Msys Documentation

Release 1.7.300

D.E. Shaw Research

May 17, 2020

CONTENTS

1 Overview **3**

1.1 Molecular Structures . . . 3
1.2 Forcefields . . . 4
1.3 Reading and Writing Files . . . 4

2 Atom selections **7**

2.1 Grammar . . . 7
2.2 Differences with VMD . . . 8
2.3 Smarts pattern selections . . . 11
2.4 Parameter type selections . . . 11
2.5 Comparison selections . . . 11
2.6 User-defined keywords . . . 12
2.7 User-defined atom selection macros . . . 12

3 Python Scripting **13**

3.1 Overview . . . 13
3.2 The msys module . . . 17
3.3 Pfx . . . 43
3.4 Molfile . . . 46
3.5 AnnotatedSystem . . . 51
3.6 SmartsPattern . . . 52

4 Nonbonded parameters **55**

4.1 Alternative nonbonded tables . . . 55
4.2 Overriding nonbonded interactions . . . 56
4.3 Alchemical nonbonded interactions . . . 56
4.4 NonbondedInfo . . . 56

5 Command line tools **57**

5.1 Conversion . . . 57
5.2 Information . . . 57
5.3 Basic Manipulation . . . 57
5.4 Structure building . . . 58
5.5 Validation . . . 59

6 Recipes **61**

6.1 Obtaining force-field parameters for certain atoms . . . 61
6.2 Adding artificial bonds . . . 61
6.3 Adding energy groups . . . 62
6.4 Remove selected constraints . . . 62
6.5 Canonicalize position restraint terms . . . 63

6.6 Processing multi-entry files (e.g. SDF files) . . . 64
6.7 Processing large SDF files . . . 65
6.8 Change the mass of selected atoms . . . 65

7 DMS Files **67**
7.1 Overview . . . 67
7.2 Chemical Structure . . . 68
7.3 Forcefields . . . 70
7.4 Alchemical systems . . . 80
7.5 References . . . 81

8 Release notes **83**

Bibliography **105**

Python Module Index **107**

Index **109**

Msys is a library for inspecting and manipulating chemical structures of the sort used in molecular simulations. Its main features include:

- A hierarchical organization of structure information, as opposed to the flat tables typically used in molecular file formats;
- A powerful VMD-like atom selection language;
- Representation of molecular forcefields which supports sharing of parameters by many groups of terms;
- Conversion to and from many chemical file formats, with forcefield and chemical information preserved where possible;
- Command line tools for scripting common tasks;
- C++ and Python interfaces.

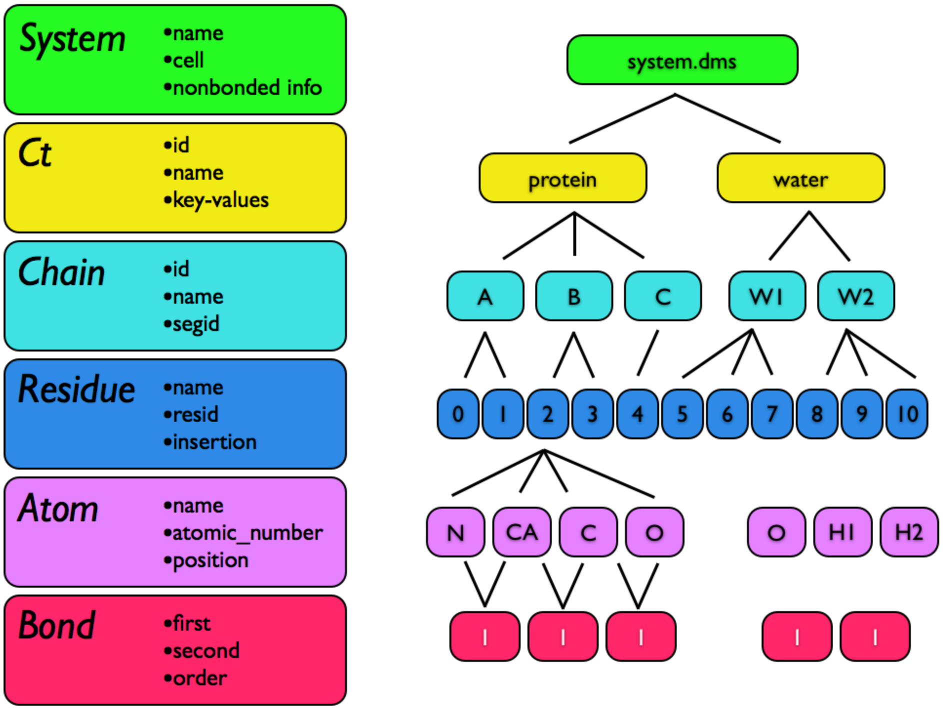
Contents:

CHAPTER ONE

OVERVIEW

This section describes how msys represents particles and forcefields, and the relationship between the msys representation and the various chemical file formats msys supports.

1.1 Molecular Structures


System
•name
•cell
•nonbonded info
Ct
•id
•name
•key-values
Chain
•id
•name
•segid
Residue
•name
•resid
•insertion
Atom
•name
•atomic_number
•position
Bond
•first
•second
•order
system.dms
protein
water
A
B
C
W1
W2
0
1
2
3
4
5
6
7
8
9
10
N
CA
C
O
O
H1
H2
1
1
1
1
1

All molecular structure in Msys is held in an object called a *System*. Within a *System*, individual particles, including physical atoms as well as pseudo-particles used as interaction sites, are represented as *Atoms*. Bonds between *Atoms* are represented by a *Bond* structure. *Atoms* are grouped into *Residues*, and *Residues* are grouped into *Chains*. Finally, *Chains* are grouped into higher-level structures called “components”, or *Cts* for short.

Every structure type is contained within its parent type; thus, even a single particle *System* will contain (at least) one *Residue*, *Chain*, and *Ct*. If there is no information in the file to delineate separate *Residues*, *Chains*, etc., then only a single such entity will be created.

There are also several other miscellaneous tables in each *System*:

- The *cell* holds the periodic cell information. It consists of three vectors, each with three components.
- The *nonbonded_info* structure holds meta-information about the type of nonbonded interactions used in the forcefield.
- There may be one or more auxiliary tables, indexed by name, which hold arbitrary additional forcefield data or other user-defined tables. These are indexed by name. The main use for auxiliary tables is to hold "cmap"-style tables from Charmm-style forcefields.

The *Ct* is the highest level of molecular organization, after the *System*. Many file formats, including MAE, SDF, etc., contain multiple structures, and it can be convenient to represent the entire contents of such a file in a single msys *System* without losing the distinction between structure records. When msys loads such a multi-component file, each entry gets placed in its own *Ct*. Another use for the *Ct* objects is when one *System* is appended to another. If there were no *Ct* objects, then *Chains* in one system might be unintentionally combined with *Chains* in the other system if the *Chains* had the same name. Finally, *Ct* blocks provide a space for arbitrary metadata about system components to be stored.

Chains in msys represent collections of *Residues*. Their main purpose is to hold the traditional chain and segment name information used in popular formats such as PDB.

Chains have just two settable properties: *name* and *segid*. When loading chemical systems, *Residues* are grouped into *Chains* entities based on their chain name and/or segid in the file, whichever is applicable.

A *Residue* in msys is a collection of *Atoms*. *Residues* have three settable attributes: *name*, *resid*, and *insertion*.

Finally, the *Atom* class represents all particles in the *System*, including real atoms as well as virtual and dummy particles. Each *Atom* has an atomic number, position, mass, and a number of other built-in properties.

1.2 Forcefields

A *System* also holds a set of *TermTables* representing the interactions between *Atoms*. A *TermTable* can be thought of as a particular kind of interaction; for example, a fully parameterized system would likely contain a `stretch_harm` *TermTable* to represent two-body covalent bond forces. Each *Term* in a *TermTable* refers to the same number of atoms, though there can be any number of *Terms* in a given *TermTable*.

Typically, many of the interactions in a *TermTable* are parameterized using identical parameters, especially when there are many identical copies of the same molecule in the *System*. For compactness, and also for ease of forcefield parameterization, a *TermTable* holds a separate table called a *ParamTable* which contains the interaction properties that can be shared by many *Terms*. Changes to an entry in a *ParamTable* will affect the interaction strengths of every *Term* referencing that entry. However, as illustrated below, operations on an individual *Term* will affect the interaction properties of just that *Term*; behind the scenes, Msys takes care of creating a copy of a *Term*'s parameters as needed.

It is also possible for developers to construct multiple *TermTables* that share the very same *ParamTable*, so that changes to a shared *ParamTable* affect multiple *TermTables* or *Systems*.

1.3 Reading and Writing Files

Msys reads and writes many popular chemical file formats. While most file formats have some concept of particles, residues, and chains, the way in which these groupings are specified varies by file type. Even within a file type, groupings are not always done consistently; for example, a PDB file might have both segment and chain identifiers, and there is no requirement in the file that there be any relationship between them.

In addition, many chemical file formats, including MAE, MOL2, SDF, as well as DMS, can contain multiple, logically distinct chemical groups or components. In some contexts, such as an MD simulation, it makes sense to consider all the components as part of a single system. In other contexts, such as processing a large batch of ligand structures, one wants to consider the components one at a time.

Forcefield information is also present in different file types in widely disparate forms. If forcefield information is read in one format and written out in another, it must be done with minimal loss of precision.

1.3.1 Mapping of residues and chains

As mentioned earlier, Msys groups all *Atoms* into *Residues*, and all *Residues* into *Chains*. This hierarchy is, unfortunately, rarely made explicit in the chemical system files in wide use, so Msys must infer the grouping based on the values of certain particle attributes.

Msys uses the `chain` and `segid` particle properties to group *Residues* into *Chains*. Within a chain, *Atoms* are grouped into *Residues* based on their `resname` and `resid` attributes. Thus, in Msys, every *Atom* within a given *Residue* has by definition the same `resname` and `resid`. By the same token, every *Atom* and *Residue* within a given *Chain* has the same `chain` and `segid`.

Upon loading a system, the number of *Chains* will be given by the number of distinct `chain` and `segid` pairs appearing in the particle table, and, within a given *Chain*, the number of *Residues* will be given by the number of distinct `resname` and `resid` pairs appearing in atoms sharing the *Chain's* `chain` and `segid`. After loading a system, one is free to modify the `resname` and `resid` of any *Residue*. Bear in mind, however, that if two initially distinct *Residues* in the same *Chain* come to have identical `resname` and `resid`, they will be merged into a single *Residue* upon saving and loading.

1.3.2 Whitespace in atom, residue and chain names

The PDB file format specifies that atom and residue names should be aligned to particular columns within a 4-column region. Unfortunately, some have taken this alignment requirement to mean that an atom's name actually includes the surrounding whitespace! When Msys loads a chemical system, the following fields are stripped of leading and trailing whitespace before they are inserted into the structure: `name` (atom name), `resname` (residue name), `chain` (chain identifier), and `segid` (segment identifier).

CHAPTER TWO

ATOM SELECTIONS

Msys implements an atom selection language similar to that of VMD. From Python, the language may be used to select atoms or clone subsets of a `System`, using the `select`, `selectIds`, or `clone` methods of `System`. In C++, the `Atomselect` function provides all atom selection functionality.

2.1 Grammar

The Msys atom selection grammar supports several primitive types which can be combined in various ways using logical operators. Primitive types include the following:

- Keyword selections: an attribute followed by one or more values, ranges, or regular expressions:

```
name CA               # a single value
resid 10 20 30        # multiple values
resid 10 to 30        # a range of values
name "C.*"            # regular expression, recognized by double quotes
resid 5 8 to 10       # combination of single value and a range
```

- Singleword selections: A boolean attribute. This includes both built-in singlewords such as `protein`, as well as those defined as macros, such as `acidic`

```
protein               # selects atoms identified as protein
acidic                # expands to 'resname ASP GLU
```

- String functions: these are similar in form to keyword selections, but they use their arguments in special ways:

```
smarts 'c[$(c[Ox1])]c'  # a smarts query.
paramtype nonbonded HP  # query on the nonbonded type of a particle
```

- Comparisons: An inequality formed by two expressions, at least one of which should be a function of atom attributes:

```
x > 0                 # atoms in the positive x halfspace
```

From these primitive selections, more complex selections may be formed using the following constructs.

- Boolean operators: AND, OR, NOT:

```
protein and not hydrogen    # heavy protein atoms
not oxygen and water        # same as "(not oxygen) and water"
not oxygen or water         # same as "(not oxygen) or water"
```

- SAME `attribute` AS `(selection)`:

```
same residue as name CA      # residues containing a CA atom
```

- `Within` selections of various flavors:

```
within 3 of protein          # atoms within 3 of protein, including protein
exwithin 3 of protein        # atoms within 3 of protein, but not protein
pbwithin 3 of protein        # uses minimum image distance
withinbonds 2 of resid 10    # atoms within two bonds of resid 10
```

- `Nearest selections:`

```
nearest 10 to residue 2      # nearest 10 atoms to any atom in residue 2
pbnearest 10 to residue 2    # same, but with minimum image distance
```

In comparisons, expressions can be formed in the following ways.

- Numeric literals, and keywords with numeric types:

```
x < 3                        # atoms whose x coordinate is less than 3
```

- Composition of expressions using various mathematical operators, parentheses, and functions:

```
x + y * z < 3                # the usual precedence rules apply
sqr(x)/36 + sqr(z)/125 < 1   # an ellipsoidal cylinder
```

2.2 Differences with VMD

Although the atom selection language in msys is similar to VMD's, there are some important differences to bear in mind if you switch between them:

- Element matching: In Msys, the atom selections "carbon", "hydrogen", "oxygen", etc. are based on the atomic number of the atoms. In VMD, these atom selections are computed using a regular expression based on the atom name:

```
vmd > atomselect macro oxygen name "O.*"

vmd > atomselect macro hydrogen name "[0-9]?H.*"

vmd > atomselect macro nitrogen name "N.*"

vmd > atomselect macro carbon name "C.*" and not ion
```

- Implicit 'and': in VMD, selections can sometimes be concatenated with an implicit 'and'; e.g. "water within 3 of protein" will be parsed by VMD as "water and within 3 of protein". In Msys, omitting the 'and' will result in a parse error.
- Field size: DMS and MAE files can hold chain, segment, and residue names of arbitrary length. In Msys, these values are used as-is. In VMD, the values are truncated; in particular, chain will be truncated to a single character in VMD, but not by Msys.
- Data representation: Msys has no concept of secondary structure, so the "sheet", helix", etc. atom selection keywords are not implemented in msys.
- Floating-point roundoff: There may occasionally be differences in the results of distance based atom selections simply due the fact that Msys stores positions as doubles, while VMD stores them as floats.

2.2.1 Built-in selections

The following selection keywords are available:

keyword	type	definition
atomicnumber	integer	*Atom*.atomic_number
element	string	Abbreviation for element *Atom*.atomic_number
chain	string	*Chain*.name
segid	string	*Chain*.segid
charge	float	*Atom*.charge
fragment	integer	Connected residues will all have the same fragment id, except that the connection check will not follow disulfide bridges, identified as atoms whose name is “SG”.
index	integer	*Atom*.id
mass	float	*Atom*.mass
name	string	*Atom*.name
numbonds	integer	*Atom*.nbonds - includes bonds to pseudoatoms
degree	integer	number of bonds to real atoms; 0 for pseudoatoms
resid	integer	*Residue*.resid
residue	integer	*Residue*.id
resname	string	*Residue*.name
fragid	integer	*Atom*.fragid. Connnected atoms will all have the same fragid.
x, y, z	float	*Atom*.x, *Atom*.y, *Atom*.z, the position.
vx, vy, vz	float	*Atom*.vx, *Atom*.vy, *Atom*.vz, the velocity.

The following selection singlewords are available.

singleword	definition
all	Every atom.
none	No atoms.
water	Atoms belonging to a residue containing the atomic number and bond structure of water, as well as those residues whose residue name is one of the following: “H2O”, “HH0”, “OHH”, “HOH”, “OH2”, “SOL”, “WAT”, “TIP”, “TIP2”, “TIP3”, “TIP4”, “SPC”.
hydrogen	atomic number 1
backbone	This singleword includes both protein backbone as well as nucleic backbone. Protein backbone is identified by searching for atoms named “CA”, “C”, “O”, and “N” in the same residue, and for atoms named “OT1”, “OT2”, “OXT”, “O1”, or “O2” that are bonded to one of the members of the first list. If at least four such atoms are found, those atoms are identified as backbone. Similarly, nucleic acid backbone atom names are P”, “O1P”, “O2P”, “OP1”, “OP2”, “C3*”, “C3’”, “O3*”, “O3’”, “C4*”, “C4’”, “C5*”, “C5’”, “O5*”, or “O5’”; or atoms named “H5T” or “H3T” bonded to a member of the first set. At least four such atoms must be found in the same residue in order to be identified as backbone.
protein	residues containing protein backbone atoms.
nucleic	residues containing nucleic backbone atoms.

