit.log` by the ASGL `get_prms.F` program, but they are only used to guess the means of 179.8° and 0° and standard deviations of 1.5° and 1.5° for the two peaks, respectively. The distribution of ω dihedral angles in the models calculated with these ω restraints will be checked carefully and the restraint parameters will be adapted as needed.

The weights of the peaks are not determined reliably by least-squares fitting in this case because the second weight is very close to 0 (in principle, they can even be less than zero). Therefore, they are determined separately by establishing $p(c\omega \mid R+1)$ where $c\omega$ is the class of the ω dihedral angle (1 or 2, *trans* or *cis*).

The MDT table is constructed with the following MDT Python script:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments
```

(continues on next page)

(continued from previous page)

```
env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp_1 = mdt.features.ResidueType(mlib, delta=1)
omega_class = mdt.features.OmegaClass(mlib)

# Table of the subsequent residue type relative to the omega class
m = mdt.Table(mlib, features=(xray, restyp_1, omega_class))

a = alignment(env)
f = modfile.File('../../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

The contents of the MDT table are then plotted with ASGL as follows:

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp_1 = mdt.features.ResidueType(mlib, delta=1)
omega_class = mdt.features.OmegaClass(mlib)

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp_1, omega_class),
              offset=(0,0,0), shape=(1,-2,-1))

text = """
SET X_LABEL_STYLE = 2, X_TICK_LABEL = -999 -999
SET X_TICK = -999 -999 -999
SET TICK_FONT = 5, CAPTION_FONT = 5
SET Y_TICK = -999 -999 -999
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 1 1, XY_SCOLUMNS = 2 1
FILL_COLUMN COLUMN = 2, COLUMN_PARAMETERS = 1 1
SET BAR_XSHIFT = 0.5
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=8, dimensions=1,
             plot_position=1, every_x_numbered=20, text=text, x_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving `an omega weights plot`.

The library `omega.py` is edited manually to replace the means and standard deviations with *179.8 0.0 2.3 2.3*.

Mainchain dihedral angles Φ and Ψ

The initial runs in `run1` explored Ramachandran maps extracted from different representative sets of structures (e.g., clustered by 40% sequence identity) and stratification by the crystallographic residue B_{iso} as well as resolution and residue type. We ended up with the sample and stratification described above.

The 2D histograms *p(Φ, Ψ | R)* are derived with:

```
from modeller import *
import mdt
import mdt.features

# See ../bonds/make_mdt.py for additional comments

env = environ()
log.minimal()
env.io.atom_files_directory = ['/salilab/park2/database/pdb/divided/']

mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
psi = mdt.features.PsiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))
phi = mdt.features.PhiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))

m = mdt.Table(mlib, features=(xray, restyp, psi, phi))

a = alignment(env)
f = modfile.File('../cluster-PDB/pdb_60.pir', 'r')
while a.read_one(f):
    m.add_alignment(a)

m.write('mdt.mdt')
```

They are plotted with

```
from modeller import *
import os
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
psi = mdt.features.PsiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))
phi = mdt.features.PhiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))

m = mdt.Table(mlib, file='mdt.mdt')
m = m.reshape(features=(xray, restyp, psi, phi),
              offset=(0,0,0,0), shape=(1,-2,-1,-1))

text = """
SET TICK_FONT = 5, CAPTION_FONT = 5
SET WORLD_WINDOW = -999 -999 -999 -999
SET NO_XY_SCOLUMNS = 0 0, DPLOT_BOUNDS 0.0 -999
TRANSFORM TRF_TYPE=LOGARITHMIC4, TRF_PARAMETERS=10 1
"""
m.write_asgl(asglroot='asgl1-a', plots_per_page=3, dimensions=2,
```

(continues on next page)

(continued from previous page)

```
            plot_position=9, every_x_numbered=12, every_y_numbered=12,
            text=text, x_decimal=0, y_decimal=0)

os.system("asgl asgl1-a")
os.system("ps2pdf asgl1-a.ps")
```

giving a set of Φ/Ψ plots.

The distributions are clearly not 2D Gaussian functions and need to be approximated by 2D cubic splines. Exploring and visualizing various smoothing options results in the following file to produce the final MODELLER MDT library:

```
from modeller import *
import mdt
import mdt.features

env = environ()
mlib = mdt.Library(env)
xray = mdt.features.XRayResolution(mlib, bins=[(0.51, 2.001, 'High res(2.0A)')])
restyp = mdt.features.ResidueType(mlib)
psi = mdt.features.PsiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))
phi = mdt.features.PhiDihedral(mlib, bins=mdt.uniform_bins(72, -180, 5.0))

m = mdt.Table(mlib, file='mdt.mdt')

# Eliminate the bins corresponding to undefined values:
m = m.reshape(features=(xray, restyp, psi, phi), offset=(0,0,0,0),
              shape=(1,-2,-1,-1))

# Let's get rid of the resolution variable from the output MDT table:
m = m.integrate(features=(restyp, psi, phi))

# Process the raw histograms to get appropriate pdf 1D splines for restraints:

# Start by smoothing with a uniform prior (equal weight when 10 points per bin),
# producing a normalized distribution that sums to 1 (not a pdf when dx != 1):
m = m.smooth(dimensions=2, weight=10)

# Normalize it to get the true pdf (Integral p(x) dx = 1):
# (the scaling actually does not matter, because I am eventually taking the
#  log and subtracting the smallest element of the final pdf, so this command
#  could be omitted without impact):
m = m.normalize(to_pdf=True, dimensions=2, dx_dy=(5., 5.), to_zero=True)

# Take the logarithm of the smoothed frequencies
# (this is safe: none of bins is 0 because of mdt.smooth()):
m = m.log_transform(offset=0., multiplier=1.)

# Reverse the sign:
m = m.linear_transform(offset=0., multiplier=-1.)

# Offset the final distribution so that the lowest value is at 0:
m = m.offset_min(dimensions=2)

mdt.write_2dsplinelib(file("phipsi.py", "w"), m, density_cutoff=0.1)
```

The raw, smooth, and transformed surfaces are visualized and compared best with Mathematica.

1.3.3 Non-bonded restraints

A general pairwise distance- and atom-type dependent statistical potential for all atom type pairs has been derived by Min-yi Shen (DOPE). MDT could, however, be used to derive specialized pairwise non-bonded restraints.

1.4 The `mdt` Python module

MDT, a module for protein structure analysis.

1.4.1 Setup of the MDT system

class mdt.**Library**(*env*, *distance_atoms=('CA', 'CA')*, *special_atoms=False*, *hbond_cutoff=3.5*)

Library data used in the construction and use of MDTs.

Parameters

- *env*: the Modeller environment to use
- *distance_atoms*: the atom types to use for the *features.ResidueDistance* feature
- *special_atoms*: whether to treat disulfide and termini atoms specially for atom class features (see *features.AtomType*)
- *hbond_cutoff*: maximum separation between two H-bonded atoms (see *features.HydrogenBondDonor*)

angle_classes

Angle classes; see *BondClasses*

atom_classes

Atom classes; see *BondClasses*

bond_classes

Bond classes; see *BondClasses*

dihedral_classes

Dihedral classes; see *BondClasses*

hbond_classes

Hydrogen bond atom classes; see *HydrogenBondClasses*

tuple_classes

Atom tuple classes; see *TupleClasses* and *Tuple features*

class mdt.**TupleClasses**(*mlib*)

Classifications of tuples of atoms into classes. Usually accessed as *Library.tuple_classes*. These classes are used by *tuple* or *tuple pair* features.

read(*filename*)

Read atom tuple information from *filename*. This is a text file with a format similar to that accepted by *BondClasses.read()*. The file can consist either of sets of atom triplets (named with TRPGRP lines and containing triples of atoms named on TRIPLET lines) or sets of atom doublets using DBLGRP and DOUBLET lines. Each atom but the first in each doublet or triplet can also be restricted to match only in certain residue types by naming the residue in parentheses before the rest of the atom name (and CHARMM-style + or - qualifier). For example, a suitable atom triplet file looks like:

```
TRPGRP 't1'
  TRIPLET 'ALA' 'CA' '+C' '-C'
TRPGRP 't2'
  TRIPLET 'ALA' 'CA' '(CYS)+C' '-C'
```

The first triplet is named 't1' and will match any set of three atoms where the first is called CA in an ALA residue, and the other two atoms are C atoms in the previous and next residue. The second triplet is similar but will only include triplets where the next residue is a CYS.

class mdt.**BondClasses**(*mlib*, *n_atom*)

Classifications of atoms/bonds/angles/dihedrals into classes. These classes are used by *atom* and *chemical bond* features. Usually accessed as *Library.atom_classes*, *Library.bond_classes*, *Library.angle_classes*, or *Library.dihedral_classes*. (There is no need to create your own BondClasses objects.)

read(*filename*)

Read class information from *filename*. This is a text file with a simple format. Each line either denotes the start of a new named class, or names a member of the last-named class, as a residue name followed by one or more atom names. For example, an atom class file might start with:

```
ATMGRP 'AC'
  ATOM 'ALA' 'CA'
  ATOM 'ALA' 'C'
  ATOM '*' 'CB'
```

Thus, the first atom class is called 'AC' and any CA or C atom in an ALA residue, or the CB atom in any residue, will be placed in this class.

Bond class files are similar but use BNDGRP and BOND lines, each of which names two atoms:

```
BNDGRP 'ALA:C:+N'
  BOND 'ALA' 'C' '+N'
```

Note that CHARMM-style + or - prefixes can be added to atom names for all but the first atom on a BOND line, to indicate the atom must be found in the next or previous residue.

Angle class files use ANGGRP and ANGLE lines; each ANGLE line names three atoms. Dihedral class files use DIHGRP and DIHEDRAL lines; each DIHEDRAL line names four atoms.

class mdt.**HydrogenBondClasses**(*mlib*)

Classifications of atoms into hydrogen bond classes. Usually accessed as *Library.hbond_classes*. These classes are used by the *features.HydrogenBondAcceptor*, *features.HydrogenBondDonor* and *features.HydrogenBondSatisfaction* features.

read(*filename*)

Read hydrogen bond atom class information from a file

1.4.2 Creation and manipulation of data tables

class mdt.**Table**(*mlib*, *file=None*, *features=None*, *bin_type=<mdt._BinType object>*, *shape=[]*)
A multi-dimensional table.

Parameters

- *mlib*: the MDT *Library* object to use
- *file*: if specified, the filename to read the initial table from (if the name ends with '.hdf5', `Table.read_hdf5()` is used, otherwise `Table.read()`)
- *features*: if specified (and *file* is not), a list of feature types to initialize the table with (using `Table.make()`)
- *bin_type*: type of storage for bin data (see *Storage for bin data*).
- *shape*: if specified with *features*, the shape of the new table (see `Table.make()`)

Individual elements from the table can be accessed in standard Python fashion, e.g.

```
>>> import mdt.features
>>> import modeller
>>> env = modeller.environ()
>>> mlib = mdt.Library(env)
>>> restyp1 = mdt.features.ResidueType(mlib, protein=0)
>>> restyp2 = mdt.features.ResidueType(mlib, protein=1)
>>> gap = mdt.features.GapDistance(mlib, mdt.uniform_bins(10, 0, 1))
>>> m = mdt.Table(mlib, features=(restyp1,restyp2,gap))
>>> print m[0,0,0]
```

You can also access an element as m[0][0][0], a 1D section as m[0][0], or a 2D section as m[0]. See `TableSection`.

add_alignment(*aln*, *distngh=6.0*, *surftyp=1*, *accessibility_type=8*, *residue_span_range=(-99999, -2, 2, 99999)*, *chain_span_range=(-99999, 0, 0, 99999)*, *bond_span_range=None*, *disulfide=False*, *exclude_bonds=False*, *exclude_angles=False*, *exclude_dihedrals=False*, *sympairs=False*, *symtriples=False*, *io=None*, *edat=None*)
Add data from a Modeller alignment to this MDT. This method will first scan through all proteins, pairs of proteins, or triples of proteins in the alignment (it will scan all triples if the `mdt.Library` contains features defined on all of proteins 0, 1 and 2, pairs if the features are defined on two different proteins, and individual proteins otherwise). Within each protein, it may then scan through all residues, atoms, etc. if the features request it (see *the scan types table*).

Parameters

- *aln*: Modeller alignment.
- *distngh*: distance below which residues are considered neighbors. Used by `features.NeighborhoodDifference`.
- *surftyp*: 1 for PSA contact area, 2 for surface area. Used by `features.AtomAccessibility`.
- *accessibility_type*: PSA accessibility type (1-10). Used by `features.AtomAccessibility`.
- *residue_span_range*: sequence separation (inclusive) for *residue pair*, *atom pair* and *tuple pair* features. For the two residue indices r1 and r2 in the tuple-tuple and atom- atom cases, or two alignment position indices in the residue-residue case, the following must be true:

residue_span_range[0] <= (r2 - r1) <= residue_span_range[1]

residue_span_range[2] <= (r2 - r1) <= residue_span_range[3]

For symmetric residue-residue features, only one condition must be met:

residue_span_range[2] <= abs(r2 - r1) <= residue_span_range[3]

For example, the default value of (-99999, -2, 2, 99999) excludes all pairs within the same residue (for which the sequence separation is 0) or within adjacent residues (for which the separation is 1 or -1).