The following are implemented as macros.

macro	definition
at	resname ADE A THY T
acidic	resname ASP GLU
cyclic	resname HIS PHE PRO TRP TYR
acyclic	protein and not cyclic
aliphatic	resname ALA GLY ILE LEU VAL
alpha	protein and name CA
amino	protein
aromatic	resname HIS PHE TRP TYR
basic	resname ARG HIS LYS HSP
bonded	degree > 0
buried	resname ALA LEU VAL ILE PHE CYS MET TRP
cg	resname CYT C GUA G
charged	basic or acidic
hetero	not (protein or nucleic)
hydrophobic	resname ALA LEU VAL ILE PRO PHE MET TRP
small	resname ALA GLY SER
medium	resname VAL THR ASP ASN PRO CYS ASX PCA HYP
large	protein and not (small or medium)
neutral	resname VAL PHE GLN TYR HIS CYS MET TRP ASX GLX PCA HYP
polar	protein and not hydrophobic
purine	resname ADE A GUA G
pyrimidine	resname CYT C THY T URA U
surface	protein and not buried
lipid	resname DLPE DMPC DPPC GPC LPPC PALM PC PGCL POPC POPE

Table 1 – continued from previous pag

macro	definition
lipids	lipid
legacy_ion	resname AL BA CA Ca CAL CD CES CLA CL ‘Cl-‘ Cl CO CS CU Cu CU1 CUA HG IN IOD K ‘K+’ MG MN3 MC
ion	degree 0 and not atomicnumber 0 1 2 5 6 7 8 10 18 36 54 86
ions	ion
sugar	resname AGLC
solvent	not (protein or sugar or nucleic or lipid)
carbon	atomicnumber 6
nitrogen	atomicnumber 7
oxygen	atomicnumber 8
sulfur	atomicnumber 16
noh	not hydrogen
heme	resname HEM HEME

2.3 Smarts pattern selections

A SMARTS pattern is like a regular expression for molecular structures; it’s a concise way of specifying what sort of atom types and topology you are looking for. SMARTS patterns can be embedded in an atom selection by providing the keyword ‘smarts’ followed by one or more SMARTS patterns, which you will need to surround in single quotes if it contains any special characters like parentheses:

```
# select benzene rings
mol.select("smarts 'c1ccccc1'")
```

See the description of the `Smarts` class for more information.

2.4 Parameter type selections

If a ParamTable contains a column named ‘type’, you can query for atoms which participate in an interaction involving that type using the ‘paramtype’ keyword. For example:

```
# select atoms whose nonbonded type is 'H1'
mol.select("paramtype nonbonded H1")
```

2.5 Comparison selections

Comparisons are formed from two expressions and a binary comparison operator. The available comparison operators are the usual inequality and equality operators: <, >, <=, >=, ==, and `!=`. Expressions can be built up from numeric literals and from keywords of float type, in the following ways:

- Binary mathematical operators: `+`, `-`, `*`, and `/`; e.g., “x * y - z < 3”.
- The C-style modulus function `%`; e.g., “residue % 10 == 0” for every 10th residue.
- Unary `-`.
- The functions `sqr`, `sqrt`, and `abs`; e.g., “sqrt(sqr(x)+sqr(y))<5”.

2.6 User-defined keywords

In addition to the aforementioned built-in keywords, any atom property may also be used as an atom selection keyword. For example:

```
# add atom property 'foo' to a system.  The default value is empty string
mol.addAtomProp('foo', str)

# set the foo property to 'jrg' for all alpha carbons
for a in mol.select('name CA'): a['foo'] = 'jrg'

# check that selecting for foo equal to jrg is equivalent to 'name CA'
assert mol.select('foo jrg') == mol.select('name CA')
```

2.7 User-defined atom selection macros

This feature was removed in msys 1.7.7.

CHAPTER

THREE

PYTHON SCRIPTING

Most of the functionality in msys is exposed in its Python interface.

3.1 Overview

This section introduces the Python interface and explains how to use it effectively. We begin by introducing some concepts that pervade the Python interface, then move to some examples.

3.1.1 Msys ids

In Msys, instances of the *Atom*, *Bond*, *Residue*, and *Chain* classes are all *Handles*, in the sense that they refer to a piece of data held by the parent *System*. All Msys handles have an immutable `id` property that uniquely identifies them within their parent *System*. Objects that hold references to other objects do so through the `id` of that object. Two handles of the same type will compare equal to each other if and only if they belong the same *System* and possess the same `id`.

When you load a system from a file, or create one from scratch, these `ids` will be numbered consecutively, starting at zero. Deleting *Atoms*, *Bonds*, etc. from the *System* can introduce gaps in the set of `ids`, but, once created, the `id` of an object never changes.

When Msys writes a DMS file, the primary keys of the particles will be contiguous starting at 0, and will appear in the order in which the particles appear in the *System*, even if the `ids` of the atoms in the *System* are noncontiguous due to deletions. When Msys loads a DMS file, if the primary keys happen to be noncontiguous, Msys will still create a *System* with the usual contiguous ids.

3.1.2 Msys properties

Many objects in Msys (in particular, *Atoms*, *Bonds*, *Terms*, and *Params*) can have typed attributes given to all members of the set to which they belong. In Msys, these attributes are referred to as *properties*, or *props* for short, and have a type of either *int*, *float*, or *str* (string). The available property names and their types can be queried in the appropriate parent object, using the `props`, `atom_props`, etc. properties of the parent. The value of the property for a given element can be read and modified using a dictionary-like interface on the element itself:

```python
mol = msys.LoadDMS('input.dms')
# find all distinct values of the 'grp_energy' atom property, if it exists
grp_energy_vals = set()
if 'grp_energy' in mol.atom_props:
  for atm in mol.atoms:
    grp_energy_vals.add( atm['grp_energy'] )
```

(continues on next page)

(continued from previous page)

```
# add a new property 'foo' of type 'float'
mol.addAtomProp('foo', float)
# Set the value of foo to the z coordinate of the atom
for a in mol.atoms: a['foo'] = a.pos[2]
```

When you add a property to a set of elements, the initial value will be 0 for *int* and *float* types, and the empty string for *str* types. If a property with the same name and type already exists, no action is taken. An exception is thrown if you try to add a property with the same name but different type from an existing property.

3.1.3 Getting started

Once you have your Python environment by loading the appropriate modules, fire up Python, import the msys module, and load a dms or mae file:

```
import msys

# Load the entire contents of a DMS file
dms=msys.LoadDMS('system.dms')

# Import an MAE file, performing conversions on its forcefield data
mae=msys.LoadMAE('system.mae')
```

You can also create a new *System* from scratch:

```
# mol = msys.CreateSystem()
```

A *System* resides entirely in memory; changes to the *System* will not persist until/unless you write it back out to a file:

```
# Save the system as a DMS file
msys.SaveDMS(dms, 'output.dms')

# Export to MAE file
msys.SaveMAE(dms, 'output.mae')
```

Msys also lets you append chemical systems to an existing file, for certain file formats. The supported Save methods will have an 'append' option in their function signatures.

The full set of *Atoms*, *Bonds*, *Residues*, *Chains*, and *TermTables* are available by fetching them from the system. You can also fetch the bonds involving a particular atom, the atoms in a residue, or the bonds in a chain in a similar way:

```
# get the number of atoms, and the total charge
atoms = dms.atoms
natoms = len(atoms)
total_charge = sum(a.charge for a in atoms)

# find the atoms participating in double bonds
for chn in mol.chains:
  for res in chn.residues:
    for atm in res.atoms:
      for bnd in atm.bonds:
        if bnd.order == 2:
          print "atom %d in chain %s residue %s:%d has a double bond" % (
            atm.id, chn.name, res.name, res.num)

# iterate over tables, print atoms per term and number of terms
```

(continues on next page)

(continued from previous page)

```
for t in mol.tables:
  print "table %s: %d atoms, %d terms" % (t.name, t.natoms, t.nterms)

# fetch the stretch_harm table.  Throws an exception if no such table
stretch = mol.table('stretch_harm')
```

Atom selections let you fetch a list of atoms using the VMD atom selection language. The `select` method returns a list of *Atoms*, which is just a subset of the list that would be returned by the `atoms` property:

```
# fetch the backbone atoms.  Note that bb is just a Python list
bb = mol.select('backbone')
```

Once you have the atoms, if you actually want to work with the residues or chains, it's easy to do:

```
# get the set of distinct residues in the backbone
bb_residues = set(a.residue for a in bb)
```

Note that the atoms returned by `select` refer back to the original system. Msys also provides the means to create a new *System* independent of the original, using either the `CloneSystem` function or the `clone` method of System. When you clone a subset of a *System*, the *Terms* in the forcefield whose atoms are completely contained in the selection will be copied to the new *System*:

```
# get the list of protein atoms
pro_atoms = mol.select('protein')

# construct a new system containing just the protein
protein = msys.CloneSystem(pro_atoms)

# Atoms in the cloned system have the same attributes as the originals,
# but modifications to one do not affect the other
assert pro_atoms[0].charge == protein.atoms[0].charge
pro_atoms[0].charge += 3
assert pro_atoms[0].charge != protein.atoms[0].charge
```

The `clone` method of *System* is a more concise way of selecting a set of atoms, then immediately creating a new *System* from it:

```
# create a new System with all the hydrogens removed
hless = mol.clone('not hydrogen')

# create a copy of the original
dup = mol.clone()
```

You can append the structure and associated forcefield from one *System* onto another using System's `append` method:

```
# duplicate the protein by appending to itself
protein.append(protein)

# load a water system and append it to the protein system.  Just as for
# CloneSystem, after appending water to protein, modifications to water
# will not affect any atoms in protein.
water = msy.LoadDMS('water.dms')
protein.append(water)
```

Terms in a system's forcefield can be accessed and modified by going through the corresponding *TermTable*:

```
stretch = protein.table('stretch_harm')
terms = stretch.terms
params = stretch.params
props = params.props # ['fc', 'r0']
print "%d stretch terms, %d stretch params" % (terms.nterms, params.nparams)
```

You can change the properties of a selected *Term* using a dictionary-like interface:

```
# Change the force constant of the first stretch term to 42
stretch.terms[0]['fc] = 42
```

3.1.4 Adding new forcefield terms

Msys provides an interface for adding a *TermTable* corresponding to a “standard” forcefield term and configuring that table with its category and its the expected set of properties:

```
# Get the available set of TermTable schemas:
schemas = msys.TableSchemas()

# For bonded, constraint, virtual, and polar terms, as well as
the exclusion table:
table = mol.addTableFromSchema('posre_harm')  # position restraints

# Get the available set of nonbonded schemas
nb_schemas = msys.NonbondedSchemas()

# For a nonbonded table:
nb = mol.addNonbondedFromSchema('vdw_12_6')
```

The `addNonbondedFromSchema` also takes care of configuring the `nonbonded_info` properties of the *System*; see the section on nonbonded parameters for more details.

If you have a new table type that hasn’t made it into Msys’ canonical set, you can simply use `addTable` and configure the table yourself:

```
table = mol.addTable('funky_harm', 2)
table.params.addProp('fk', float)
table.params.addProp('r0', float)
```

If a table with a given name already exists in a *System*, `addTable` and `addTableFromSchema` will just return the existing table.

3.1.5 Files with multiple components

To examine every structure in a multi-component file without having to load them all into memory at once, use **LoadMany**. Unlike the **Load** function, which always returns one System, **LoadMany** is a generator which iterates over molecular structures in the input file:

```
for mol in msys.LoadMany('input.mol2'):
   print mol.name
```

Not every file format supports LoadMany; in cases where it doesn’t, LoadMany will stop after a single iteration, yielding just one *System*.

If you use LoadMany to load a file, each *System* will have only one *Ct*. However, if you use Load to import an MAE or DMS file, and the file contains multiple components, the new *System* will contain *Ct* elements corresponding to those components:

```
mol = msys.Load('small_vancomycin_complex.mae')
for ct in mol.cts:
   print ct.name

# prints:
# vancomycin_diala_complex
# SPC water box
```

The ct information wil be preserved when saving the System back to an MAE or DMS file.

You can create a multi-ct system from existing *Systems* using the `append` method:

```
pro = msys.Load('protein.dms')
pro.ct(0).name = 'protein'
wat = msys.Load('water.dms')
wat.ct(0).name = 'water'
pro.append(wat)
assert pro.ncts == 2     # assuming there was 1 ct in protein.dms and wat.dms
assert pro.ct(1).name == 'water'
msys.Save(pro, 'combined.dms')
```

3.2 The msys module

This is the high-level Python interface for msys, intended for use by chemists.

class `msys.`**`AnnotatedSystem`**(*sys*, *allow_bad_charges=False*)

System that has been annotated with additional chemical information

The AnnotatedSystem class provides chemical annotation useful primarily for evaluating smarts patterns. The system is expected to already have have chemical reasonable bond orders and formal charges, and to have no missing atoms (e.g. hydrogens). If these criteria cannot be met, set allow_bad_charges=True in the constructor to bypass these checks; in that case the AnnotatedSystem can still be used to evaluate smarts patterns, but patterns making use of the electronic state of the system (e.g. aromaticity, hybridization, etc.) will not be correct (the system will appear to be entirely aliphatic). You may also use the AssignBondOrderAndFormalCharge function to assign reasonable bond orders and formal charges, assuming there are no missing atoms.

The AnnotatedSystem defines a model for aromaticity. First, the SSSR (smallest set of smallest rings) is determined. Next, rings which share bonds are detected and grouped into ring systems. Rings are initially marked as nonaromatic. If the ring system taken as a whole is deemed to be aromatic, then all rings within it are aromatic as well; otherwise, individual rings are checked for aromaticity. Rings are checked in this fashion until no new rings are found to be aromatic.

A ring system is deemed to be aromatic if it satisfies Huckel's 4N+2 rule for the number of electrons in the ring(s). An internal double bond (i.e. a bond between two atoms in the ring) adds 2 to the electron count. An external double bond (a bond between a ring atom and an atom not in that ring) adds 1 to the electron count. An external double bond between a carbon and a nonaromatic carbon makes the ring unconditionally nonaromtic. An atom with a lone pair and no double bonds adds 2 to the electron count.

`__init__`(*sys*, *allow_bad_charges=False*)

Construct from System. AnnotatedSystem is not updated if System is subsequently modified.

`__repr__`()

Return repr(self).

__weakref__
list of weak references to the object (if defined)

aromatic(*atom_or_bond*)
Is atom or bond aromatic

degree(*atom*)
Number of (non-pseudo) bonds

property errors
List of errors found during system analysis if allow_bad_charges=True

hcount(*atom*)
Number of bonded hydrogens

hybridization(*atom*)
Atom hybridization – 1=sp, 2=sp2, 3=sp3, 4=sp3d, etc.

Equal to 0 for hydrogen and atoms with no bonds, otherwise max(1, a.degree() + (a.lone_electrons+1)/2 - 1).

loneelectrons(*atom*)
Number of lone electrons

ringbondcount(*atom*)
Number of ring bonds

valence(*atom*)
Sum of bond orders of all (non-pseudo) bonds

msys.**ApplyDihedralGeometry**(*a*, *b*, *c*, *r*, *theta*, *phi*)
Return the position of atom d with cd length r, bcd angle theta, and abcd dihedral phi, all in radians.

msys.**AssignBondOrderAndFormalCharge**(*system_or_atoms*, *total_charge=None*, *compute_resonant_charges=False*, ***, *timeout=60.0*)
Assign bond orders and formal charges to a molecular system.

Determines bond orders and formal charges by preferring neutral charges and placing negative charges with more electronegative atoms, under octet constraints and the total system charge constraint. Assigns the bond orders and formal charges to the system. Can assign to a subset of atoms of the system, provided these atoms form complete connected fragments.

Parameters

- **system_or_atoms** – either a System or a list of Atoms
- **total_charge** – if not None, integral total charge
- **compute_resonant_charges** (*bool*) – compute and store resonant charge in atom property ‘resonant_charge’ and resonant bond order in bond property ‘resonant_order’.
- **timeout** (*float*) – maximum time allowed, in seconds. Note: calling this function on a chemically incomplete system, i.e. just protein backbone, cause msys to hit the timeout.

class msys.**Atom**(*_ptr*, *_id*)
Represents an atom (or pseudoparticle) in a chemical system

__contains__(*key*)
does atom property key exist?

__getitem__(*key*)
get atom property key

__lt__(*that*)
Return self<value.

`__repr__`()
Return repr(self).

`__setitem__`(*key*, *val*)
set atom property key to val

`addBond`(*other*)
create and return a Bond from self to other

`property aromatic`

`property atomic_number`

`property bonded_atoms`
Atoms bonded to this atom

`property bonds`
Bonds connected to this atom

`property charge`

`findBond`(*other*)
Find the bond between self and Atom other; None if not found

`property formal_charge`

`property fragid`

`property mass`

`property name`

`property nbonds`
number of bonds to this atom

`property nhydrogens`
number of bonded hydrogens

`property pos`
position

`remove`()
remove this Atom from the System

`property valence`
sum of bond orders

`property vel`
velocity

`property vx`

`property vy`

`property vz`

`property x`

`property y`

`property z`

`class` `msys.`**`Bond`**(*_ptr*, *_id*)
Represents a bond in a System

`__contains__`(*key*)
does custom Bond property exist?