- *chain_span_range*: works like *residue_span_range*, but for the chain indices. It is used only by the *atom pair* and *tuple pair* features. The default value of (-99999, 0, 0, 99999) allows all interactions. For example, using (-99999, -1, 1, 99999) instead would exclude all interactions within the same chain.
- *bond_span_range*: if given, it should be a list of two integers which specify the minimum and maximum number of bonds that separate a pair of atoms in the scan. It is used only by the *atom pair* and *tuple pair* features. (See `features.AtomBondSeparation` for more details.) The bond library (see `Library.bond_classes`) must be loaded to use this. For example, using (1, 2) will include only atoms that are directly chemically bonded or that are both bonded to a third atom, while (0, 9999) will only exclude pairs of atoms that have no path of bonds between them (e.g. atoms in different chains or when at least one of the atoms is not involved in any bonds). As a special case, if the maximum span is negative, no limit is enforced. For example, (2, 99999) will include all atoms that have a path of bonds between them except directly bonded pairs (and thus exclude pairs in different chains) while (2, -1) will also include inter-chain interactions.
- *disulfide*: if True, then the *bond_span_range* considers disulfide bonds (defined as any pair of SG atoms in CYS residues less than 2.5 angstroms apart) when calculating the bond separation between atoms. Only disulfide bridges within 3 residues of the atom pair are considered for computational efficiency.
- *exclude_bonds*: if True, then all pairs of atoms involved in a chemical bond (see `Library.bond_classes`) are excluded from *atom pair* and *tuple pair* features.
- *exclude_angles*: if True, then the 1-3 pair of atoms from each angle are excluded (see *exclude_bonds*).
- *exclude_dihedrals*: if True, then the 1-4 pair of atoms from each dihedral are excluded (see *exclude_bonds*).
- *sympairs*: if True, then protein pair scans are done in a symmetric fashion - e.g. when scanning an alignment of A, B and C, the following pairs are scanned: AB, BC, AC. By default a non-symmetric scan is performed, scanning AB, BC, AC, BA, CB, CA.
- *symtriples*: if True, then protein triple scans are done in a symmetric fashion - e.g. when scanning an alignment of A, B and C, the following triples are scanned: ABC, ACB, BAC. By default a non-symmetric scan is performed, scanning ABC, ACB, BAC, CBA, BCA, CAB.

add_alignment_witherr(*aln*, *distngh=6.0*, *surftyp=1*, *accessibility_type=8*, *residue_span_range=(-99999, -2, 2, 99999)*, *chain_span_range=(-99999, 0, 0, 99999)*, *bond_span_range=None*, *disulfide=False*, *exclude_bonds=False*, *exclude_angles=False*, *exclude_dihedrals=False*, *sympairs=False*, *symtriples=False*, *io=None*, *edat=None*, *errorscale=1*)

Add data from a Modeller alignment to this MDT. Same as add_alignment except the errors in data are taken into account. The parameter errorscale controls how the error is used:

- ***0*: the errors are ignored; this function is the same as** add_alignment.
- ***>0*** [the errors are taken into account by propagating the errors] in each axis of each atom into the calculated distances or angles. The errors in the position of individual atoms are first calculated using B-iso, X-ray resolution, and R-factor, and then divided by this errorscale value.

clear()
Clear the table (set all bins to zero)

close(*dimensions*)
Attempt to 'close' the MDT, so that it is useful for creating splines of periodic features.

If *dimensions* = 1, it makes the two terminal points equal to their average. If *dimensions* = 2, it applies the averages to both pairs of edges and then again to all four corner points.

Returns the closed MDT.

Return type *Table*

copy(*bin_type=None*)
If *bin_type* is specified, it is the storage type to convert the bin data to (see *Storage for bin data*).

Returns a copy of this MDT table.

Return type *Table*

entropy_full()
Print full entropy information.

entropy_hx()
The MDT is integrated to get a 1D histogram, then normalized by the sum of the bin values. Finally, entropy is calculated as Σ_i -p_i ln p_i

Returns the entropy of the last dependent variable.

Return type float

exp_transform(*offset*, *expoffset*, *multiplier*, *power*)
Apply an exponential transform to the MDT. Each element in the new MDT, *b*, is obtained from the original MDT element *a*, using the following relation: *b = offset + exp(expoffset + multiplier * a ^ power)*.

Return type *Table*

get_array_view()
Get a NumPy array 'view' of this Table. The array contains all of the raw data in the MDT table, allowing it to be manipulated with NumPy functions. The data are not copied; modifications made to the data by NumPy affect the data in the Table (and vice versa).

Functions that destroy the data in the Table (*Table.make()*, *Table.read()* and *Table.read_hdf5()*) cannot be called if any NumPy array views exist, since they would invalidate the views. The views must first be deleted.

If MDT was not built with NumPy support, a NotImplementedError exception is raised. If NumPy cannot be loaded, an ImportError is raised.

Returns a view of this table.

Return type NumPy array

integrate(*features*)
Integrate the MDT, and reorder the features. This is useful for squeezing large MDT arrays into smaller ones, and also for eliminating unwanted features (such as X-ray resolution) in preparation for *Table.write()*.

Parameters

- *features*: the new features (all must be present in the original MDT).

Returns the integrated MDT.

Return type *Table*

inverse_transform(*offset*, *multiplier*, *undefined=0.0*)

Apply an inverse transform to the MDT. Each element in the new MDT, *b*, is obtained from the original MDT element *a*, using the following relation: *b = offset + multiplier / a*. Where *a* is zero, *b* is assigned to be *undefined*.

Returns the transformed MDT.

Return type *Table*

linear_transform(*offset*, *multiplier*)

Apply a linear transform to the MDT. Each element in the new MDT, *b*, is obtained from the original MDT element *a*, using the following relation: *b = offset + a * multiplier*.

Returns the transformed MDT.

Return type *Table*

log_transform(*offset*, *multiplier*, *undefined=0.0*)

Apply a log transform to the MDT. Each element in the new MDT, *b*, is obtained from the original MDT element *a*, using the following relation: *b = ln(offset + multiplier * a)*. Where this would involve the logarithm of a negative number, *b* is assigned to be *undefined*.

Returns the transformed MDT.

Return type *Table*

make(*features*, *shape=[]*)

Clear the table, and set the features. *features* must be a list of previously created objects from the *mdt.features* module. If given, *shape* has the same meaning as in *Table.reshape()* and causes the table to use only a subset of the feature bins.

ValueError is raised if any views of the table exist (see *Table.get_array_view()*).

n_protein_pairs

Number of protein pairs

n_proteins

Number of proteins

normalize(*dimensions*, *dx_dy*, *to_zero*, *to_pdf*)

Normalize or scale the MDT. It does not really matter what the contents of the input MDT are; sensible contents include the raw or normalized frequencies.

Parameters

- *dimensions*: specifies whether a 1D or a 2D table is normalized. More precisely, the input distributions are *p(x | a, b, c, ...)* if *dimensions* = 1, or *p(x, y | a, b, c, ...)* if *dimensions* = 2, where y and x are the second to last and last features in the list of features.
- *dx_dy*: widths of the bins (either one or two numbers, depending on *dimensions*). If the value of either dx or dy is -999, the corresponding bin width is extracted from the MDT data structure (not available for all features).
- *to_zero*: if the histogram is empty, setting this True will set the bin values to zero, and False will yield a uniform distribution. It has no effect when the histogram is not empty.

- *to_pdf*: if False, the output is obtained by scaling the input such that for 1D histograms $\Sigma_i p(x_i) = 1$, and for 2D histograms $\Sigma_{i,j} p(x_{i,j}) = 1$. Note that *dx_dy* is **not** taken into account during this scaling.

  If it is True, the normalization takes into account *dx_dy* so that the normalized distribution is actually a PDF. That is, $\Sigma_i p(x_i)\, dx = 1$ for 1D and $\Sigma_{i,j} p(x_{i,j})\, dx\, dy = 1$ for 2D, where dx and dy are the widths of the bins.

Returns the normalized MDT.

Return type *Table*

offset_min (*dimensions*)
Offset the MDT by the minimum value, either in each 1D section (*dimensions* = 1) or in each 2D section (*dimensions* = 2).

Returns the transformed MDT.

Return type *Table*

open_alignment (*aln*, *distngh=6.0*, *surftyp=1*, *accessibility_type=8*, *sympairs=False*, *symtriples=False*, *io=None*, *edat=None*)
Open a Modeller alignment to allow MDT indices to be queried (see *Source*). Arguments are as for *Table.add_alignment()*.

Return type *Source*

pdf
Whether this MDT is a PDF

read (*file*)
Read an MDT from *file*. ValueError is raised if any views of the table exist (see *Table.get_array_view()*).

read_hdf5 (*file*)
Read an MDT in HDF5 format from *file*. ValueError is raised if any views of the table exist (see *Table.get_array_view()*).

reshape (*features*, *offset*, *shape*)
Reorder the MDT features and optionally decrease their ranges. When an MDT is created, each feature has exactly the bins defined in the *Library*'s bin file. However, for each feature, you can change the offset (initial number of bins from the bin file to omit) from the default 0, and the shape (total number of bins).

All parameters should be lists with the same number of elements as the MDT has features.

Parameters

- *features*: the new ordering of the MDT features.
- *offset*: the new offset (see *offset*).
- *shape*: the new shape (see *shape*). If any element in this list is 0 or negative, it is added to the MDT's existing shape to get the new value. Thus, a value of 0 would leave the shape unchanged, -1 would remove the last (undefined) bin, etc.

Returns the reshaped MDT.

Return type *Table*

sample_size
Number of sample points

smooth (*dimensions*, *weight*)
Smooth the MDT with a uniform prior. The MDT is treated either as a histogram (if *dimensions* = 1) or

a 2D density (*dimensions* = 2) of dependent features (the last 1 or 2 features in the table) and a uniform distribution is added followed by scaling:

$p_i = w_1 / n + w_2 v_i / S$

$S = \Sigma_i^n v_i$

$w_1 = 1 / (1 + S / (weight * n))$

$w_2 = 1 - w_1$

where *v* is the input MDT array, *n* is the number of bins in the histogram, and *p* is the output MDT array, smoothed and normalized. *weight* is the number of points per bin in the histogram at which the relative weights of the input histogram and the uniform prior are equal.

The sum of the bins in the output MDT array is 1, for each histogram.

Note that the resulting output MDT array is not necessarily a PDF, because the bin widths are not taken into account during scaling. That is, the sum of all bin values multiplied by the bin widths is not 1 if the bin widths are not 1.

Returns the smoothed MDT.

Return type *Table*

super_smooth (*dimensions*, *prior_weight*, *entropy_weighing*)

Multi-level smoothing. This super-smoothes the raw frequencies in the MDT using the hierarchical smoothing procedure for 1D histograms described in Sali and Blundell, JMB 1993. It was also employed in Sali and Overington, Prot Sci. 1994.

Briefly, the idea is to recursively construct the best possible prior distribution for smoothing 1D data *p(x | a, b, c, ...)*. The best prior is a weighted sum (weights optionally based on entropy) of the best possible estimate of *p(x | a, b, ...)* integrated over *c* for each *c*. Each one of these can itself be obtained from a prior and the data, and so on recursively.

The example above is for a single dependent feature (*x*), which is the case when *dimensions* = 1. *x* should be the last feature in the table. *dimensions* can be set to other values if you have more dependent features - for example, *dimensions* = 2 will work with *p(x, y | a, b, c, ...)* where *x* and *y* are the last two features in the table.

Parameters

- *dimensions*: Number of dependent features.
- *prior_weight*: Weight for the prior distribution.
- *entropy_weighing*: Whether to weight distributions by their entropies.

Returns the smoothed MDT.

Return type *Table*

symmetric

True if a symmetric scan can be performed

write (*file*, *write_preamble=True*)

Write the table to *file*. If *write_preamble* is False, it will only write out the contents of the MDT table, without the preamble including the feature list, bins, etc. This is useful for example for creating a file to be read by another program, such as Mathematica.

write_asgl (*asglroot*, *text*, *dimensions*, *plot_position*, *plots_per_page*, *plot_density_cutoff=-1.0*, *plot_type='HIST2D'*, *every_x_numbered=1*, *every_y_numbered=1*, *x_decimal=1*, *y_decimal=1*)

Make input files for ASGL.

Parameters

- *asglroot*: filename prefix for ASGL TOP script and data files.
- *text*: ASGL command lines that are written for each plot.
- *dimensions*: whether to make 1D or 2D plots.
- *plot_position*: position of the plot on the page, in ASGL convention.
- *plots_per_page*: number of plots per page.
- *plot_density_cutoff*: the minimal sum of the bin values that each plot has to have before it is actually written out; otherwise it is ignored. This helps to avoid wasting paper on empty plots when the MDT array data are sparse.
- *plot_type*: select 'HIST2D' or 'PLOT2D' when *dimensions* = 2.
- *every_x_numbered*: spacing for labels on the X axis.
- *every_y_numbered*: spacing for labels on the Y axis.
- *x_decimal*: the number of decimal places used to write X feature values.
- *y_decimal*: the number of decimal places used to write Y feature values.

write_hdf5(*file*, *gzip=False*, *chunk_size=10485760*)

Write an MDT in HDF5 format to *file*. Certain library information (such as the mapping from feature values to bin indices, and atom or tuple class information) and information about the last scan is also written to the file. (This information will be missing or incomplete if *add_alignment()* hasn't first been called.) Note that this information is not read back in by *read_hdf5()*; it is intended primarily for other programs that want to reproduce the environment in which the MDT was generated as closely as possible.