__getitem__(*key*)
get custom Bond property

__lt__(*that*)
Return self<value.

__repr__()
Return repr(self).

__setitem__(*key*, *val*)
set custom Bond property

property atoms
Atoms in this Bond

property first
first Atom in the bond (the one with lower id)

property order
bond order (int)

other(*atom*)
atom in bond not the same as given atom

remove()
remove this Bond from the System

property second
second Atom in the bond (the one with higher id)

exception msys.**BrokenBondsError**

__weakref__
list of weak references to the object (if defined)

msys.**CalcAngle**(*a*, *b*, *c*)
Angle in radians of atoms or positions a-b-c.

msys.**CalcDihedral**(*a*, *b*, *c*, *d*)
Dihedral angle in radians of atoms or positions a-b-c-d

msys.**CalcDistance**(*a*, *b*)
Distance between atoms or positions a and b

msys.**CalcPlanarity**(*pos_or_atoms*)
Planarity of positions or atoms

class msys.**Chain**(*_ptr*, *_id*)
Represents a chain (of Residues) in a System

__repr__()
Return repr(self).

addResidue()
append a new Residue to this Chain and return it

property ct
Return the Ct for this chain

property name

property nresidues
number of residues in this chain

remove()
remove this Chain from the System

property residues
list of Residues in this Chain

property segid

selectResidue(*resid=None*, *name=None*, *insertion=None*)
Returns a single Residue with the given resid, name, and/or insertion code. If no such residue is found, returns None. If multiple such residues are found within this chain, raises an exception.

msys.**CloneSystem**(*atoms*)
Call System.clone(atoms) using the System from the first atom.

DEPRECATED. Use System.clone directly instead.

msys.**ComputeTopologicalIds**(*system*)
Compute and return the topological ids for the atoms or system

msys.**ConvertFromOEChem**(*oe_mol*, *force=False*)
Construct a System from the given OEChem OEMol

Parameters

- **oe_mol** (*oechem.OEMol*) – the OEChem molecule to convert
- **force** (*bool*) – whether sanity checks should be performed before conversion

Returns System

Return type mol (*System*)

msys.**ConvertFromRdkit**(*rdmol*)
Construct an msys System from an RDMol :param mol: system :type mol: rdkit.ROMol

Returns System

Notes

All atoms will be assigned to the same Residue. Only the first conformer will be used, if any. There must not be any implicit hydrogens. Bonds will be kekulized since msys doesn't maintain aromaticity. Chiral tags will not be maintained.

msys.**ConvertToOEChem**(*mol_or_atoms*)
Construct an OEChem OEMol from the given System

Parameters **mol** – System or [Atoms]

Returns oechem.OEMol

Notes

If [Atoms] are give, only bonds involving the specified atoms will be passed to to the OEMol; this is the same behavior as System.clone(atoms).

msys.**ConvertToRdkit**(*mol*, *sanitize=True*)
Construct an RDKit ROMol from the given System

Parameters

- **mol** (System) – System

- **sanitize** (*bool*) – whether to sanitize the molecule

Returns rdkit.ROMol

Notes: alchemical systems may require sanitize=False

msys.**CreateParamTable**()
Create a new, empty ParamTable

msys.**CreateSystem**()
Create a new, empty System

class msys.**Ct**(*_ptr*, *_id*)
Represents a list of Chains in a System

The Ct class exists mainly to provide a separate namespace for chains. If you merge two systems each of which has a chain A, you probably want the chains to remain separate. Cts accomplish this.

The Ct class also provides a key-value namespace for assigning arbitrary properties to Systems.

__delitem__(*key*)
remove property key

__getitem__(*key*)
get ct property key

__setitem__(*key*, *val*)
set ct property key to val

addChain()
append a new Chain to this Ct and return it

append(*system*)
Appends atoms and forcefield from system to self. Returns a list of of the new created atoms in self. Systems must have identical nonbonded_info.vdw_funct. Does not overwrite the global cell information in self.

property atoms
list of Atoms in this Ct

property bonds
list of Bonds in this Ct

property chains
list of Chains in this Ct

get(*key*, *d=None*)
get ct property key, else d, which defaults to None

keys()
available Ct properties

property name
Name of Ct

property natoms
number of Atoms in the Ct

property nchains
number of Chains in this Ct

remove()
remove this Ct from the System

`msys.`**`FindDistinctFragments`**(*system*, *key='graph'*)

Find connected sets of atoms with identical topology.

Parameters

- **`system`** – System chemical system
- **`key`** – str one of 'graph', 'inchi', 'oechem_smiles', or list of strings, one per fragment

Returns

mapping from representative fragment id to ids of fragments having identical topology.

Return type dict[int -> [int]]

Notes

Fragments are distinguished according to value given by 'key'; if 'graph', topologically identical fragments will be considered identical even if they have different stereochemistry. Choose one of the other options to include stereochemistry in the fragment disambiguation.

`msys.`**`FormatDMS`**(*system*)

Return the DMS form of the system as bytes

`msys.`**`FormatJson`**(*system*)

Json formatted system (EXPERIMENTAL)

`msys.`**`FormatSDF`**(*mol*)

Return System in sdf format

`msys.`**`FromSmilesString`**(*smiles*, *forbid_stereo=True*)

Construct a System from a smiles string.

Parameters

- **`smiles`** (*`str`*) – the smiles string
- **`forbid_stereo`** (*`bool`*) – if True, raise exception if smiles has stereo

EXPERIMENTAL. In particular, stereo information in the smiles string is ignored. Set forbid_stereo=False to permit stereo specifications to be silently ignored. This flag may be removed at a later date once stereo support has been added.

`msys.`**`GetBondsAnglesDihedrals`**(*system*)

Return bonds, angles and dihedrals deduced from bond topology Returns: dict

`msys.`**`GetRingSystems`**(*atoms*)

Get ring systems for the given atoms

`msys.`**`GetSSSR`**(*atoms*, *all_relevant=False*)

Get smallest set of smallest rings (SSSR) for a system fragment.

The SSSR is in general not unique; the SSSR of a tetrahedron is any three of its four triangular faces. The set of rings that is the union of all SSSR's (all relevant rings) may be obtained by setting all_relevant to True.

Arguments: atoms – [msys.Atom, …, msys.Atom] from a single system all_relevant – bool Returns: [[msys.Atom, …, msys.Atom], …, [msys.Atom, …, msys.Atom]]

`class` `msys.`**`Graph`**(*system_or_atoms*, *colors=None*)

Represents the chemical topology of a System

Used mainly to implement graph isomorphism; see the match() method

`__init__`(*system_or_atoms*, *colors=None*)

Construct Graph :param system_or_atoms: System or [Atoms] :param colors: None, [Int] or Callable

If colors is provided, it is used to specify the vertex color for each atom in the graph. By default, atomic number is used. A color may be provided for each atom, or a callable accepting an Atom as argument. Atoms with color zero are ignored for purposes of graph matching, and no mapping will be returned for them.

`__weakref__`

list of weak references to the object (if defined)

`atoms`()

ordered atoms in graph

`hash`()

string hash of atoms and bonds in graph

`static hash_atoms`(*atoms*)

string hash of specified atoms

Parameters **`atoms`** (*[`msys.Atom`]*) – list of atoms

`match`(*graph*)

Find a graph isomorphism between self and the given Graph. If no isomorphism could be found, return None; otherwise return mapping from atoms in this graph to atoms in that graph.

`matchAll`(*graph*, *substructure=False*)

Find all graph isomorphisms between self and the given Graph. If no isomorphism could be found, return empty list; otherwise return list of dicts mapping atoms in this graph to atoms in that graph. If substructure is True, return isomorphisms between self and any subgraph of the given Graph.

`size`()

number of atoms in graph

`msys.`**`GuessHydrogenPositions`**(*atoms*)

Experimental

`class` `msys.`**`HydrogenBondFinder`**(*system*, *donors*, *acceptors*, *cutoff=3.5*)

Find candidate hydrogen bonds.

More hbonds will be found than are “realistic”; further filtering may be performed using the energy attribute of the returned hbonds. A reasonable filter seems to be around -1.0 (more negative is stronger); i.e. energies greater than that are more likely than not to be spurious.

The HydrogenBond class can also be used directly to compute hydrogen bond geometry and energies by supplying donor, acceptor and hydrogen positions.

`__init__`(*system*, *donors*, *acceptors*, *cutoff=3.5*)

Parameters

- **`system`** (`System`) – msys system
- **`donors`** – selection string, list of ids, or list of Atoms
- **`acceptors`** – selection string, list of ids, or list of Atoms
- **`cutoff`** (*`float`*) – distance cutoff for donor and acceptor

Note: If Atoms are provided, they must be members of system.

`__weakref__`
list of weak references to the object (if defined)

`find`(*pos=None*)
Find hydrogen bonds for the given positions, defaulting to the current positions of the input system.

class msys.**InChI**(*system*, *DoNotAddH=True*, *SNon=False*, *FixedH=True*)
InChI holds an the result of an inchi invocation for a structure

`__init__`(*system*, *DoNotAddH=True*, *SNon=False*, *FixedH=True*)
Initialize self. See help(type(self)) for accurate signature.

`__str__`()
Return str(self).

`__weakref__`
list of weak references to the object (if defined)

property auxinfo
Auxiliary info

property key
inchi key for this object's string.

property message
Message returned by inchi during calculation

property ok
Was an inchi computed successfully?

property string
Computed inchi string

class msys.**IndexedFileLoader**(*path*, *idx_path=None*)
Supports random access to multi-structure files

`__getitem__`(*index*)
Get structure at given index

Parameters **index** (`int`) – 0-based index

Returns msys System

Return type mol (*System*)

`__init__`(*path*, *idx_path=None*)
Open an indexed file loader, creating an index file if needed. :param path: file path. File type is inferred from the extension. :type path: str :param idx_path: index file path. Defaults to $path.idx. :type idx_path: str

Note: You need write permission to the location of the index file.

`__len__`()
number of entries

`__weakref__`
list of weak references to the object (if defined)

property path
path to source file

`msys.`**`LineIntersectsTriangle`**(*r*, *s*, *a*, *b*, *c*)

True if line segment rs intersects triangle abc

`msys.`**`Load`**(*path*, *structure_only=False*, *without_tables=None*)

Infer the file type of path and load the file.

Parameters

- **`path`** – if str, a file path or PDB accession code. if int, an Anton jobid (require 'yas' garden module)
- **`structure_only`** (`bool`) – Omit force tables and pseudo atoms
- **`without_tables`** (`bool`) – Omit force tables.

Returns new System

`msys.`**`LoadDMS`**(*path=None*, *structure_only=False*, *buffer=None*)

Load the DMS file at the given path and return a System containing it. If structure_only is True, only Atoms, Bonds, Residues and Chains will be loaded, along with the GlobalCell, and no pseudos (atoms with atomic number less than one) will be loaded.

If the buffer argument is provided, it is expected to hold the contents of a DMS file, and the path argument will be ignored.

`msys.`**`LoadMAE`**(*path=None*, *ignore_unrecognized=False*, *buffer=None*, *structure_only=False*)

load the MAE file at the given path and return a System containing it. Forcefield tables will be created that attempt to match as closely as possible the force terms in the MAE file; numerical differences are bound to exist. If ignore_unrecognized is True, ignore unrecognized force tables.

If the buffer argument is provided, it is expected to hold the contents of an MAE file, and the path argument will be ignored.

If the contents of the file specified by path, or the contents of buffer, are recognized as being gzip-compressed, they will be decompressed on the fly.

If structure_only is True, no forcefield components will be loaded.

`msys.`**`LoadMany`**(*path*, *structure_only=False*, *error_writer=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)

Iterate over structures in a file, if the file type supports iteration.

for mol in LoadMany('input.mol2'): ...

If there was an error reading a structure, LoadMany returns None for that iteration. If error_writer is not None, it's write() method is invoked with the contents of the exception message as argument. error_writer defaults to sys.stderr.

`msys.`**`LoadMol2`**(*path*, *multiple=False*)

Load a mol2 file at the given path. If multiple is True, return a list of Systems, one for each MOLECULE record. Otherwise, return just one System corresponding to the first MOLECULE record.

`msys.`**`LoadPDB`**(*path*)

Load a PDB file at the given path and return a System. No bonds will be created, even if CONECT records are parent.

`msys.`**`LoadPrmTop`**(*path*, *structure_only=False*)

Load an Amber7 prmtop file at the given path and return a System. Coordinates and global cell information are not present in the file.

`msys.`**`LoadXYZ`**(*path*)

Load an xyz file at the given path. Guesses bonds based on guessed atomic numbers based on atom name.

`msys.`**`MatchFragments`**`(`*mol1*, *mol2*, *key='graph'*`)`

construct an atom to atom mapping for all fragments from mol1 to mol2

Parameters

- **`mol1`** – System
- **`mol2`** – System
- **`key`** – see FindDistinctFragments

Returns dict[Atom -> Atom] or None

`msys.`**`NonbondedSchemas`**`()`

available nonbonded schemas for System.addNonbondedFromSchema

`class` `msys.`**`Param`**`(`*_ptr*, *_id*`)`

A *Param* instance is a reference to a row in a *ParamTable*. Use the `dict`-style interface to get and set values in the row. Msys will take care of converting input values to the type of the corresponding column, and raise an exception if the conversion cannot be performed.

`__getitem__``(`*prop*`)`

get the value of prop

`__repr__``()`

Return repr(self).

`__setitem__``(`*prop*, *val*`)`

update the value of prop with val

`duplicate``()`

create a new entry in the parent parameter table with the same values as this one, returning it.

`property id`

id in parent table

`keys``()`

sorted list of available properties

`property system`

parent System

`property table`

parent ParamTable

`class` `msys.`**`ParamTable`**`(`*_ptr*`)`

The *ParamTable* class is a 2d table, whose rows are indexed by `id` and whose columns are properties; see the discussion of properties in the Overview. A *ParamTable* is used by *TermTables* to hold the shared parameters for its *Terms*.

`__eq__``(`*x*`)`

Return self==value.

`__hash__``()`

Return hash(self).

`__init__``(`*_ptr*`)`

Initialize self. See help(type(self)) for accurate signature.

`__ne__``(`*x*`)`

Return self!=value.

`addParam``(`***kwds*`)`

add and return a new Param().

If keyword arguments are supplied, they will be assigned to the newly created Param before returning it.

`addProp`(*name*, *type*)
add a new property of the given type, which must be int, float, or str.

`delProp`(*name*)
removes the property with the given name.

`find`(*name*, *value*)
return the Params with the given value for name

`property nparams`
number of Params

`property nprops`
number of properties

`param`(*id*)
fetch the Param with the given id

`property params`
list of all Params in table

`propType`(*name*)
type of the property with the given name

`property props`
names of the properties

`msys.`**`ParseSDF`**(*text*)
Iterate over blocks in sdf format text. Accepts normal and gzipped text.

`msys.`**`ReadCrdCoordinates`**(*mol*, *path*)
Read coordinates from the given Amber crd file into the given System.

`msys.`**`ReadPDBCoordinates`**(*mol*, *path*)
Read coordinates and box from the given pdb file into the given System.

`class` `msys.`**`Residue`**(*_ptr*, *_id*)
Represents a residue (group of Atoms) in a System

`__repr__`()
Return repr(self).

`addAtom`()
append a new Atom to this Residue and return it

`property atoms`
list of Atoms in this Residue

`property center`
return geometric center of positions of atoms in residue

`property chain`
parent chain

`property insertion`
insertion code

`property name`
residue name

`property natoms`
number of atoms in this residue

remove()
remove this Residue from the System

property resid
the PDB residue identifier

selectAtom(*name=None*)
Returns a single Atom from this residue with the given name, or None if no such atom is present. If multiple atoms in the residue have that name, raise an exception.

msys.**Save**(*mol*, *path*, *append=False*, *structure_only=False*)
Save the given system to path, using a file format guessed from the path name. Not all formats support both append and structure_only options; see the corresponding SaveXXX functions.

msys.**SaveDMS**(*system*, *path*, *structure_only=False*, *unbuffered=False*)
Export the System to a DMS file at the given path.

msys.**SaveMAE**(*system*, *path*, *with_forcefield=True*, *append=False*)
Export the System (or list of systems) to an MAE file at the given path.

msys.**SaveMol2**(*system*, *path*, *selection='none'*, *append=False*, *moe=True*)
Export the System to a mol2 file at the given path. You should probably call AssignBondOrderAndFormalCharge() before exporting the system.

Parameters

- **system** (`System`) – msys system
- **path** (`str`) – file name to save to
- **selection** (`str`) – msys selection string to restrict to
- **append** (`bool`) – if True, don't clobber path, just append to it
- **moe** (`bool`) – output guadinium groups with aromatic bonds so that MOE will read correctly

msys.**SavePDB**(*system*, *path*, *append=False*, *reorder=False*)
Export the System to a PDB file at the given path.

msys.**SerializeMAE**(*system*, *with_forcefield=True*)
Return the MAE form of the System as bytes.

class msys.**SmartsPattern**(*pattern*)
A class representing a compiled SMARTS pattern

The Msys smarts implementation is similar to that of *Daylight smarts <http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html>*, with support for arbitrarily nested recursive smarts. A few features are not currently supported; warnings will be generated when these constructs are used in a smarts pattern.