Parameters

- *gzip*: If True, compress the table in the HDF5 file with gzip using the default compresion level; if a number from 0-9, compress using that gzip compression level (0=no compression, 9=most); if False (the default) do not compress.
- *chunk_size*: when using gzip, the table must be split up into chunks (otherwise it is written contiguously). This parameter can either be a list (the same length as the number of features) defining the size of each chunk, or it can be the approximate number of data points in each chunk, in which case the dimensions of the chunk are chosen automatically.

class mdt.**TableSection**(*mdt*, *indices*)

A section of a multi-dimensional table. You should not create TableSection objects directly, but rather by indexing a *Table* object, as a TableSection is just a 'view' into an existing table. For example,

```
>>> m = mdt.Table(mlib, features=(residue_type, xray_resolution))
>>> print m[0].entropy()
```

would create a section (using m[0]) which is a 1D table over the 2nd feature (X-ray resolution) for the first bin (0) of the first feature (residue type), and then get the entropy using the *TableSection.entropy()* method.

entropy()

Entropy of all points in the table

features

Features in this MDT; a list of *Feature* objects

mean_stdev()

Mean and standard deviation of the table

offset
Array offsets; see *Feature.offset*

shape
Array shape; the number of bins for each feature

sum()
Sum of all points in the table

class mdt.**Feature**(*mdt*, *indx*)
A single feature in an MDT. Generally accessed as *TableSection.features*.

bins
Feature bins; a list of *Bin* objects

ifeat
Integer type

offset
Offset of first bin compared to the MDT library feature (usually 0, but can be changed with *Table.reshape()*)

periodic
Whether feature is periodic

class mdt.**Bin**(*feature*, *indx*)
A single bin in a feature. Generally accessed as *Feature.bins*.

range
Bin range; usually the minimum and maximum floating-point values for the feature to be placed in this bin.

symbol
Bin symbol

class mdt.**Source**(*mdt*, *mlib*, *aln*, *distngh*, *surftyp*, *accessibility_type*, *sympairs*, *symtriples*, *io*, *edat*)
A source of data for an MDT (generally a Modeller alignment, opened with *Table.open_alignment()*).

index(*feat*, *is1*, *ip1*, *is2*, *ir1*, *ir2*, *ir1p*, *ir2p*, *ia1*, *ia1p*, *ip2*, *ibnd1*, *ibnd1p*, *is3*, *ir3*, *ir3p*)
Return the bin index (starting at 1) of a single MDT feature. (Arguments ending in 2 and 3 are used for features involving pairs or triples of proteins.)

> **Warning:** This is a low-level interface, and no bounds checking is performed on these parameters. Avoid this function if possible.

Parameters

- *feat*: MDT feature object from *mdt.features* module.
- *is1*: index of the sequence within the alignment.
- *ip1*: position within the sequence (i.e. including gaps).
- *ir1*: residue index (i.e. not including alignment gaps).
- *ir1p*: second residue index for residue-residue features.
- *ia1*: atom index.
- *ia1p*: second atom index for atom-atom features.
- *ibnd1*: bond or tuple index.

- *ibnd1p*: second bond/tuple index for bond-bond or tuple-tuple features.

sum(*residue_span_range=(-99999, -2, 2, 99999)*, *chain_span_range=(-99999, 0, 0, 99999)*, *bond_span_range=None*, *disulfide=False*, *exclude_bonds=False*, *exclude_angles=False*, *exclude_dihedrals=False*)
Scan all data points in the source, and return the sum. See `Table.add_alignment()` for a description of the *residue_span_range*, *chain_span_range* and *exclude_** arguments.

1.4.3 Library information

mdt.**version**
The full MDT version number, as a string, e.g. '5.0' or 'SVN'.

mdt.**version_info**
For release builds, the major and minor version numbers as a tuple of integers - e.g. (5, 0). For SVN builds, this is the same as 'version'.

1.4.4 Utility functions

mdt.**uniform_bins**(*num*, *start*, *width*)
Make a list of *num* equally-sized bins, each of which has the given *width*, and starting at *start*. This is suitable for input to any of the classes in `mdt.features` which need a list of bins.

mdt.**write_bondlib**(*fh*, *mdt*, *density_cutoff=None*, *entropy_cutoff=None*)
Write out a Modeller bond library file from an MDT. The input MDT should be a 2D table (usually of bond type and bond distance). For each bond type, the 1D MDT section (see `TableSection`) of bond distance is examined, and its mean and standard deviation used to generate a Modeller harmonic restraint.

Parameters

- *fh*: Python file to write to
- *mdt*: input MDT `Table` object
- *density_cutoff*: if specified, MDT bond distance sections with sums below this value are not used
- *entropy_cutoff*: if specified, MDT bond distance sections with entropies above this value are not used

mdt.**write_anglelib**(*fh*, *mdt*, *density_cutoff=None*, *entropy_cutoff=None*)
Write out a Modeller angle library file from an MDT. See `write_bondlib()` for more details. The MDT should be a 2D table, usually of angle type and bond angle.

mdt.**write_improperlib**(*fh*, *mdt*, *density_cutoff=None*, *entropy_cutoff=None*)
Write out a Modeller dihedral angle library file from an MDT. See `write_bondlib()` for more details. The MDT should be a 2D table, usually of dihedral type and bond dihedral angle.

mdt.**write_splinelib**(*fh*, *mdt*, *dihtype*, *density_cutoff=None*, *entropy_cutoff=None*)
Write out a Modeller 1D spline library file from an MDT. The MDT should be a 2D table, usually of residue type and a chi dihedral angle. *dihtype* should identify the dihedral type (i.e. chi1/chi2/chi3/chi4). The operation is similar to `write_bondlib()`, but each MDT section is treated as the spline values. No special processing is done, so it is expected that the user has first done any necessary transformations (e.g. normalization with `Table.normalize()` to convert raw counts into a PDF, negative log transform with `Table.log_transform()` and `Table.linear_transform()` to convert a PDF into a statistical potential).

mdt.**write_2dsplinelib**(*fh*, *mdt*, *density_cutoff=None*, *entropy_cutoff=None*)
Write out a Modeller 2D spline library file from an MDT. See *write_splinelib()* for more details. The input MDT should be a 3D table, e.g. of residue type, phi angle, and psi angle.

mdt.**write_statpot**(*fh*, *mdt*)
Write out a Modeller statistical potential file (as accepted by group_restraints.append()). The MDT is assumed to be a 3D table of distance against the types of the two atoms. No special processing is done, so it is expected that the user has first done any necessary transformations (e.g. normalization with *Table.normalize()* to convert raw counts into a PDF, negative log transform with *Table.log_transform()* and *Table.linear_transform()* to convert a PDF into a statistical potential).