- Directional bonds; e.g. \ and /; these are treated as single bonds (i.e. as a - character).
- Chiral specification (`@`, `@@`, etc); ignored.
- Implicit hydrogen (`h`): treated as explicit `H`.
- Explicit degree (`D`): treated as bond count `X`.
- Isotopes: (`[12C]`): ignored.
- Atom class (`[C:6]`): ignored.

On the other hand, Msys does support hybridization using the ^ token, as in OpenBabel:

```
[c^2]       select sp2 aromatic carbon
```

`__init__`(*pattern*)
Initialize with SMARTS pattern

`__repr__`()
Return repr(self).

`__weakref__`
list of weak references to the object (if defined)

`findMatches`(*annotated_system*, *atoms=None*)
Return list of lists representing ids of matches of this pattern in this system, optionally requiring that the first atom match belongs to the given set of atoms. An AnnotatedSystem must be used here, which can be constructed from a System after calling AssignBondOrderAndFormalCharge.

`match`(*annotated_system*)
Return True if a match is found anywhere; False otherwise.

This is much faster than checking for an empty result from findMatches.

`property natoms`
Number of atoms in the compiled smarts pattern

`property pattern`
The pattern used to initialize the object

`property warnings`
Warnings, if any, emitted during compilation

`class` `msys.`**`SpatialHash`**(*pos*, *ids=None*, *box=None*)
SpatialHash provides an interface for efficient spatial queries on particle positions.

`__init__`(*pos*, *ids=None*, *box=None*)
Construct from particle positions. If ids are provided, they should be a numpy array of type uint32 and specify which rows of the Nx3 pos array are to be hashed. If box is provided, it must be a 3x3 array of doubles, and the search will be performed using periodic boundary conditions.

`__weakref__`
list of weak references to the object (if defined)

`findContacts`(*radius*, *pos*, *ids=None*, *reuse_voxels=False*)
Find pairs of particles within radius of each other.

Parameters

- **radius** (*`float`*) – search radius
- **pos** (*`array[float]`*) – positions
- **ids** (*`array[uint]`*) – particle indices
- **reuse_voxels** (*`bool`*) – assume voxelize(R>=radius) has already been called
- **ignore_excluded** (*`bool`*) – exclude atom pairs in the exclusion table.

Returns Mx1 arrays of ids and distances.

Return type i, j, dists (tuple)

The first array corresponds to ids in the call to findContacts; the second column to the ids passed to the SpatialHash constructor.

IMPORTANT: pairs with the same id in the constructor and the call the findContacts are excluded from the output set. Therefore, the positions passsed to findContacts should correspond to the same atom indices as the positions passed to the SpatialHash constructor.

findNearest(*k*, *pos*, *ids=None*)

Find at most k particles from pos with the smallest minimum distance to some particle in the spatial hash. If ids is not provided, it defaults to arange(len(pos)).

findWithin(*radius*, *pos*, *ids=None*, *reuse_voxels=False*)

Find particles from pos which are within the given radius of some particle in the spatial hash (i.e. provided in the SpatialHash constructor). By default, voxelization is performed at the same resolution as the query radius, but this can be overridden by calling voxelize() manually, then calling findWithin() with reuse_voxels=True. pos is expected to be an Nx3 array of floats. The ids parameter defaults to arange(len(pos)); supply an array of ids to limit the search to a subset of rows in pos.

voxelize(*radius*)

Perform voxelization such that findWithin queries with reuse_voxels=True at a radius equal to or less than the given radius can be performed accurately. For queries at radius less than the voxelization, it may be worthwhile to revoxelize at a smaller radius. Note that, by default, findWithin calls voxelize with the query radius as argument, so it is not strictly necessary ever to use this method.

class `msys.`**System**(*_ptr*)

The *System* class holds all structure and forcefield data for a single chemical system. Create a new *System* using `msys.CreateSystem()`, or from a file using `msys.LoadDMS` or `msys.LoadMAE`.

A *System* organizes the information in a DMS file into several different groups:

- Tables - *TermTables* are grouped and accessed by name
- cell - the unit cell vectors for the *System*, in the form of a 3x3 NumPy array.
- nonbonded_info - the NonbondedInfo object describing the type of nonbonded interactions.
- provenance - a list of Provenance objects describing how the input file has been processed.
- Auxiliary tables: Everything else in the DMS file that does not fit into one of the above categories finds its way into an auxiliary table. Notable denizens of this category include:
  - cmap tables
  - forcefield (annotation for parameters in the DMS file)

__eq__(*x*)

Return self==value.

__getinitargs__()

Pickle support (requires cPickle.HIGHEST_PROTOCOL)

__hash__()

Return hash(self).

__init__(*_ptr*)

Construct from SystemPtr. Do not invoke directly; use CreateSystem() instead.

__ne__(*x*)

Return self!=value.

__repr__()

Return repr(self).

addAtom()

add and return a new Atom in its own residue

addAtomProp(*name*, *type*)
add a custom atom property with the given name and type. type should be int, float, or str.

addAuxTable(*name*, *table*)
add or replace extra table with the given name.

addBondProp(*name*, *type*)
add a custom bond property with the given name and type. type should be int, float, or str.

addChain(*ct=4294967295*)
add and return a new Chain. If no ct is given, the chain will be added to the first ct, creating one if necessary.

addCt()
add and return a new Ct

addNonbondedFromSchema(*funct*, *rule=''*)
Add a nonbonded table to the system, and configure the nonbonded info according to funct and rule. funct must be the name of recognized nonbonded type. rule is not checked; at some point in the future we might start requiring that it be one of the valid combining rules for the specified funct. If nonbonded_info's vdw_funct and vdw_rule are empty, they are overridden by the provided values; otherwise, the corresponding values must agree if funct and rule are not empty. A nonbonded table is returned.

addResidue()
add and return a new Residue in its own chain

addTable(*name*, *natoms*, *params=None*)
add a table with the given name and number of atoms. If a table with the same name already exists, it is returned, otherwise the newly created table is returned. If no ParamTable params is supplied, a new one is created.

addTableFromSchema(*type*, *name=None*)
Add a table to the system if it not already present, returning it. If optional name field is provided, the table will be added with the given name; otherwise the name is taken from the table schema.

analyze()
Assign atom and residue types. This needs to be called manually only if you create a system from scratch, using msys.CreateSystem(); in that case, analyze() should be called before performing any atom selections.

append(*system*)
Appends atoms and forcefield from system to self. Returns a list of of the new created atoms in self. Systems must have identical nonbonded_info.vdw_funct. Overwrites self.global_cell with system.global_cell only when self.global_cell is all zeros.

asCapsule()
Return a capsule wrapper of the internal SystemPtr.

The capsule holds a bare pointer and therefore must not outlive self.

atom(*id*)
return the atom with the specified id

atomPropType(*name*)
type of the given atom property

property atom_props
return the list of custom atom properties.

property atoms
return list of all atoms in the system

atomsGroupedBy(*prop*)
Return dictionary mapping representative values of the given atom property to lists of atoms having that property. If the property does not exist in this system, returns an empty dictionary.

atomsel(*sel*)
Create and return an atom selection object (Atomsel). :param sel: str atom selection, or list of GIDs (possibly empty). :type sel: object

Note: Even if ids are provided, the ids of the selection are in sorted order.

auxtable(*name*)
auxiliary table with the given name

property auxtable_names
names of the auxiliary tables

property auxtables
all the auxiliary tables

bond(*id*)
return the bond with the specified id

bondPropType(*name*)
type of the given bond property

property bond_props
return the list of custom bond properties.

property bonds
return list of all bonds in the system

property cell
The GlobalCell for this System

property center
return geometric center of positions of all atoms

chain(*id*)
return the chain with the specified id

property chains
return list of all chains in the system

clone(*sel=None*, *share_params=False*, *use_index=False*, *forbid_broken_bonds=False*)
Clone the System, returning a new System. If selection is provided, it should be an atom selection string, a list of ids, or a list of Atoms.

If share_params is True, then ParamTables will be shared between the old and new systems. By default, copies of the ParamTables are made, but ParamTables shared _within_ the old system will also be shared in the new system.

If forbid_broken_bonds is True, an exception will be thrown if the selected atoms do not include all atoms connected by bonds.

coalesceTables()
Invoke TermTable.coalesce on each table

ct(*id*)
return the Ct with the specified id

property cts
return list of all cts in the system

delAtomProp(*name*)
remove the given custom atom property

delAtoms(*atoms*)
remove the given Atoms from the System

delAuxTable(*name*)
remove auxiliary table with the given name.

delBondProp(*name*)
remove the given custom bond property

delBonds(*bonds*)
remove the given Bonds from the System

delChains(*chains*)
remove the given Chains from the System

delResidues(*residues*)
remove the given Residues from the System

findBond(*a1*, *a2*)
return the bond between the specified atoms, or None if not found

findContactIds(*cutoff*, *ids=None*, *other=None*, *pos=None*, *ignore_excluded=False*)
Find atoms not bonded to each other which are within cutoff of each other. If ids is not None, consider only atoms with the given ids. If other is not None, consider only atom pairs such that one is in ids and the other is in other. If pos is not None, use pos as positions, which should be natoms x 3 regardless of the size of ids or other. pos may be supplied only when there are no deleted atoms in the structure. If ignore_excluded=True, exclusions from the exclusion table are used. If ignore_excluded=False, bonds in System.bonds are still excluded.

Returns a list of (id 1, id 2, distance) tuples for each contact found.

classmethod fromCapsule(*cap*)
Construct from a capsule wrapper of a SystemPtr.

getCell()
return copy of unit cell as 3x3 numpy array

getPositions()
get copy of positions as Nx3 array

getTable(*name*)
Return the TermTable with the given name, or None if not present.

getVelocities()
get copy of velocities as N3x array

guessBonds(*replace=True*, *reanalyze=True*, *periodic=False*)
Guess bond connectivity based on an atomic-number based atom radius.

Replaces any existing bonds, unless replace=False is specified.

Reanalyzes fragids and atom types unless reanalyze=False is specified. In that case, you MUST call updateFragids() manually before making any use of the fragment assignment (fragids will be out of date).

hash(*sorted=True*)
hash of contents of this system.

The hash is insensitive to provenance and System.name.

property name
The name of the System, taken from the input file name

property natoms
number of atoms

property nbonds
number of bonds

property nchains
number of chains

property ncts
number of Cts

property nonbonded_info
NonbondedInfo for this System

property nresidues
number of residues

property positions
Nx3 list of lists of positions of all atoms

property provenance
return a list of Provenance entries for this system

residue(*id*)
return the residue with the specified id

property residues
return list of all residues in the system

save(*path*, *structure_only=False*)
Write self to path

Parameters

- **path** (`str`) – file path
- **structure_only** (`bool`) – write only atom information, not forcefield

Returns self

select(*seltext*)
return a list of Atoms satisfying the given VMD atom selection.

selectArr(*seltext*)
Return the ids of the Atoms satisfying the given VMD atom selection as a numpy array of type uint32.

selectChain(*name=None*, *segid=None*)
Returns a single Chain with the matching name and/or segid, or raises an exception if no single such chain is present.

selectCt(*name=None*)
Return a single Ct with the matching name, or raises an exception if no single such Ct is present

selectIds(*seltext*, *pos=None*, *box=None*)
Return the ids of the Atoms satisfying the given VMD atom selection. This can be considerably faster than calling select().

if pos is supplied, it should be an Nx3 numpy array of positions, where N=self.natoms.

If box is supplied, it should be a 3x3 numpy array of cell vectors, like System.cell.

setCell(*cell*)
set unit cell from from 3x3 array

setPositions(*pos*)
set positions from Nx3 array

setVelocities(*vel*)
set velocities from Nx3 array

sorted()
Return a clone of the system with atoms reordered based on their order of appearance in a depth-first traversal of the structure hierarchy.

table(*name*)
Get the TermTable with the given name, raising ValueError if not present.

property table_names
names of the tables in the System

property tables
all the tables in the System

property topology
list of bonded atoms for each atom in the System

translate(*xyz*)
shift coordinates by given amount

updateFragids()
Find connected sets of atoms, and assign each a 0-based id, stored in the fragment property of the atom. Return a list of fragments as a list of lists of atoms.

class `msys.`**SystemImporter**(*system*)
Maps atoms to residues, chains and cts

__init__(*system*)
construct from system to be constructed

Parameters **system** (`System`) – msys System

__weakref__
list of weak references to the object (if defined)

addAtom(*chain*, *segid*, *resnum*, *resname*, *aname*, *insertion=''*, *ct=0*)
Add atom to system

Returns newly added Atom

initialize(*atoms*)
Process existing atoms in the system

Parameters **atoms** (`list[msys.Atom]`) –

Note: can be called multiple times; each time clears the internal tables so that subsequent atoms do not share residues, chains, or cts with previously added atoms.

---

property system
input system

`msys.`**TableSchemas**()
available schemas for System.addTableFromSchema

class msys.**Term**(*_ptr*, *_id*)

A *Term* is a handle for an entry in a *TermTable*.

The properties of a *Term* can be read and updated using a dictionary like interface. Both "term properties" and properties from the *ParamTable* are accessed through the same interface. To add or remove properties, use the provided methods in the *TermTable* or *ParamTable* instance. If a *Term*'s `param` is shared by another *Term* in any other *TermTable*, Msys will take care of providing the *Term* with its own *Param* containing a copy of the original properties before applying the changes. However, if you a modify a *Param* through its dictionary interface, you will affect all *Terms* that happen to share that *Param*:

```
# fetch the stretch_harm table
table = mol.table('stretch_harm')
# update the properties of just the first Term
table.term(0)['fc'] = 320
# update the properties of all terms that use this param!
table.term(0).param['fc'] = 320
```

__getitem__(*prop*)

get the value of property prop

__repr__()

Return repr(self).

__setitem__(*prop*, *val*)

set the value of property prop

property atoms

list of Atoms for this Term

property id

id of this term in its TermTable

keys()

union of table.params.props and table.term_props

property param

The Param corresponding to this Term's parameters

remove()

remove the given Term from its TermTable

property system

parent System of parent TermTable

property table

parent TermTable

class msys.**TermTable**(*_ptr*)

Each TermTable is intended to describe a specific type of interaction, e.g. stretch, angle, Lennard-Jones, constraint_hoh, etc. A TermTable has an arity (given by the natoms property) which specifies how many atoms are involved in each interaction: one for nonbonded terms, two for stretch terms, etc. Each interaction instance is described by a *Term*. Each Term references the appropriate number of atoms, and exactly one Param, which lives in a ParamTable owned (or possible shared) by the TermTable.

The functional form described by a TermTable is not part of msys; all msys does is represent the forcefield parameters in a generic way.

__eq__(*x*)

Return self==value.

__hash__()

Return hash(self).

`__init__`(*_ptr*)
Construct from TermTablePtr. Do not invoke directly; use System.addTable or System.table instead

`__ne__`(*x*)
Return self!=value.

`__repr__`()
Return repr(self).

`addTerm`(*atoms*, *param=None*)
Add a Term to the table, with given initial param. The atoms list must have natoms elements, and each Atom must belong to the same System as the TermTable. If param is not None, it must belong to the ParamTable held by the TermTable.

`addTermProp`(*name*, *type*)
add a custom Term property of the given type

`property category`
A string describing what kind of TermTable this is. Possibilities are: *bond*, *constraint*, *virtual*, *polar*, *nonbonded*, and *exclusion*.

`coalesce`()
Reassign param for each Term in this Table to a member of the distinct set of Params used by those Terms.

`count_overrides`()
return the number of pairwise nonbonded interactions which are affected by the overrides in the given term table.

`delTermProp`(*name*)
remove the custom Term property

`delTermsWithAtom`(*atom*)
remove all terms whose atoms list contains the given Atom

`findExact`(*atoms*)
return the terms that contain precisely the given atoms in the given order.

`findWithAll`(*atoms*)
return the terms that contain all the given atoms in any order

`findWithAny`(*atoms*)
return the terms that contain at least one of the given atoms

`findWithOnly`(*atoms*)
return the terms that contain only the given atoms

`getOverride`(*pi*, *pj*)
get override for given pair of params, or None if not present.

`hasTerm`(*id*)
Does a Term with the given id exist in the table?

`property name`
name of this table

`property natoms`
number of atoms in each term

`property noverrides`
number of parameter overrides

`property nterms`
number of terms

property override_params
parameter table containing override values

overrides()
return a mapping from pairs of Params in self.params to a Param in self.override_params.

property params
The ParamTable for terms in this table.

property props
Table properties

remove()
Remove this table from its parent system

replaceWithSortedTerms()
Replace table in self.system with a version of self that has terms sorted by atom ids.

setOverride(*pi*, *pj*, *op*)
override the interaction between params pi and pj with the interaction given by op. pi and pj must be Params from self.params; op must be a param from self.override_params, or None to remove the override.

property system
The System whose atoms are referenced by this table.

term(*id*)
returns the Term in the table with the given id

termPropType(*name*)
type of the given Term property

property term_props
names of the custom properties

property terms
returns a list of all the Terms in the table

3.2.1 The Atomsel class

class msys.atomsel.**Atomsel**(*ptr*, *seltext*)
Supports alignment of molecular structures

__init__(*ptr*, *seltext*)
don't use directly - use System.atomsel()

__len__()
number of selected atoms

__repr__()
Return repr(self).

__str__()
Return str(self).

alignCoordinates(*other*)
If other is an Atomsel instance, align the coordinates of other's System with self. If other is a numpy array, align the array with self, using corresponding indices.