1.4.5 Bin storage types

mdt.**Float**
mdt.**Double**
mdt.**Int32**
mdt.**UnsignedInt32**
mdt.**Int16**
mdt.**UnsignedInt16**
mdt.**Int8**
mdt.**UnsignedInt8**
See *Storage for bin data*.

1.4.6 Exceptions

exception mdt.**MDTError**
A generic MDT error.

exception mdt.**FileFormatError**
A file is of the wrong format.

1.5 The `mdt.features` Python module

MDT features.

1.5.1 Protein features

These features yield a single value for each protein in the alignment. Each feature takes some common arguments:

- *mlib*: the *mdt.Library* to create the feature in.
- *bins*: list of bins (see *Specification of bins*).

- *protein*: the protein index on which to evaluate the feature from each group of proteins (individual protein, pairs, triples) selected from the alignment (0 for the first, 1 for the second, 2 for the third). See `mdt.Table.add_alignment()` for more details.

class mdt.features.**XRayResolution**(*mlib*, *bins*, *protein=0*, *nmr=0.45*)
Protein X-ray resolution in angstroms. Proteins with a resolution of -1.00 (generally NMR structures) are actually reported as having a resolution of *nmr*. This decreases the number of bins required to hold all defined resolutions while still separating NMR from X-ray structures.

class mdt.features.**RadiusOfGyration**(*mlib*, *bins*, *protein=0*)
Protein radius of gyration in angstroms. The calculation of the center of mass used for this feature is not mass weighted.

class mdt.features.**SequenceLength**(*mlib*, *bins*, *protein=0*)
Protein sequence length (number of residues).

class mdt.features.**HydrogenBondSatisfaction**(*mlib*, *bins*, *protein=0*)
Hydrogen bond satisfaction index for a protein. This is the average difference, over all atoms in the protein, between the HydrogenBondDonor value and the atom's donor valency plus the same for the acceptor, as defined in the hydrogen bond file (see `mdt.Library.hbond_classes`).

class mdt.features.**AlphaContent**(*mlib*, *bins*, *protein=0*)
Alpha content of the protein. This is simply the fraction, between 0 and 1, of residues in the first mainchain conformation class (see `MainchainConformation`).

1.5.2 Protein pair features

These features yield a single value for each pair of proteins in the alignment. Each feature takes some common arguments:

- *mlib*: the `mdt.Library` to create the feature in.
- *bins*: list of bins (see *Specification of bins*).
- *protein1* and *protein2*: the indexes of proteins in each group of proteins selected from the alignment to evaluate the feature on; each can range from 0 to 2 inclusive. See `mdt.Table.add_alignment()` for more details.

class mdt.features.**SequenceIdentity**(*mlib*, *bins*, *protein1=0*, *protein2=1*)
Fractional sequence identity, between 0 and 1, between two sequences. This is the number of identical aligned residues divided by the length of the shorter sequence.

1.5.3 Residue features

These features yield a single value for each residue in each sequence in the alignment. Each feature takes some common arguments:

- *delta*: if non-zero, don't calculate the feature for the residue position returned by the residue scan - instead, offset it by *delta* residues in the sequence. Applied before *align_delta*.
- *align_delta*: if non-zero, don't calculate the feature for the alignment position returned by the residue scan - instead, offset it by *align_delta* alignment positions. Applied after *delta*.
- *pos2*: if True, force a residue pair scan, and evaluate the feature on the second residue in each pair.
- *mlib*, *bins*, *protein*: see *Protein features*. Note that some residue features do not use the *bins* argument, because they have a fixed number of bins.

class mdt.features.**ResidueType**(*mlib*, *protein=0*, *delta=0*, *align_delta=0*, *pos2=False*)
Residue type (20 standard amino acids, gap, undefined).

class mdt.features.**ResidueAccessibility**(*mlib*, *bins*, *protein=0*, *delta=0*, *align_delta=0*, *pos2=False*)

Residue solvent accessibility. This is derived from the atomic solvent accessibility; see *AtomAccessibility*.

class mdt.features.**Chi1Dihedral**(*self*, *mlib*, *protein=0*, *delta=0*, *align_delta=0*, *pos2=False*)
class mdt.features.**Chi2Dihedral**
class mdt.features.**Chi3Dihedral**
class mdt.features.**Chi4Dihedral**
class mdt.features.**PhiDihedral**
class mdt.features.**PsiDihedral**
class mdt.features.**OmegaDihedral**
class mdt.features.**AlphaDihedral**

Residue dihedral angle, from -180 to 180 degrees.

class mdt.features.**Chi1Class**(*self*, *mlib*, *protein=0*, *delta=0*, *align_delta=0*, *pos2=False*)
class mdt.features.**Chi2Class**
class mdt.features.**Chi3Class**
class mdt.features.**Chi4Class**
class mdt.features.**Chi5Class**
class mdt.features.**PhiClass**
class mdt.features.**PsiClass**
class mdt.features.**OmegaClass**

Residue dihedral class. These classes are defined by MODELLER to group common regions of dihedral space for each residue type.

class mdt.features.**MainchainConformation**(*mlib*, *protein=0*, *delta=0*, *align_delta=0*, *pos2=False*)

Residue mainchain conformation (Ramachandran) class. This is a classification of the residue's phi/psi angles into classes as defined in Modeller's modlib/af_mnchdef.lib file and described in Sali and Blundell, JMB (1993) 234, p785. The default classes are A (right-handed alpha-helix), P (poly-proline conformation), B (idealized beta-strand), L (left-handed alpha-helix), and E (extended conformation).

class mdt.features.**ResidueGroup**(*mlib*, *protein=0*, *delta=0*, *align_delta=0*, *pos2=False*, *residue_grouping=0*)

Residue group.

class mdt.features.**SidechainBiso**(*mlib*, *bins*, *protein=0*, *delta=0*, *align_delta=0*, *pos2=False*)

Residue average sidechain B_{iso}. A zero average B_{iso} is treated as undefined. If the average of these values over the whole protein is less than 2, each residue's value is multiplied by $4\ \pi^2$.

1.5.4 Residue pair features

These features yield a single value for each pair of residues in each sequence in the alignment. See *Protein features* for a description of the common arguments.

class mdt.features.**ResidueDistance**(*mlib*, *bins*, *protein=0*)

Distance between a pair of residues. This is defined as the distance between the 'special' atoms in each residue. The type of this special atom can be specified by the distance_atoms argument when creating a *mdt.Library* object. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**AverageResidueAccessibility**(*mlib*, *bins*, *protein=0*)

Average solvent accessibility of a pair of residues. See *ResidueAccessibility*.

class mdt.features.**ResidueIndexDifference**(*mlib*, *bins*, *protein=0*, *absolute=False*)

Difference in sequence index between a pair of residues. This can either be the simple difference (if *absolute*

is False) in which case the feature is asymmetric, or the absolute value (if *absolute* is True) which gives a symmetric feature.

1.5.5 Aligned residue features

These features yield a single value for residues aligned between two proteins. For each pair of proteins, every alignment position is scanned, and the feature is evaluated for each pair of aligned residues. See *Protein pair features* for a description of the common arguments.

class mdt.features.**PhiDihedralDifference**(*self*, *mlib*, *bins*, *protein1=0*, *protein2=1*)
class mdt.features.**PsiDihedralDifference**
class mdt.features.**OmegaDihedralDifference**
Shortest difference in dihedral angle (in degrees) between a pair of aligned residues.

class mdt.features.**NeighborhoodDifference**(*mlib*, *bins*, *protein1=0*, *protein2=1*)
Residue neighborhood difference. This is the average of the distance scores (from a residue-residue scoring matrix) of all aligned residues where the residue in the first sequence is within a cutoff distance of the scanned residue. (This cutoff is set by the distngh argument to *mdt.Table.add_alignment()*.)

class mdt.features.**GapDistance**(*mlib*, *bins*, *protein1=0*, *protein2=1*)
Distance, in alignment positions, to the nearest gap. Note that positions which are gapped in both sequences are ignored for the purposes of this calculation (a 'gap' is defined as a gap in one sequence aligned with a residue in the other).

1.5.6 Aligned residue pair features

These features yield a single value for each pair of residues aligned between two proteins. For each pair of proteins, each pair of alignment positions is scanned, and the feature is evaluated for each pair of pairs of aligned residues. See *Protein pair features* for a description of the common arguments.

class mdt.features.**ResidueDistanceDifference**(*mlib*, *bins*, *protein1=0*, *protein2=1*)
Distance between two residues in the second protein, minus the distance between the equivalent residues in the first protein. See *ResidueDistance*. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**AverageNeighborhoodDifference**(*mlib*, *bins*, *protein1=0*, *protein2=1*)
Average residue neighborhood difference for a pair of alignment positions. See *NeighborhoodDifference*.

class mdt.features.**AverageGapDistance**(*mlib*, *bins*, *protein1=0*, *protein2=1*)
Average distance to a gap from a pair of alignment positions. See *GapDistance*.

1.5.7 Atom features

These features yield a single value for each atom in the first protein in each group of proteins selected from the alignment. Each feature takes some common arguments:

- *pos2*: if True, force an atom pair scan, and evaluate the feature on the second atom in each pair.
- *mlib*, *bins*: see *Protein features*. Note that some atom features do not use the *bins* argument, because they have a fixed number of bins.

class mdt.features.**AtomAccessibility**(*mlib*, *bins*, *pos2=False*)
Atom solvent accessibility. This is calculated by the PSA algorithm, and controlled by the surftyp and accessibility_type arguments to *mdt.Table.add_alignment()*. The feature is considered undefined if the atom's Cartesian coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**FractionalAtomAccessibility**(*mlib*, *bins*, *pos2=False*)
Fractional atom solvent accessibility, from 0 to 1. This is the atom solvent accessibility (see *AtomAccessibility*) divided by the volume of the atom, derived from its van der Waals radius. The feature is considered undefined if the atom's Cartesian coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**AtomType**(*mlib*, *pos2=False*)
Type of an atom, as classified by the atom class file. See *mdt.Library.atom_classes*.

class mdt.features.**HydrogenBondDonor**(*mlib*, *bins*, *pos2=False*)
Number of hydrogen bond donors. It is defined as the sum, over all atoms within hbond_cutoff (see *mdt.Library*) of the atom, of their donor valencies as defined in the hydrogen bond file (see *mdt.Library.hbond_classes*). The feature is considered undefined if the atom's Cartesian coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**HydrogenBondAcceptor**(*mlib*, *bins*, *pos2=False*)
Number of hydrogen bond acceptors. It is defined as the sum, over all atoms within hbond_cutoff (see *mdt.Library*) of the atom, of their acceptor valencies as defined in the hydrogen bond file (see *mdt.Library.hbond_classes*). The feature is considered undefined if the atom's Cartesian coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**HydrogenBondCharge**(*mlib*, *bins*, *pos2=False*)
Hydrogen bond charge. It is defined as the sum, over all atoms within hbond_cutoff (see *mdt.Library*) of the atom, of their charges as defined in the hydrogen bond file (see *mdt.Library.hbond_classes*).

class mdt.features.**AtomTable**(*mlib*, *bins*, *table_name*, *func*, *pos2=False*)
A tabulated atom feature. The feature is simply a table of N floating-point numbers, where N is the number of atoms in the system. This table is provided by a Python function, so can be used to implement user-defined features or to pass in features from other software. A simple example to use the x coordinate as a feature:

```
def func(aln, struc, mlib, libs):
    return [a.x for a in struc.atoms]
f = mdt.features.AtomTable(mlib, bins, "x coordinate", func)
```

1.5.8 Atom pair features

These features yield a single value for each pair of atoms in the first protein in each group of proteins selected from the alignment. See *Protein features* for a description of the common arguments.

class mdt.features.**AtomDistance**(*mlib*, *bins*)
Distance in angstroms between a pair of atoms. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**AtomBondSeparation**(*mlib*, *bins*, *disulfide=False*)
Number of bonds between a pair of atoms. For example, two atoms that are directly bonded return '1', while two at opposite ends of an angle return '2'. The bonds between atoms in each standard amino acid are derived from the bond class file, so this must be read in first (see *mdt.Library.bond_classes*). For atoms in different residues, the residues are assumed to be linked by a peptide backbone, and the number of bonds is calculated accordingly. Atoms in different chains, or atoms of types not referenced in the bond class file, are not connected. If disulfide is set to True, disulfide bridges are also considered (if two residues have SG atoms within 2.5 angstroms, they are counted as bonded). If disulfide is set to False (the default) any disulfide bridges are ignored. Either way, no account is taken of patches and other modifications such as terminal oxygens (unless bonds to OXT are explicitly listed in the bond class file). If a pair of atoms is not connected it is placed in the 'undefined' bin.

1.5.9 Tuple features

These features yield a single value for each tuple of atoms in the first protein in each group of proteins selected from the alignment. (The set of tuples must first be read into the *mdt.Library*.) Each feature takes some common arguments:

- *mlib*: the *mdt.Library* to create the feature in.
- *pos2*: if True, force a tuple pair scan, and evaluate the feature on the second tuple in each pair.

class mdt.features.**TupleType**(*mlib*, *pos2=False*)
Type of an atom tuple, as classified by the tuple class file. See *mdt.Library.tuple_classes*.