In either case, return the aligned RMSD.

alignedRMSD(*other*)
Return the aligned rmsd to other.

currentRMSD(*other*)
compute RMS distance to other object, which may be Atomsel or an array of positions. In either it must be the case that len(other) equals len(self) or len(self.system)

property ids
ids of selected atoms in the parent system

raw_alignment(*other*)
Compute alignment to other object. Compute and return aligned rmsd, and rotational and translational transformations.

property system
parent system

3.2.2 Dealing with duplicate parameters

After performing various modifications to a *TermTable*, you may find that the associated *ParamTable* contains many entries whose values are all identical. The redundant parameters can be removed by first "coalescing" the parameter assignments of each *Term* to a set of distinct *Params*, then cloning the *System*. When a *System* is cloned, only the *Params* which are referenced by at least one *Term* in the *TermTable* are copied to the new *System*:

```
import msys
mol=msys.CreateSystem()
a1=mol.addAtom()
a2=mol.addAtom()
a3=mol.addAtom()
table = mol.addTableFromSchema('stretch_harm')
p1=table.params.addParam()
p1['fc']=320
p1['r0']=1.0
t1=table.addTerm([a1,a2], p1)
t2=table.addTerm([a1,a3], p1)

# At this point we have two terms and one param.  Suppose we ignore the
# fact that t1 and t2 share a Param, and we just update their properties
# to the same value:

t1['r0']=1.2
t2['r0']=1.2

# Now we have two Params, because when we updated t1, we created a second
# Param that was unshared by t2.  When we updated t2, p1 was unshared, so
# no duplicate was made.
assert table.params.nparams==2

# But we could get by with only a single Param.  Let's do that:
mol.coalesceTables()

# At this point t1 and t2 are sharing a Param, and the other one is unused:
assert t1.param==t2.param
assert table.params.nparams==2
assert table.nterms==2

# When we clone, the unused params are not copied to the new system.
mol2=mol.clone()
assert mol2.table('stretch_harm').params.nparams==1
```

class `msys.`**`TermTable`**(*_ptr*)

Each TermTable is intended to describe a specific type of interaction, e.g. stretch, angle, Lennard-Jones, constraint_hoh, etc. A TermTable has an arity (given by the natoms property) which specifies how many atoms are involved in each interaction: one for nonbonded terms, two for stretch terms, etc. Each interaction instance is described by a *Term*. Each Term references the appropriate number of atoms, and exactly one Param, which lives in a ParamTable owned (or possible shared) by the TermTable.

The functional form described by a TermTable is not part of msys; all msys does is represent the forcefield parameters in a generic way.

`addTerm`(*atoms*, *param=None*)

Add a Term to the table, with given initial param. The atoms list must have natoms elements, and each Atom must belong to the same System as the TermTable. If param is not None, it must belong to the ParamTable held by the TermTable.

`addTermProp`(*name*, *type*)

add a custom Term property of the given type

`property category`

A string describing what kind of TermTable this is. Possibilities are: *bond*, *constraint*, *virtual*, *polar*, *nonbonded*, and *exclusion*.

`coalesce`()

Reassign param for each Term in this Table to a member of the distinct set of Params used by those Terms.

`count_overrides`()

return the number of pairwise nonbonded interactions which are affected by the overrides in the given term table.

`delTermProp`(*name*)

remove the custom Term property

`delTermsWithAtom`(*atom*)

remove all terms whose atoms list contains the given Atom

`findExact`(*atoms*)

return the terms that contain precisely the given atoms in the given order.

`findWithAll`(*atoms*)

return the terms that contain all the given atoms in any order

`findWithAny`(*atoms*)

return the terms that contain at least one of the given atoms

`findWithOnly`(*atoms*)

return the terms that contain only the given atoms

`getOverride`(*pi*, *pj*)

get override for given pair of params, or None if not present.

`hasTerm`(*id*)

Does a Term with the given id exist in the table?

`property name`

name of this table

`property natoms`

number of atoms in each term

`property noverrides`

number of parameter overrides

property nterms
number of terms

property override_params
parameter table containing override values

overrides()
return a mapping from pairs of Params in self.params to a Param in self.override_params.

property params
The ParamTable for terms in this table.

property props
Table properties

remove()
Remove this table from its parent system

replaceWithSortedTerms()
Replace table in self.system with a version of self that has terms sorted by atom ids.

setOverride(*pi*, *pj*, *op*)
override the interaction between params pi and pj with the interaction given by op. pi and pj must be Params from self.params; op must be a param from self.override_params, or None to remove the override.

property system
The System whose atoms are referenced by this table.

term(*id*)
returns the Term in the table with the given id

termPropType(*name*)
type of the given Term property

property term_props
names of the custom properties

property terms
returns a list of all the Terms in the table

3.2.3 Sharing ParamTables

Forcefield developers will (we hope!) appreciate the ability for Msys to parameterize multiple *TermTables* from potentially different *Systems* using a single *ParamTable* instance. Normally, when a *System* is loaded from an input file, or a *TermTable* is created using the scripting interface, each *TermTable* refer to a *ParamTable* of its very own, and no other *TermTable* can or will reference it. However, at the time that a *TermTable* is created, a *ParamTable* can be provided which will be used to hold the *Param* entries for the *Terms* in the *TermTable*:

```
# create two independent systems
m1=msys.CreateSystem()
m2=msys.CreateSystem()

# add some atoms
m1.addAtom()
m2.addAtom()
m2.addAtom()

# create a free-standing ParamTable and add some Params
params=msys.CreateParamTable()
```

(continues on next page)

(continued from previous page)

```
p1=params.addParam()
p2=params.addParam()

# create a table in system 1 which uses the free ParamTable
table1=m1.addTable("table", 1, params)

# no other TermTable is using the ParamTable
assert not params.shared

# create a table in system 2 which also uses the free ParamTable
table2=m2.addTable("table", 1, params)

# now the ParamTable is shared
assert params.shared
assert table1.params == table2.params

# Add some terms to each table
t1=table1.addTerm(m1.atoms, p2)
t2=table2.addTerm(m2.atoms[1:], p2)

assert t1.param == t2.param
assert t2.param == p2

# modifications to the the original table and its params are propagated
# to each table
params.addProp("fc", float)
p1['fc']=32
p2['fc']=42
assert t1['fc']==42
assert t2['fc']==42

# p1 is shared by multiple TermTables, but within a TermTable, p1 is not
# shared.  Modifications to t1['fc'] will affect t2!
t1['fc'] = 52
assert t2['fc'] == 52
```

3.3 Pfx

A high level interface for wrapping, centering, and alignment.

This module provides simple, yet performant methods for manipulating trajectories of systems with connected atoms and periodic boundary conditions.

`msys.pfx.`**`aligned_rmsd`**(*X*, *Y*, *weight=None*) → mat, rmsd
Compute the matrix aligning Y onto X, optionally with weights. Return the matrix and the rmsd.

`msys.pfx.`**`inverse_3x3`**(*A*) → Ainv

`msys.pfx.`**`svd_3x3`**(*A*) → U, w, V
svd_3x3 computes the singular value decomposition of the 3x3 matrix A. The result is always calculated and returned in double precision.

3.3.1 What pfx does

Pfx can be configured to perform a number of tasks related to postprocessing of trajectories of molecular systems. There are four main issues which **Pfx** is designed to deal with:

1. *Fixing bonds*. If two atoms with a bond between them are found in different periodic images, one of them must be shifted by some integer linear combination of the unit cell vectors so that the distance between them is no greater than half a unit cell vector along each cell vector direction. When there are multiple atoms bonded together, the bond fixing operation must be applied to the bonds composing this connected component in topologically sorted order.
2. *Gluing components*. Some molecular systems, such as multimeric ion channels, contain components which are not explicitly bonded to each other, but which do stay together during the simulation and should therefore be kept together during postprocessing. For each set of glued components, **Pfx** finds the transformations which minimize the square distance between the centers of each component.
3. *Centering and alignment*. **Pfx** can either center a selected set of atoms on the origin, or align a selected set to a reference structure.
4. *Wrapping components*. Any of the preceeding operations could place the center of a connected set of atoms outside the unit cell centered at the origin. **Pfx** shifts each connected component to bring it as close to the origin as possible, which maintaining any glued components. Importantly, if an alignment has been performed in the previous step, then the rotational part of the alignment transformation must be applied to the unit cell before performing the wrapping. Another subtlety is that when alignemnt has been performed, the wrapping should be performed about the center of the reference selection, not necessarily the origin. Otherwise, if the reference structure is far from the origin, wrapping could undo the alignment.

The main work of **pfx** is does in the *apply* method. The arguments to *apply* are a coordinate set and, optionally, a periodic cell and/or a set of velocities. Here's what happens when you call *apply*, assuming that both the periodic cell and the velocities have been provided:

1. Fix bonds.
2. Glue components.
3. Translate the entire system to bring the centered or aligned atoms to the origin.
4. Compute a rotational transformation which aligns the system to the centered reference structure.
5. Apply the rotation to the input positions, unit cel, and velocities.
6. Wrap connected and glued components.
7. Shift the entire system to the center of the reference structure.

3.3.2 Specifying topology

A **Pfx** instance is constructed from a bond topology. The bond topology indicates both how many atoms are in the molecular system as well as which atoms are bonded together. **Pfx** analyzes this topology to find the connected components comprising the system. If the *fixbonds* argument is True, then **Pfx** also computes and caches a topologically sorted list of bonds from the topology, so that the bond fixing step can be performed efficiently.

3.3.3 Specifying glue

The *glue* method of **Pfx** lets you specify atoms which should be kept together even if there is no explicit bond between them. Suppose the ids of all the protein atoms in a multimeric protein are passed as the argument to *glue*. **Pfx** first finds the set of connected components which overlap with the selection. In the *glue components* step, the centers of these components will be brought together. Moreover, in the *wrap components* step, all the protein atoms will be treated as a single component for the purpose of wrapping.

Suppose now that only one residue from each monomer of a multicomponent protein is included in a *glue* selection. The same set of connected components will be kept together as before, when the entire protein was glued; however, the centers of the connected components will be computed from just the glued residue in each monomer, rather than from all atoms of each monomer. The *wrap components* step will be unchanged.

The *glue* method can be called multiple times on the same **Pfx** instance. It is perfectly valid for glue selections in different invocations to overlap.

3.3.4 Performing both centering and alignment

When viewing trajectories, chemists often want to specify both "center" and "fit" selections. But what does this mean? If you center on, say, atoms 1-10, and align atoms 10-20, one operation will undo the other. The only sensible approach seems to be to apply the "center" specification to whatever is being used for the reference structure, and then use the "fit" selection to align non-reference frames to that selection.

3.3.5 What about periodicfix?

The algorithms in this module are essentially the same as those in periodicfix. So why a new module? Here are some reasons:

- Periodicfix isn't consistent about its use of float and double, and does a lot of interconversion. This makes it slow.
- Periodicfix doesn't have both single and double precision versions available from Python. This one does.
- Periodicfix makes you specify weights to align a subset of atoms. Pfx doesn't use weights; or, if you like, the weights must be zero or one. In practice it's been found that that's all anyone needs. Having to specify weights is cumbersome. If someone really wants to have weight support in pfx we can add it some day.
- Periodicfix has separate topology and fragment wrapper types, which make the Python interface more cumbersome. Pfx has just one type.
- Periodicfix has accreted additional functionality which has nothing to do with periodic images or alignment, including contact finding and a hydrogen bond energy function.
- The svd in periodicfix is greatly inferior to the one here. Yes, it would be easy replace the one in periodicfix with this one.

3.4 Molfile

Structure and coordinate file manipulation library.

Reading a structure file:

```
reader = molfile.mae.read('/path/to/foo.mae')
```

Iterating through the frames in a file:

```
for frame in molfile.dtr.read('/path/to/foo.dtr').frames():
    function( frame.pos, frame.vel, frame.time, frame.box )
```

Random access to frames (only dtr files support this currently):

```
f27 = molfile.dtr.read('/path/to/foo.dtr').frame(27) # 0-based index
```

Write a trajectory to a frameset (dtr):

```
f = msys.molfile.Frame(natoms)
w = msys.molfile.dtr.write('output.dtr', natoms=natoms)
for i, xyz in enumerate(xyzs):
    f.pos[:] = xyz
    f.time = i
    w.frame(f)
w.close()
```

Convert an mae file to a pdb file:

```
input=molfile.mae.read('foo.mae')
output=molfile.pdb.write('foo.pdb', atoms=input.atoms)
output.frame(input.frames().next())
output.close()
```

Write every 10th frame in a dtr to a trr:

```
input=molfile.dtr.read('big.dtr')
output=molfile.trr.write('out.trr, natoms=input.natoms)
for i in range(0,input.nframes, 10):
    output.frame( input.frame(i) )
output.close()
```

Write a frame with a specified set of gids:

```
f = molfile.Frame(natoms, with_gids=True
f.gid[:] = my_gids
f.pos[:] = my_positions
w.frame(f)
```

Read the raw fields from a frameset (dtr):

```
dtr = molfile.DtrReader('input.dtr')     # also works for stk
for i in range(dtr.nframes):
    f = dtr.frame(i)
    keyvals = dict()
    frame = dtr.frame(i, keyvals=keyvals)
    ## use data in keyvals
```

Write raw fields to a frameset (dtr):

```
dtr = molfile.DtrWriter('output.dtr', natoms=natoms)
keyvals = dict( s = "a string",
                f = positions.flatten(),     # must be 1d arrays
                i = numpy.array([1,2,3]),
                )
dtr.append( time = my_time, keyvals = keyvals )
```

class msys.molfile.**Plugin**

Interface to readers and writers

property can_read

property can_write

property filename_extensions

property name

property prettyname

read (*(Plugin)arg1*, *(str)path*[, *(bool)double_precision=False*]) → Reader :

Open a file for reading

property version

write (*(Plugin)arg1*, *(str)path*[, *(object)atoms=None*[, *(object)natoms=None*]]) → Writer :

write(path,atoms=None,natoms=None)

class msys.molfile.**DtrReader**

fileinfo (*(DtrReader)arg1*, *(int)arg2*) → tuple :

fileinfo(index) -> path, time, offset, framesize, first, last, filesize, dtrpath, dtrsize file contains frames [first, last)

frame (*(DtrReader)arg1*, *(int)index*[, *(object)bytes=None*[, *(object)keyvals=None*]]) → object :

frame(index, bytes=None, keyvals=None) -> Frame Read bytes from disk if bytes are not provided If keyvals is not None, it should be a dict, and raw data from the frame will be provided.

frameset_is_compact (*(DtrReader)arg1*, *(int)arg2*) → bool

frameset_path (*(DtrReader)arg1*, *(int)arg2*) → str

frameset_size (*(DtrReader)arg1*, *(int)arg2*) → int

static fromTimekeys (*(str)arg1*, *(list)arg2*, *(list)arg3*) → DtrReader

index_ge (*(DtrReader)arg1*, *(float)arg2*) → int

index_gt (*(DtrReader)arg1*, *(float)arg2*) → int

index_le (*(DtrReader)arg1*, *(float)arg2*) → int

index_lt (*(DtrReader)arg1*, *(float)arg2*) → int

index_near (*(DtrReader)arg1*, *(float)arg2*) → int

keyvals (*(DtrReader)arg1*, *(int)arg2*) → dict :

keyvals(index) -> dict() Read raw fields from frame.

property metadata

property natoms

property nframes

property nframesets

property path

reload(*(DtrReader)arg1*) → int :
reload() -> number of timekeys reloaded – reload frames in the dtr/stk

times(*(DtrReader)arg1*) → object

total_bytes(*(DtrReader)arg1*) → int

class `msys.molfile.`**Frame**

property box

property dpos

property dvel

property extended_energy

property fpos

property fvel

property kinetic_energy

moveby(*(Frame)arg1*, *(float)x*, *(float)y*, *(float)z*) → None

property pos

property position

property potential_energy

property pressure

property pressure_tensor

select(*(Frame)arg1*, *(object)indices*) → Frame

property temperature

property time

property total_energy

property vel

property velocity

property virial_tensor

class `msys.molfile.`**Atom**

addbond()
addbond(atom) – add atom to self.bonds and self to atom.bonds

altloc
segment name

anum
atomic number

bfactor
temperature factor

bonds
bonded atoms

chain
chain name

charge

delbond()
delbond(atom) – remove atom from self.bonds and self from atom.bonds

getbondorder()
getbondorder(atom) – bond order for bond with given atom.

insertion
segment name

mass
mass in amu

name
atom name

occupancy

radius
vdw radius

resid
residue id

resname
residue name

segid
segment name

setbondorder()
setbondorder(atom,val) – set bond order for bond with given atom.