1.5.10 Tuple pair features

These features yield a single value for each pair of tuples of atoms in the first protein in each group of proteins selected from the alignment. (The set of tuples must first be read into the *mdt.Library*.) See *Protein features* for a description of the common arguments.

class mdt.features.**TupleType**(*mlib*, *pos2=False*)
Type of an atom tuple, as classified by the tuple class file. See *mdt.Library.tuple_classes*.

class mdt.features.**TupleDistance**(*mlib*, *bins*)
Distance in angstroms between the first atom in each of two tuples The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**TupleAngle1**(*mlib*, *bins*)
Angle (0-180) between the first atom in the first tuple, the first atom in the second tuple, and the second atom in the second tuple. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**TupleAngle2**(*mlib*, *bins*)
Angle (0-180) between the second atom in the first tuple, the first atom in the first tuple, and the first atom in the second tuple. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**TupleDihedral1**(*mlib*, *bins*)
Dihedral (-180-180) between the second atom in the first tuple, the first atom in the first tuple, the first atom in the second tuple, and the second atom in the second tuple. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**TupleDihedral2**(*mlib*, *bins*)
Dihedral (-180-180) between the third atom in the first tuple, the second atom in the first tuple, the first atom in the first tuple, and the first atom in the second tuple. Only works for atom triplets. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**TupleDihedral3**(*mlib*, *bins*)
Dihedral (-180-180) between the first atom in the first tuple, the first atom in the second tuple, the second atom in the second tuple, and the third atom in the second tuple. Only works for atom triplets. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

1.5.11 Chemical bond features

These features yield a single value for each defined chemical bond, angle or dihedral in the first protein in each group of proteins selected from the alignment. (The definitions of the chemical connectivity must first be read from a bond class file; see the *bond_clases*, *angle_classes* and *dihedral_classes* attributes in *mdt.Library*.) See *Protein features* for a description of the common arguments.

class mdt.features.**BondType**(*mlib*)
Type of a bond, as classified by the bond class file. See *mdt.Library.bond_classes*.

class mdt.features.**AngleType**(*mlib*)
Type of an angle, as classified by the angle class file. See *mdt.Library.angle_classes*.

class mdt.features.**DihedralType**(*mlib*)
Type of a dihedral, as classified by the dihedral class file. See *mdt.Library.dihedral_classes*.

class mdt.features.**BondLength**(*mlib*, *bins*)
Length of a bond in angstroms. See *mdt.Library.bond_classes*. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**Angle**(*mlib*, *bins*)
Angle (0-180). See *mdt.Library.angle_classes*. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

class mdt.features.**Dihedral**(*mlib*, *bins*)
Dihedral angle (-180-180). See *mdt.Library.dihedral_classes*. The feature is considered undefined if any of the atom coordinates are equal to the Modeller 'undefined' value (-999.0).

1.5.12 Group features

These features are used to make combinations of other features. Each feature takes some common arguments:

- *mlib*: the *mdt.Library* to create the feature in.
- *feat1*: an existing feature object that will be included in this group.
- *feat2*: another existing feature object to include.
- *nbins*: the number of bins in this feature.

class mdt.features.**Cluster**(*mlib*, *feat1*, *feat2*, *nbins*)
Cluster feature. When evaluated, it evaluates the two other features grouped in this feature, and converts the pair of bin indices for those features into a single bin index, which is returned. Use the *add()* method to control this conversion.

add(*child_bins*, *bin_index*)
Add a single mapping from a pair of child feature bin indices into this feature's bin index (all indexes start at 0). For example, calling *add((1,2), 3)* would cause this Cluster feature to return bin index 3 if the child features were in bins 1 and 2 respectively. This method can be called multiple times (even for the same *bin_index*) to add additional mappings from child bin indices to bin index. If no mapping from a given pair of child indices is present, the undefined bin index is returned.

1.6 Copyright and license

1.7 MDT change history

1.7.1 MDT 5.5 04-07-2020

- Add more Python 3 support to the build system.

1.7.2 MDT 5.4 05-25-2017

- Development is now open and hosted at GitHub.
- Experimental cmake build support.
- Installation packages now available for Homebrew ("brew tap salilab/salilab; brew install mdt") and Anaconda Python ("conda install -c salilab mdt").

1.7.3 MDT 5.3 05-19-2015

- A new function *write_statpot()* can generate a Modeller statistical potential file given a suitable input table.
- A new feature *Cluster* allows clustering of two features into a single one.
- A new feature *AtomTable* takes as input a table of precalculated per-atom values, and can be used to implement user-defined features or to use externally-calculated properties.
- To save space, the data for the MDT table itself can be compressed when writing to an HDF5 file with *Table.write_hdf5()*.
- Certain library information (such as the mapping from feature values to bin indices, and atom or tuple class information) and information about the last scan is now written into MDT files in HDF5 format by *Table.write_hdf5()*.
- The maximum value of bond_span_range can now be -1, to allow atom pairs that have no path of bonds between them. This is helpful to include inter-chain interactions, for example.

1.7.4 MDT 5.2 10-29-2012

- A new method *Table.get_array_view()* allows the raw MDT table data to be modified using NumPy functions.
- Disulfide bonds can now be considered in the calculation of atom bond separation, by the *AtomBondSeparation* feature and the *Table.add_alignment()* method.
- Atoms in atom tuples can now be restricted to match only in certain residue types.

1.7.5 MDT 5.1 09-29-2011

- All atom features, with the exception of *AtomType*, are now considered undefined if the atom coordinates are equal to the Modeller undefined value (-999.0).
- Support for bond separation, with a new *AtomBondSeparation* feature and a bond_span_range argument to *Table.add_alignment()*.
- Support Python 3 (requires Modeller 9.10 or later).

- "scons test" now reports the Python coverage (and also C coverage, if using gcc and adding "coverage=true" to the scons command line).
- The *Table* constructor now takes an optional 'shape' argument, which acts identically to that accepted by *Table.reshape()*.

1.7.6 MDT 5.0 03-31-2011

- First open source (GPLv2) release.
- Duplicated Modeller Fortran code removed; MDT now uses Modeller itself for handling of protein structures and alignments.
- Added scans over atom pairs, atom tuples, atom tuple pairs, and chemical bonds.
- Complete documentation, examples, and unit tests added.
- TOP scripting interface replaced with Python.
- Support storing MDT tables in binary form, using the HDF5 format and library.

1.7.7 MDT 4.0 April 2002

- Reorganize directory structure.

1.7.8 MDT 3.1 March 2002

- Allow for a structure to be assessed against an existing MDT table.

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

m

Index

I

L

M

N

O

P

R

S

T

U

V

W

X