type
atom type

class msys.molfile.**SeqFile**
Read csv-like files with column names in the first row

class Reader(*path*)

at_time_near(*time*)

frame(*n*)

frames()

get_prop(*prop*)

property natoms

property nframes

filename_extensions = 'seq'

name = 'seq'

classmethod read(*path*)
Open an eneseq file for reading

class msys.molfile.**Grid**(*data*, *name=''*, *axis=None*, *origin=None*)

property axis

property data

property name

property origin

3.4.1 Reader

class msys.molfile.**Reader**
Structure or trajectory open for reading

A Reader is a handle to an open file. Use the atoms member to fetch the atomic structure from the file, assuming it exists. To access frames, there are two methods.

frames()
returns a FrameIter object for iteration over frames. FrameIter has two methods: the usual next() method which returns a Frame, and skip(n=1), which advances the iterator by n frames without (necessarily) reading anything. FrameIter is a very poor iterator: once a frame has been read or skipped, it can't be loaded again; you have use a brand new Reader.

frame(*n*)
returns the nth frame (0-based index). Currently only the dtr plugin supports this method.

grid(*n*)
return the nth grid. For dx and ccp4 files.

at_time_ge(*(Reader)arg1*, *(float)time*) → Frame

at_time_gt(*(Reader)arg1*, *(float)time*) → Frame

at_time_le(*(Reader)arg1*, *(float)time*) → Frame

at_time_lt(*(Reader)arg1*, *(float)time*) → Frame

at_time_near(*(Reader)arg1*, *(float)time*) → Frame

property atoms
list of Atoms

property bondorders
list of bond orders

frame(*(Reader)arg1*, *(int)arg2*) → Frame

grid_data(*(Reader)arg1*, *(int)arg2*, *(object)arg3*) → None

grid_meta(*(Reader)arg1*, *(int)arg2*) → object

property has_velocities
reads velocities

property natoms
number of atoms

next(*(Reader)arg1*) → Frame :
Return the next frame

property nframes
number of frames

property ngrids
number of grids

reopen(*(Reader)arg1*) → Reader :
reopen file for reading

skip(*(Reader)arg1*) → None :
Skip the next frame

property times
all times for frames in trajectory

property topology
bond adjacency graph

3.4.2 Writer

class `msys.molfile.`**Writer**
Structure or trajectory open for writing

Writers are initialized with a path and either an array of Atoms or an atom count. If the Writer supports structure writing, Atoms must be provided; if the Writer only writes frames, either one will do.

frame(*f*)
If the writer supports frame writing, appends frame f to the end of the file.

grid(*g*)
If the writer supports grid writing, writes Grid g to the file, where g is an instance of molfile.Grid, either returned from reader.grid(n) or created from scratch.

close()
Invoked when the Writer goes out of scope, but it's not a bad idea to invoke it explicitly.

close(*(Writer)arg1*) → None

frame(*(Writer)arg1*, *(Frame)arg2*) → Writer

sync(*(Writer)arg1*) → None

truncate(*(Writer)arg1*, *(float)arg2*) → bool

3.5 AnnotatedSystem

class `msys.`**AnnotatedSystem**(*sys*, *allow_bad_charges=False*)
System that has been annotated with additional chemical information

The AnnotatedSystem class provides chemical annotation useful primarily for evaluating smarts patterns. The system is expected to already have have chemical reasonable bond orders and formal charges, and to have no missing atoms (e.g. hydrogens). If these criteria cannot be met, set allow_bad_charges=True in the constructor to bypass these checks; in that case the AnnotatedSystem can still be used to evaluate smarts patterns, but patterns making use of the electronic state of the system (e.g. aromaticity, hybridization, etc.) will not be correct (the system will appear to be entirely aliphatic). You may also use the AssignBondOrderAndFormalCharge function to assign reasonable bond orders and formal charges, assuming there are no missing atoms.

The AnnotatedSystem defines a model for aromaticity. First, the SSSR (smallest set of smallest rings) is determined. Next, rings which share bonds are detected and grouped into ring systems. Rings are initially marked as nonaromatic. If the ring system taken as a whole is deemed to be aromatic, then all rings within it are aromatic as well; otherwise, individual rings are checked for aromaticity. Rings are checked in this fashion until no new rings are found to be aromatic.

A ring system is deemed to be aromatic if it satisfies Huckel's 4N+2 rule for the number of electrons in the ring(s). An internal double bond (i.e. a bond between two atoms in the ring) adds 2 to the electron count. An external double bond (a bond between a ring atom and an atom not in that ring) adds 1 to the electron count. An external double bond between a carbon and a nonaromatic carbon makes the ring unconditionally nonaromtic. An atom with a lone pair and no double bonds adds 2 to the electron count.

`aromatic`(*atom_or_bond*)
Is atom or bond aromatic

`degree`(*atom*)
Number of (non-pseudo) bonds

`property errors`
List of errors found during system analysis if allow_bad_charges=True

`hcount`(*atom*)
Number of bonded hydrogens

`hybridization`(*atom*)
Atom hybridization – 1=sp, 2=sp2, 3=sp3, 4=sp3d, etc.

Equal to 0 for hydrogen and atoms with no bonds, otherwise max(1, a.degree() + (a.lone_electrons+1)/2 - 1).

`loneelectrons`(*atom*)
Number of lone electrons

`ringbondcount`(*atom*)
Number of ring bonds

`valence`(*atom*)
Sum of bond orders of all (non-pseudo) bonds

3.6 SmartsPattern

class `msys.`**`SmartsPattern`**(*pattern*)
A class representing a compiled SMARTS pattern

The Msys smarts implementation is similar to that of *Daylight smarts <http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html>*, with support for arbitrarily nested recursive smarts. A few features are not currently supported; warnings will be generated when these constructs are used in a smarts pattern.

- Directional bonds; e.g. `\` and `/`; these are treated as single bonds (i.e. as a - character).
- Chiral specification (`@`, `@@`, etc); ignored.
- Implicit hydrogen (`h`): treated as explicit `H`.
- Explicit degree (`D`): treated as bond count `X`.
- Isotopes: (`[12C]`): ignored.
- Atom class (`[C:6]`): ignored.

On the other hand, Msys does support hybridization using the ^ token, as in OpenBabel:

```
[c^2]       select sp2 aromatic carbon
```

findMatches(*annotated_system*, *atoms=None*)

Return list of lists representing ids of matches of this pattern in this system, optionally requiring that the first atom match belongs to the given set of atoms. An AnnotatedSystem must be used here, which can be constructed from a System after calling AssignBondOrderAndFormalCharge.

match(*annotated_system*)

Return True if a match is found anywhere; False otherwise.

This is much faster than checking for an empty result from findMatches.

property natoms

Number of atoms in the compiled smarts pattern

property pattern

The pattern used to initialize the object

property warnings

Warnings, if any, emitted during compilation

CHAPTER

FOUR

NONBONDED PARAMETERS

Nonbonded parameters for particles in Msys are handled as follows. A *System* may have at most one *TermTable* whose name is “nonbonded” and whose category is also “nonbonded”. The *ParamTable* for the nonbonded table, not surprisingly, holds the nonbonded parameters for all the atoms. Atoms are assigned a nonbonded type by creating *Terms* in the nonbonded *TermTable*. There should be exactly one *Term* for each *Atom*, and each *Atom* should be represented by exactly one *Term*.

The *System* class has a method called `addNonbondedFromSchema` which is a shortcut for creating a nonbonded table of a particular type. The argument to `addNonbondedFromSchema` will be the `vdw_funct` that appears in the `nonbonded_info` field of the *System*. The following `vdw_funct` values are currently supported:

- `vdw_12_6` : parameters `sigma`, `epsilon`
- `vdw_exp_6` : parameters `alpha`, `epsilon`, `rmin`
- `vdw_exp_6s` : parameters `sigma`, `epsilon`, `lne`

When a *System* is created by loading a DMS or MAE file, a nonbonded table will be created if and only if the input file contains nonbonded information. When saving a *System* to a DMS file, Msys checks that there is at most one nonbonded table, and if one exists, ensures that every *Atom* is found in exactly one *Term*.

4.1 Alternative nonbonded tables

Starting with version 1.6.0, Msys supports export to DMS files of systems containing nonbonded tables not named “nonbonded”. Any number of such tables may be created, either with or without the traditional “nonbonded” table”:

```
m=msys.CreateSystem()
a=m.addAtom()
disp = m.addTable('nonbonded_dispersion', 1)
repl = m.addTable('nonbonded_repulsion', 1)
elec = m.addTable('nonbonded_charge', 1)
for t in disp, repl, elec:
    t.category='nonbonded'
    p=t.params.addParam()
    t.addTerm([a],p)

disp.params.addProp('foo', float)
repl.params.addProp('bar', float)
elec.params.addProp('charge', float)
m.nonbonded_info.vdw_funct = "disp_repl_charge"
m.nonbonded_info.vdw_rule = "geom/geom/geom"
```

The `vdw_funct` attribute should reflect the nature of the nonbonded schemas that are present in the system.

Note that if there is no table named “nonbonded”, then the particle table in the DMS file will not contain an `nbtype` column.

4.2 Overriding nonbonded interactions

Nonbonded interactions between particles are usually calculated by looking up the nonbonded parameters (e.g., charge, sigma, epsilon) of the two interacting particles, performing some sort of combining operation on those parameters (e.g., geometric mean of the charge, arithmetic mean of the sigma), then using those values in the functional form of the interaction.

The DMS and MAE file formats allow one to specify nonbonded types whose combined values are to be taken from a table, rather than computed according to a combining rule. In Msys, overrides to the parameters in a *TermTable* are stored as a mapping from pairs of entries in the `params` to a entry in the `override_params` *ParamTable*. Pairs of *Params* are stored such that the `id` of the first *Param* is less than or equal to the `id` of the second *Param*; hence, there are no redundant or conflicting overrides: if parameters *i* and *j* have an override, then parameters *j* and *i* must be considered to have the same override.

4.3 Alchemical nonbonded interactions

DMS files use a table called *alchemical_particle* to indicate which particles have alchemical nonbonded states, and the parameters for those states. Msys represents the information in that table with a *TermTable* called *alchemical_nonbonded*. This table will share a *ParamTable* with the regular *nonbonded* table, but will contain *Terms* only for the alchemical particles. The parameter for each *Term* in *alchemical_nonbonded* will correspond to the B state of the term’s particle. Additional per-particle information, such as *chargeB*, *chargeC*, or *moiety*, will appear as term properties for the particles.

4.4 NonbondedInfo

class msys.**NonbondedInfo**

property es_funct
Name of the electrostatic functional form

property vdw_funct
Name of the vdw functional form; e.g., ‘vdw_12_6’

property vdw_rule
Nonbonded combining rule; e.g., ‘arithmetic/geometric’

CHAPTER

FIVE

COMMAND LINE TOOLS

Msys is packaged with a set of command line tools that wrap functionality present in the Python interface.

5.1 Conversion

5.1.1 mae2dms

5.1.2 dms2mae

5.2 Information

5.2.1 dms-version

5.2.2 dms-info

5.2.3 dms-dump

5.2.4 dms-diff

5.3 Basic Manipulation

5.3.1 dms-fix-mass

5.3.2 dms-frame

5.3.3 dms-reorder-atoms

5.3.4 dms-select

5.3.5 dms-sequence

`msys.sequence.`**`Sequences`**(*system_or_chain*, *distinct=True*)

return list of sequences, one for each chain, for the given input. The sequence will be returned as a string, with characters corresponding to the 1-letter sequence codes. If a Chain is provided instead of a System, only one sequence will be returned.

If distinct is True, only distinct sequences will be returned.

5.3.6 dms-set

`msys.update.`**`Update`**(*mol*, *atoms*, *key*, *val*)

update the system by setting properties corresponding to key to the value val. key can take the following forms: 1) foo – same as atom.foo 2) atom.foo – atom property foo 3) residue.foo – residue property foo 4) chain.foo – chain property foo 5) table.foo – property foo of all terms 6) box.foo – where foo is x, y, z, or d (sets all three)

5.4 Structure building

5.4.1 dms-grease

dms-grease input.dms lipid.dms output.dms [options]

Tile a lipid bilayer around a solute.

dms-grease builds a new chemical system consisting of the input system plus a lipid bilayer constructed by tiling *lipid.dms* in the x-y plane. If the *input.dms* is given as "-", then a pure membrane will be built.

An error will be encountered if only one of *input.dms* and *lipid.dms* have forcefield information; this is because Msys refuses to write DMS files that have only partial information for the nonbonded atom types. If you don't have forcefield information for one of the input files, use the *–structure-only* option to ignore the forcefield information in the one that does.

The global cell of the new system will be orthorhombic and have x and y dimensions given by the specified size of the membrane, and z dimension given by the input structure or the lipid membrane template, whichever is greater.

`msys.grease.`**`Grease`**(*mol*, *tile*, *thickness=0.0*, *xsize=None*, *ysize=None*, *ctname='grease'*, *verbose=True*, *square=False*)

Build and return a new system consisting of mol plus lipid bilayer. Tile is the lipid bilayer system to replicate.

If no solute is provided, the solute is treated as a point at the origin. thickness specifies the amount of solute added along the x and y axis. The dimensions of the bilayer can also be given explicitly with dimensions. If square is true, the box size will be expanded to the size of the longest dimension.

Lipids will be created in a new Ct. Their chain names will be left the same as in the original tile, but the resids will be renumbered to ensure uniqueness.

Return the greased system; no modifications are made to the input system.

5.4.2 dms-thermalize

dms-thermalize input.dms output.dms [options]

Assign Boltzmann-sampled velocities to the atoms. Atoms with zero mass will get zero velocity.

`msys.thermalize.`**`apply`**(*mol*, *T*, *seed=None*)

assign random velocities sampled from a Boltzmann distribution of temperature T.

`msys.thermalize.`**`remove_drift`**(*mol*)

Remove center of mass motion. Returns the original center of mass velocity. Zero out the velocity of pseudoparticles.

5.4.3 dms-posre

Add position restraints to a dms file, using the existing atom positions for the reference positions of the restraints. If `--replace` is specified on the command line, any existing restraints will be replaced by the new set. Otherwise, atoms that are already restrained in the existing file will be restrained using the newly provided force constraints:

```
# Add position restraints to backbone atoms with a force constant of 0.2
dms-posre input.dms out1.dms -s "backbone" -f 0.2

# Restrain CA atoms with a force constant of 0.3
dms-posre out1.dms out2.dms -s "name CA" -f 0.3

# Remove all position restraints:
dms-posre input.dms output.dms --replace
## or:
dms-posre input.dms output.dms -s none --replace
```

`msys.posre.`**`apply`**(*mol*, *atoms*, *fcx*, *fcy*, *fcz*, *replace=False*)
add position restraints to atoms

5.4.4 dms-tile

5.4.5 dms-replicate

5.4.6 dms-solvate

5.4.7 dms-neutralize

5.5 Validation

5.5.1 dms-find-knot

dms-find-knot system.dms [options]

```
/-------       \       /                  \     /                    \----\  ----------
↪-----                \         /                        \-------/
```

dms-find-knot searches for bonds which pass through a ring of atoms; e.g., a lipid tail passing through an aromatic ring in a protein. Such geometries can accidentally arise during system construction and usually indicate a badly constructed system which will behave badly during simulation.

If –untie is specified, the script will attempt to remove the knots by translating the offending bonds outside of the ring (iteratively to convergence).

The algorithm works as follows:

1. Produce a list of all cycles in the bond topology (i.e. rings)
2. **For each ring:**
    a. Use boxing and distance cutoffs to reduce the number of bonds to check against
    b. Divide the ring into N triangles
    c. Check for a triangle-line intersection between the triangle and each relevant bond

5.5.2 dms-validate

5.5.3 dtr-validate

CHAPTER

SIX

RECIPES

Here are some illustrative Python scripts for situations when a command line tool isn't available.

6.1 Obtaining force-field parameters for certain atoms

Force-field parameters are saved in "terms" inside the force-field "tables"

```
table = mol.table('stretch_harm')
atoms = mol.select('index 0 1')
terms = table.findWithAll(atoms)
print(dict(terms[0]))
# {'constrained': 0, 'fc': 317.0, 'memo': 'JCC,7,(1986),230; AA', 'r0': 1.522, 'type
↪': 'C CT'}
```

6.2 Adding artificial bonds

Msys can add force terms to a system:

```
import msys, sys
ifile, ofile = sys.argv[1:]
mol=msys.Load(ifile)
T=mol.table('stretch_harm')
P=T.params
param=P.addParam()
param['fc']=32
param['r0']=1.0
T.addTerm([mol.atom(0), mol.atom(1)], param)
# ...
msys.Save(mol, ofile)
```

6.3 Adding energy groups

Desmond and Anton use the “energy_groups” atom property to assign atoms to energy groups:

```
mol = msys.Load('system.dms')

# add an integer property.  The default value is zero.  It's a no-op
# if the property already exists, and an error if it exists but has a
# different type.
mol.addAtomProp('grp_energy', int)

# assign protein to energy group 1
for a in mol.select('protein'):
  a['grp_energy'] = 1

# save the result
msys.SaveDMS(mol, 'system_engrp.dms')
```

6.4 Remove selected constraints

Remove all constraints for atoms in a particular atom selection, and also update the ‘constrained’ property of the stretch_harm terms for the stretch terms that are no longer constrained:

```
import sys, msys
ifile, ofile=sys.argv[1:]
seltext='hydrogen and withinbonds 1 of (backbone and nitrogen)'

print "reading from %s" % ifile
mol=msys.LoadDMS(ifile)
atoms=set(mol.select(seltext))
residues=set(a.residue for a in atoms)

print "found %d atoms in selection from %d residues" % (len(atoms), len(residues))

removed_constraints=list()

for table in mol.tables:
    if table.category != 'constraint': continue
    print "doing table", table.name
    for term in table.terms:
        for a in term.atoms:
            if a in atoms:
                removed_constraints.append(set(term.atoms))
                term.remove()
                break

print "removed %d constraints" % len(removed_constraints)

stretch=mol.table('stretch_harm')
if 'constrained' in stretch.term_props:
    print "finding constrained stretch_harm terms"

    uncons=0
    for term in mol.table('stretch_harm').terms:
```

(continues on next page)

(continued from previous page)

```
        if not term['constrained']: continue
        s=set(term.atoms)
        # one of the atoms in the term must be in the original selection
        if not s.intersection(atoms):
            continue

        # mark as unconstrained if all atoms in term overlap a constraint
        for cons in removed_constraints:
            if not s.difference(cons):
                term['constrained']=0
                uncons += 1
                break

print "unconstrained %d stretch_harm terms" % uncons

print "Saving to %s" % ofile
msys.SaveDMS(mol, ofile)
```

6.5 Canonicalize position restraint terms

Some posre_harm tables have x0, y0, z0 as param properties rather than term properties. This script enforces the convention that these properties ought to be term properties:

```
def canonicalize(mol):
    posre=mol.table('posre_harm')
    props=set(('x0', 'y0', 'z0'))
    if props.issubset(set(posre.term_props)):
        print "Already canonical!"
        return

    if not props.issubset(set(posre.params.props)):
        print "Missing %s from posre params!" % (props,)
        exit(1)

    print "File is not canonical!  Fixing..."
    posre.name = '__posre_harm_old__'
    newposre=mol.addTableFromSchema('posre_harm')
    for t in posre.terms:
        p = newposre.params.addParam()
        p['fcx'] = t['fcx']
        p['fcy'] = t['fcy']
        p['fcz'] = t['fcz']
        t2 = newposre.addTerm(t.atoms, p)
        t2['x0'] = t['x0']
        t2['y0'] = t['y0']
        t2['z0'] = t['z0']
    posre.remove()
    newposre.coalesce()

def main():
    import sys
    ifile, ofile = sys.argv[1:]
    mol=msys.LoadDMS(ifile)
    canonicalize(mol)
```

(continues on next page)

(continued from previous page)

```
    mol = mol.clone()
    msys.SaveDMS(mol, ofile)

if __name__=="__main__": main()
```

6.6 Processing multi-entry files (e.g. SDF files)

To iterate over each structure in an SDF file, use msys.LoadMany. The LoadMany function is a generator, so you should iterate over its results rather than simply calling it.

Each result returned by LoadMany is a System with one ct. You'll need to access the ct member of the System in order to view or modify the data values associated with each entry.

To write entries back out to a new SDF file after filtering or modifying them, use msys.SaveSDF. It's most efficient to create your own file object in Python, and write the string returned by SaveSDF to that file.

When entries in the SDF file cannot be parsed, msys skips the next entry, and msys.LoadMany returns None for the offending entry. You should check for None in the return values of msys.LoadMany and skip them if that makes sense for your script.

Here is an example snippet which reads each entry, filters by atom count, modifies a data property, removes another data property, and writes the results to another file:

```
def process_sdf(ifile, ofile):
    fp = open(ofile, 'w')
    for i, mol in enumerate(msys.LoadMany(ifile)):
        # skip entries which couldn't be parsed
        if mol is None:
            print "Warning, skipping entry %d" % (i+1)
            continue
        # filter systems with fewer than 5 atoms
        if mol.natoms < 5:
            continue
        ct = mol.ct(0)
        # update 'THE_SCORE' property.  Note that vlaues returned by
        # get may be float, int, or string.
        score = ct.get('THE_SCORE', 0.0)
        score += 5.0
        ct['THE_SCORE'] = score
        # remove 'USELESS' property
        if ct.get('USELESS') is not None:
            del ct['USELESS']
        # write the entry back out
        fp.write(msys.SaveSDF(mol, None)
```

6.7 Processing large SDF files

If you have large sdf files with many thousands of entries, you may benefit from using a set of functions specialized for SDF files. The new functions are around 10x faster at reading SDF files and 20x faster for writing. However, there is no facility for modifying the molecular structures of each entry, though you can inspect and modify the data values. Also, the data values are always returned as strings, so you must case them to appropriate types if you wish to manipulate them as integers or floats.

The new functions are named ScanSDF and FormatSDF. Here a snippet which performs the same actions as the process_sdf function in the previous example, using the new functions:

```
def process_sdf_fast(ifile, ofile):
    fp = open(ofile, 'w')
    for i, mol in enumerate(msys.ScanSDF(ifile)):
        # skip entries which couldn't be parsed
        if mol is None:
            print "Warning, skipping entry %d" % (i+1)
            continue
        # filter systems with fewer than 5 atoms
        if mol.natoms < 5:
            continue
        # update 'THE_SCORE', coverting the existing value to float
        score = mol.get('THE_SCORE', 0.0)
        ct['THE_SCORE'] = float(score) + 5.0
        # remove 'USELESS' property
        if ct.get('USELESS') is not None:
            del ct['USELESS']
        # write the entry back out
        fp.write(msys.FormatSDF(mol)
```

6.8 Change the mass of selected atoms

Change the mass of selected hydrogens to 4, and compensate by increasing the mass of the atom bonded to the hydrogen by the appropriate amount:

```
def adjust_masses(mol, sel, compensate=True):
    for a in mol.select('hydrogen and %s' % sel):
        old = a.mass
        a.mass = 4
        if compensate:
            a.bonded_atoms[0].mass -= 4-old
```

CHAPTER

SEVEN

DMS FILES

Msys defines and implements an sqlite-based schema called DMS for chemical systems. This section provides an overview of the DMS format which will be useful for users who wish to inspect their DMS files manually using native sqlite tools, or who need to understand the functional form of the forcefield tables found in DMS files in order to, say, convert between file formats or use msys systems in their own programs.

7.1 Overview

All data in a DMS file lives in a flat list of two-dimensional tables. Each table has a unique name. Columns in the tables have a name, a datatype, and several other attributes, most importantly, whether or not the column is the primary key for the table. Rows in the tables hold a value for each of the columns. Table names, column names, and datatypes are case-preserving, but case-insensitive: thus ‘pArTiCLE’ is the same table as ‘particle’, and ‘NAME’ is the same column as ‘name’.

In addition to tables, DMS files may contain stored queries known as views. A view combines data from one or more tables, and may apply a predicate as well a sorting criterion. How this is actually done in SQL will be obvious to database afficiandos; for this specification it suffices to note that a view looks just like a table when reading a DMS file, so the views will be described in terms of the data in their columns, just as for tables. Importantly, views cannot be written to directly; one instead updates the tables to which they refer.

7.1.1 Units

Of the five datatypes available in SQLite, DMS uses three: INTEGER, a signed 64-bit int; FLOAT, a 64-bit IEEE floating point number; and TEXT, a UTF8 string. In addition, any value other than a primary key can be NULL, indicating that no value is stored for that row and column. A NULL value is allowed in the DMS file but might be regarded as an invalid value by a particular application; for example, Desmond make no use of the atomic number column in the particle table, but Viparr requires it.

Because DMS is used to store dimensionful quantities, we must declare a system of units. The units in DMS reflect a compromise between an ideal of full consistency and the reality of practical usage; in particular, the mass unit is amu, rather than an algebraic combination of the energy, length, and time units.

Dimension	Unit
TIME	picosecond
CHARGE	electron charge
LENGTH	Angstrom
ENERGY	thermochemical kcal/mol
MASS	atomic mass unit (amu)

7.1.2 Versioning

Beginning with msys 1.7.0, a **dms_version** table is included in DMS files written by msys. The version table schema consists of a major and minor version number, and will correspond to the major and minor version of msys. Going forward, msys will refuse to load DMS files whose version is is higher than the msys version; thus, if and when msys reaches version 1.8, files written by that version of msys will not (necessarily) be readable by msys 1.7. There is always the possibility that forward compatibility could be ported to a later msys 1.7 version. Backward compatibility with older dms versions will always be maintained.

The DMS versioning scheme serves to prevent problems arising from new data structures being added to the DMS file in newer versions of msys which are not properly recognized by older versions. For example, the **nonbonded_combined_param** table was added in msys 1.4.0, but because there was no dms version string at that time, older versions of msys would have treated that table as an auxiliary table instead of as a set of overrides to the nonbonded table.

7.2 Chemical Structure

The DMS file contains the identity of all particles in the structure as well as their positions and velocities in a global coordinate system. The particle list includes both physical atoms as well as pseudoparticles such as virtual sites and drude particles. The most important table has the name **particle**; all other tables containing additional particle properties or particle-particle interactions refer back to records in the **particle** table. References to particles should follow a naming convention of *p0*, *p1*, *p2*, … for each particle referenced.

7.2.1 Particles

The **particle** table associates a unique *id* to all particles in the structure. The ids of the particles must all be contiguous, starting at zero. The ordering of the particles in a DMS file for the purpose of, e.g., writing coordinate data, is given by the order of their ids. The minimal schema for the **particle** table is:

Column	Type	Description
anum	INTEGER	atomic number
id	INTEGER	unique particle identifier
msys_ct	INTEGER	ct identifier
x	FLOAT	x-coordinate in LENGTH
y	FLOAT	y-coordinate in LENGTH
z	FLOAT	z-coordinate in LENGTH
mass	FLOAT	particle mass in MASS
charge	FLOAT	particle charge in CHARGE
vx	FLOAT	x-velocity in LENGTH/TIME
vy	FLOAT	y-velocity in LENGTH/TIME
vz	FLOAT	z-velocity in LENGTH/TIME
nbtype	INTEGER	nonbonded type
resid	INTEGER	residue number
resname	TEXT	residue name
chain	TEXT	chain identifier
name	TEXT	atom name
formal_charge	FLOAT	format particle charge

Msys organizes chemical system into a hierarchical structure. The hierarchy has the following names: ct, chain, residue, atom. Rows in the particle table of a dms file are mapped into these four structural levels according to one or more columns in the particle table. The relevant columns for each structural level are:

structure	columns
ct	msys_ct
chain	chain,segid
residue	resname,resid,insertion
atom	id

Of these columns, only the id column is required. The id will be contiguous and start at 0. The id determines the order of the particles in the structure, important when dealing with simulation trajectories. The other columns are treated as 0/empty string if not present.

Particles are mapped to ct object according to their msys_ct value. Within a ct, there will be one chain object for each distinct (chain,segid) tuple. Within a chain object, there will be one residue object for each distinct (resname,resid,insertion) tuple. For example, in the following hypothetical particle table with most columns elided:

id	chain	resid
0	A	1
1	A	1
2	B	1
3	C	2
4	B	2

there would be one ct containing three chains with 1, 2, and 1 residues in chains A, B and C, respectively. Residues A/1, B/1, B/2, and C/2 would contain atoms 0-1, 2, 3 and 4.

7.2.2 Bonds

Column	Type	Description
p0	INTEGER	1st particle id
p1	INTEGER	2nd particle id
order	FLOAT	bond order

The **bond** table specifies the chemical topology of the system. Here, the topology is understood to be independent of the forcefield that describes the interactions between particles. Whether a water molecule is described by a set of stretch and angle terms, or by a single constraint term, one would still expect to find entries in the **bond** table corresponding to the two oxygen-hydrogen bonds. Bonds may also be present between a pseudoatom and its parent particle or particles; these bonds aid in visualization.

The *p0* and *p1* values correspond to an id in the **particle** table. Each *p0*, *p1* pair should be unique, non-NULL, and satisfy *p0* < *p1*.

7.2.3 The global cell

Column	Type	Description
id	INTEGER	vector index (0, 1, or 2)
x	FLOAT	*x* component in LENGTH
y	FLOAT	*y* component in LENGTH
z	FLOAT	*z* component in LENGTH

The global_cell table specifies the dimensions of the periodic cell in which particles interact. There shall be three records, with *id* 0, 1, or 2; the primary key is provided since the order of the records matters, and one would otherwise have difficulty referring to or updating a particular record in the table.

7.2.4 Additional particle properties

Additional per-particle properties not already specified in the **particle** table should be added to the particle table as columns.

Column	Type	Description
grp_temperature	INTEGER	temperature group
grp_energy	INTEGER	energy group
grp_ligand	INTEGER	ligand group
grp_bias	INTEGER	force biasing group
occupancy	FLOAT	pdb occupancy value
bfactor	FLOAT	pdb temperature factor

7.2.5 Ct properties

The **msys_ct** table holds properties of each *ct* in the System. The *msys_ct* field in the **particle** table maps each particle to a ct. The **msys_ct** table has only one required column, *msys_name*, which holds the name of the ct. Additional columns are created in this table to hold ct properties.

7.3 Forcefields

A description of a forcefield comprises the functional form of the interactions between particles in a chemical system, the particles that interact with a given functional form, and the parameters that govern a particular interaction. At a higher level, interactions can be described as being *local* or *nonlocal*. Local particle interactions in DMS are those described by a fixed set of n-body terms. These include bonded terms, virtual sites, constraints, and polar terms. Nonlocal interactions in principle involve all particles in the system, though in practice the potential is typically range-limited. These include van der Waals (vdw) interactions as well as electrostatics.

7.3.1 Metatables

In order to evaluate all the different forces between particles, a program needs to be able to find them within a DMS file that may well contain any number of other auxiliary tables. The DMS format solves this problem by providing a set of ‘metatables’ containing the names of force terms required by the forcefield as well as the names of the tables in which the force term data is found. The force terms are placed into one of four categories: bonded terms, constraints, virtual sites, polar terms, described below.

Metatable name	Description
bond_term	Interactions representing bonds between atoms, including stretch, angle, and dihedral terms, as well as 1-4 pairs and position restraints.
constraint_term	Constraints on bonds and/or angles involving a reduction in the number of degrees of freedom of the system.
virtual_term	Similar to a constraint; a set of parameters describing how a pseudoparticle is to be positioned relative to a set of parent atoms.
polar_term	Similar to a virtual site; a set of parameters describing how a pseudoparticle moves relative to its parent atoms.
nonbonded_table	Additional or alternative nonbonded interactions. Present only if such alternative tables are present.

Each table name corresponding to the values in the local term metatables is the designated string for a particular functional form. The required columns for these tables is given in the next section. Note that creators of DMS files are free to implement the schema as an SQL view, rather than as a pure table; a reader of a DMS file should not assume anything about how the columns in the table name have been assembled.

7.3.2 Bond Terms

Stretch terms

The vibrational motion between two atoms (i, j) is represented by a harmonic potential as:

$$V_s(r_{ij}) = f_c(r_{ij} - r_0)^2$$

where f_c is the bond force constant in units of $\mathrm{Energy}/\mathrm{Length}^2$ and r_0 is the equilibrium bond distance. Terms in `stretch_harm` are evaluated using this potential.

Table 1: Schema for the `stretch_harm` table

name	type	description
r0	FLOAT	equilibrium separation (LENGTH)
fc	FLOAT	force constant (ENERGY / LENGTH^2)
p0	INTEGER	1st particle
p1	INTEGER	2nd particle
constrained	INTEGER	if nonzero, constrained; default 0

Stretch terms that overlap with constraints should have the constrained field set to 1. Applications that evaluate constraint terms need not evaluate `stretch_harm` records that are marked as constrained.

Angle terms

The angle vibration between three atoms (i, j, k) is evaluated as:

$$V_a(\theta_{ijk}) = f_c(\theta_{ijk} - \theta_0)^2$$

where f_c is the angle force constant in $\mathrm{Energy}/\mathrm{Radians}^2$ and θ_0 is the equilibrium angle in radians. Beware, the explicit use of the θ_{ijk} angle will introduce discontinuities in the potential at $\theta_{ijk} = \pm\pi$. Terms in `angle_harm` are evaluated using this potential.

Table 2: Schema for the `angle_harm` table

name	type	description
theta0	FLOAT	equilibrium angle (DEGREES)
fc	FLOAT	force constant (ENERGY / RADIAN^2)
p0	INTEGER	1st particle
p1	INTEGER	2nd particle
p2	INTEGER	3rd particle
constrained	INTEGER	constrained if nonzero; default 0

The $p0$ particle forms the vertex. Angle terms that overlap with constraints should have the constrained field set to 1. Applications that evaluate constraint terms need not evaluate `angle_harm` records that are marked as constrained.

Proper dihedral terms

Two functional forms for calculating proper and improper torsion potential terms are specified. The first is:

$$V_t(\phi_{ijkl}) = f_{c0} + \sum_{n=1}^{6} f_{cn} \cos(n\phi_{ijkl} - \phi_0)$$

where $f_{c0} \ldots f_{c6}$ are dihedral angle force constants in units of Energy and ϕ_0 is the equilibrium dihedral angle in radians. The ϕ angle is formed by the planes $p0$–$p1$–$p2$ and $p1$–$p2$–$p3$. Terms in `dihedral_trig` are handled by this potential function.

Table 3: Schema for the `dihedral_trig` table.

name	type	description
phi0	FLOAT	phase (DEGREES)
fc0	FLOAT	order-0 force constant (ENERGY)
fc1	FLOAT	order-1 force constant (ENERGY)
fc2	FLOAT	order-2 force constant (ENERGY)
fc3	FLOAT	order-3 force constant (ENERGY)
fc4	FLOAT	order-4 force constant (ENERGY)
fc5	FLOAT	order-5 force constant (ENERGY)
fc6	FLOAT	order-6 force constant (ENERGY)
p0	INTEGER	1st particle
p1	INTEGER	2nd particle
p2	INTEGER	3rd particle
p3	INTEGER	4th particle

Improper dihedral terms

The second dihedral functional form is:

$$V_t(\phi_{ijkl}) = f_c(\phi_{ijkl} - \phi_0)^2 \tag{7.1}$$

where f_c is the dihedral angle force constant in units of Energy/radians^2 and ϕ_0 is the equilibrium dihedral angle in degrees (converted to radians internally). The ϕ angle is formed by the planes $p0$–$p1$–$p2$ and $p1$–$p2$–$p3$. Terms in `improper_harm` are handled by this potential function.

The harmonic dihedral term given in Equation (7.1) can lead to accuracy issues if f_c is too small, or if initial conditions are poorly chosen due to a discontinuity in the definition of the first derivative with respect to i in ϕ_{ijkl} near $\phi_0 \pm \pi$.

Table 4: Schema for the `improper_harm` table.

name	type	description
phi0	FLOAT	equilibrium separation (DEGREES)
fc	FLOAT	force constant (ENERGY / RADIANS^2)
p0	INTEGER	1st particle
p1	INTEGER	2nd particle
p2	INTEGER	3rd particle
p3	INTEGER	4th particle

CMAP torsion terms

CMAP is a torsion-torsion cross-term that uses a tabulated energy correction. It is found in more recent versions of the CHARMM forcefield. The potential function is given by:

$$V_c(\phi,\psi) = \sum_{n=1}^{4}\sum_{m=1}^{4} C_{nm}\left(\frac{\psi-\psi_L}{\Delta_\psi}\right)^{n-1}\left(\frac{\phi-\phi_L}{\Delta_\phi}\right)^{m-1}$$

where C_{nm} are bi-cubic interpolation coefficients derived from the supplied energy table, ϕ is the dihedral angle formed by particles $p0 \ldots p3$, and ψ is the dihedral angle formed by particles $p4 \ldots p7$. The grid spacings are also derived from the supplied energy table. Terms in `torsiontorsion_cmap` are handled by this potential function.

The `cmap` tables for each term can be found in `cmapN`, where `N` is a unique integer identifier for a particular table (multiple `cmap` terms in `torsiontorsion_cmap` can refer to a single `cmapN` block). The format of the `cmap` tables consists of two torsion angles in degrees and an associated energy. `cmap` tables must begin with both torsion angles equal to -180.0 and increase fastest in the second torsion angle. The grid spacing must be uniform within each torsion coordinate, but can be different from the grid spacing in other torsion coordinates. More information can be found in [Bro-2004].

Table 5: Schema for each of the tables holding the 2D `cmap` grids

name	type	description
phi	FLOAT	phi coordinate (DEGREES)
psi	FLOAT	psi coordinate (DEGREES)
energy	FLOAT	energy value (ENERGY)

The CHARMM27 forcefield uses six `cmap` tables, which have names `cmap1`, `cmap2`, ..., `cmap6` in DMS.

Table 6: Schema for the `torsiontorsion_cmap` table

name	type	description
cmap	INTEGER	name of cmap table
p0	INTEGER	1st particle
p1	INTEGER	2nd particle
p2	INTEGER	3rd particle
p3	INTEGER	4th particle
p4	INTEGER	5th particle
p5	INTEGER	6th particle
p6	INTEGER	7th particle
p7	INTEGER	8th particle

Position restraint terms

Particles can be restrained to a given global coordinate by means of the restraining potential:

$$V_r(x,y,z) = \frac{\lambda}{2}(f_{cx}(x-x_0)^2 + f_{cy}(y-y_0)^2 + f_{cz}(z-z_0)^2)$$

where f_{cx}, f_{cy}, f_{cz} are the force constants in $\mathrm{Energy/Length}^2$ and x_0, y_0, z_0 are the desired global cell coordinates (units of Length). λ is a pure scaling factor, set to 1 by default. Terms in `posre_harm` are evaluated using this potential.

Table 7: Schema for the `posre_harm` table

name	type	description
fcx	FLOAT	X force constant in ENERGY/LENGTH^2
fcy	FLOAT	Y force constant in ENERGY/LENGTH^2
fcz	FLOAT	Z force constant in ENERGY/LENGTH^2
p0	INTEGER	restrained particle
x0	FLOAT	x reference coordinate
y0	FLOAT	y reference coordinate
z0	FLOAT	z reference coordinate

Pair 12–6 terms

Pair terms in `pair_12_6_es` allow for modifying the normally calculated nonbonded interactions either by scaling the interaction energy, or by specifying new coefficients to use for a particular pair. This partial or modified energy is calculated in addition to the normally calculated interaction energy.

The functional form of the pair potential is:

$$V_p(r_{ij}) = \frac{a_{ij}}{r_{ij}^{12}} - \frac{b_{ij}}{r_{ij}^{6}} + \frac{q_{ij}}{r_{ij}}$$

The a_{ij}, b_{ij}, and q_{ij} coefficients are specified in the `pair_12_6_es` table.

Table 8: Schema for the `pair_12_6_es` table

name	type	description
aij	FLOAT	scaled LJ12 coeff in ENERGY LENGTH^{12}
bij	FLOAT	scaled LJ6 coeff in ENERGY LENGTH^6
qij	FLOAT	scaled product of charges in CHARGE^2
p0	INTEGER	1st particle
p1	INTEGER	2nd particle

Flat-bottomed harmonic well

The functional form of the flat-bottomed harmonic angle term is $V = |d|^2$ where

$$d = \begin{cases} (\theta - \theta_0 + \sigma) & \text{where } \theta - \theta_0 < -\sigma \\ 0 & \text{where } -\sigma <= \theta - \theta_0 < \sigma \\ (\theta - \theta_0 - \sigma) & \text{where } \sigma <= \theta - \theta_0 \end{cases}$$

and θ_0 is in radians.

Table 9: Schema for the `angle_fbhw` table

name	type	description
fc	FLOAT	force constant in ENERGY/RADIANS^2
theta0	FLOAT	equilibrium angle in DEGREES
sigma	FLOAT	half-width of flat-bottomed region in DEGREES
p0	INTEGER	first particle
p1	INTEGER	second particle
p2	INTEGER	third particle

The functional form of the FBHW improper term is $V = f_c d^2$ where

$$d = \begin{cases} (\phi - \phi_0 + \sigma) & \text{where } \phi - \phi_0 < -\sigma \\ 0 & \text{where } -\sigma <= \phi - \phi_0 < \sigma \\ (\phi - \phi_0 - \sigma) & \text{where } \sigma <= \phi - \phi_0 \end{cases}$$

The improper dihedral angle phi is the angle between the plane ijk and jkl. Thus fc is in ENERGY and phi0 is in RADIANS.

Table 10: Schema for the `improper_fbhw` table

name	type	description
fc	FLOAT	force constant in ENERGY/RADIANS^2
phi0	FLOAT	equilibrium improper dihedral angle in DEGREES
sigma	FLOAT	half-width of flat-bottomed region in DEGREES
p0	INTEGER	first particle
p1	INTEGER	second particle
p2	INTEGER	third particle
p3	INTEGER	fourth particle

The functional form of the FBHW posre term is $V = f_c/2d^2$ where

$$d = \begin{cases} |r - r0| - \sigma & \text{where } |r - r0| > \sigma \\ 0 & \text{where } |r - r0| <= \sigma \end{cases}$$

This is not as general as the fully harmonic position restraint term in that you can't specify different force constants for the three coordinate axes.

Table 11: Schema for the `posre_fbhw` table

name	type	description
fc	FLOAT	force constant in ENERGY/LENGTH^2
x0	FLOAT	equilibrium x coordinate in LENGTH
y0	FLOAT	equilibrium y coordinate in LENGTH
z0	FLOAT	equilibrium z coordinate in LENGTH
sigma	FLOAT	radius of flat-bottomed region in LENGTH
p0	INTEGER	restrained particle

7.3.3 Exclusions

Exclusion terms in `exclusion` are used to prevent calculation of certain non bonded interactions at short ranges. The excluded interactions are typically those that involve particles separated by one or two bonds, as these interactions are assumed to be adequately modeled by the stretch and angle terms described above.

Table 12: Schema for the `exclusion` table

name	type	description
p0	INTEGER	1st particle
p1	INTEGER	2nd particle

It is required that $p0 < p1$ for each term, and every $p0$, $p1$ pair should be unique.

7.3.4 Constraint Terms

Constraints fix the distances between pairs of particles according to a topology of rigid rods:

$$||r_i - r_j|| = d_{ij}$$
$$||r_k - r_l|| = d_{kl}$$
$$\ldots$$

The topologies that can be constrained are:

- **AHn**: n particles connected to a single particle, with $1 \leq n \leq 8$.
- **HOH**: three mutually connected particles.

The schemas in the DMS file for `AHn` and `HOH` constraints are shown in *Schema for the constraint_ahN tables* and *Schema for the constraint_hoh (rigid water) table*, respectively. Each record in the `AHn` table gives the length of the bonds between a single parent atom and `n` child atoms. Each record in the `HOH` table gives the angle between the two O-H bonds and the respective bonds lengths.

Table 13: Schema for the `constraint_ahN` tables

name	type	description
r1	FLOAT	A-H1 distance
r2	FLOAT	A-H2 distance
…		
rN	FLOAT	A-HN distance
p0	INTEGER	id of parent atom
p1	INTEGER	id of H1
p2	INTEGER	id of H2
…		
pN	INTEGER	id of HN

Table 14: Schema for the `constraint_hoh` (rigid water) table

name	type	description
theta	FLOAT	H-O-H angle in DEGREES
r1	FLOAT	O-H1 distance
r2	FLOAT	O-H2 distance
p0	INTEGER	id of heavy atom (oxygen)
p1	INTEGER	id of H1
p2	INTEGER	id of H2

A constrained particle is no longer free; each such particle has $3 - m/2$ degrees of freedom, where m is the number of independent constraints involved; for example, a pair of particles having only one distance constraint between them has five degrees of freedom. Constraints thus affect the calculation of the instantaneous temperature and pressure, which depend on the number of degrees of freedom.

The `AHnR` constraints are versions of the `AHn` constraints with additional distances sufficient to create a rigid body. As with water constraints, the alternative Reich algorithm is used by default.

Table 15: Schema for the `constraint_ah1R` table

name	type	description
r1	FLOAT	A-H1 distance
p0	INTEGER	id of parent atom
p1	INTEGER	id of H1

Table 16: Schema for the `constraint_ah2R` table

name	type	description
r1	FLOAT	A-H1 distance
r2	FLOAT	A-H2 distance
r3	FLOAT	H1-H2 distance
p0	INTEGER	id of parent atom
p1	INTEGER	id of H1
p2	INTEGER	id of H2

Table 17: Schema for the `constraint_ah3R` table

name	type	description
r1	FLOAT	A-H1 distance
r2	FLOAT	A-H2 distance
r3	FLOAT	A-H3 distance
r4	FLOAT	H1-H2 distance
r5	FLOAT	H1-H3 distance
r6	FLOAT	H2-H3 distance
p0	INTEGER	id of parent atom
p1	INTEGER	id of H1
p2	INTEGER	id of H2
p3	INTEGER	id of H3

7.3.5 Virtual sites

```
force.virtual = {
  exclude = [ ... ] # optional names to remove
  include = [ ... ] # optional names to add
  # typically, no other per-term arguments required
}
```

Virtual sites, a form of pseudoparticle, are additional off-atom interaction sites that can be added to a molecular system. These sites can have charge or van der Waals parameters associated with them; they are usually massless. The TIP4P and TIP5P water models are examples that contain one and two off-atom (virtual) sites, respectively. Because these sites are massless, it is necessary to redistribute any forces acting on them to the particles used in their construction. (A consistent way to do this can be found in [Gun-1984].) The virial in most cases must also be modified after redistributing the virtual site force.

The types of virtual site placement routines are described below.

lc2 virtual site

The lc2 virtual site is placed some fraction a along the vector between two particles (i, j).

$$\vec{r}_v = (1 - c_1)\vec{r}_i + c_1\vec{r}_j$$

Table 18: Schema for `virtual_lc2` records

name	type	description
c1	FLOAT	coefficient 1
p0	INTEGER	pseudoparticle id
p1	INTEGER	parent atom i
p2	INTEGER	parent atom j

Pseudoparticle $p0$ is placed at the fractional position $c1$ along the interpolated line between $p1$ and $p2$.

lc3 virtual site

The lc3 virtual site is placed some fraction a and b along the vectors between particles (i, j) and (i, k) respectively. The virtual particle lies in the plane formed by (i, j, k).

$$\vec{r}_v = (1 - c_1 - c_2)\vec{r}_i + c_1\vec{r}_j + c_2\vec{r}_k$$

Table 19: Schema for the `virtual_lc3` table

name	type	description
c1	FLOAT	coefficient 1
c2	FLOAT	coefficient 2
p0	INTEGER	pseudoparticle id
p1	INTEGER	parent atom i
p2	INTEGER	parent atom j
p3	INTEGER	parent atom k

fdat3 virtual site

The fdat3 virtual site is placed at a fixed distance d from particle i, at a fixed angle θ defined by particles (v, i, j) and at a fixed torsion ϕ defined by particles (v, i, j, k).

$$\vec{r}_v = \vec{r}_i + a\vec{r}_1 + b\vec{r}_2 + c\vec{r}_2 \times \vec{r}_1$$

where $\vec{r}_1$ and $\vec{r}_2$ are unit vectors defined by

$$\begin{aligned}\vec{r}_1 &\propto \vec{r}_j - \vec{r}_i \\ \vec{r}_2 &\propto \vec{r}_k - \vec{r}_j - (\vec{r}_k - \vec{r}_j) \cdot \vec{r}_1\vec{r}_1\end{aligned}$$

The coefficients a, b and c above are defined as $a = d\cos(\theta)$, $b = d\sin(\theta)\cos(\phi)$ and $c = d\sin(\theta)\sin(\phi)$.

Table 20: Schema for the `virtual_fdat3` table

name	type	description
c1	FLOAT	d coefficient
c2	FLOAT	θ coefficient
c3	FLOAT	ϕ coefficient
p0	INTEGER	pseudoparticle id
p1	INTEGER	parent atom i
p2	INTEGER	parent atom j
p3	INTEGER	parent atom k

out3 virtual site

The out3 virtual site can be placed out of the plane of three particles (i, j, k).

$$\vec{r_v} = \vec{r_i} + c_1(\vec{r_j} - \vec{r_i}) + c_2(\vec{r_k} - \vec{r_i}) + c_3(\vec{r_j} - \vec{r_i}) \times (\vec{r_k} - \vec{r_i})$$

Table 21: Schema for the `virtual_out3` table

name	type	description
c1	FLOAT	coefficient 1
c2	FLOAT	coefficient 2
c3	FLOAT	coefficient 3
p0	INTEGER	pseudoparticle id
p1	INTEGER	parent atom i
p2	INTEGER	parent atom j
p3	INTEGER	parent atom k

7.3.6 Nonbonded interactions

The functional form for nonbonded interactions, as well as the tables containing the interaction parameters and type assignments, are given by the fields in the **nonbonded_info** table, shown below:

Column	Type	Description
name	TEXT	nonbonded functional form
rule	TEXT	combining rule for nonbonded parameters

There should exactly one record in the **nonbonded_info** table. Like the local interaction tables, the *name* field indicates the functional form of the nonbonded interaction type. If the particles have no nonbonded interactions, *name* should have the special value *none*.

The parameters for nonbonded interactions will be stored in a table called **nonbonded_param**, whose schema depends on the value of *name* in **nonbonded_info**. All such schemas must have a primary key column called *id*; there are no other restrictions.

The *nbtype* column in the **particle** table gives the nonbonded type assignment. The value of the type assignment must correspond to one of the primary keys in the **nonbonded_param** table.

Typically, the parameters governing the nonbonded interaction between a pair of particles is a simple function of the parameters assigned to the individual particles. For example, in a Lennard-Jones functional form with parameters *sigma* and *epsilon*, the combined parameters are typically the arithmetic or geometric mean of *sigma* and *epsilon*. The required approach is obtained by the application from the value of *rule* in **nonbonded_info**.

For the interaction parameters that cannot be so simply derived, a table called **nonbonded_combined_param** may be provided, with a schema shown in Table~ref{tab:combinedparam}. Like the **nonbonded_param** table, the schema of **nonbonded_combined_param** will depend on the functional form of the nonbonded interactions, but there are two required columns, which indicate which entry in **nonbonded_param** are being overridden. Only *param1* and *param2* are required; the remaining columns provide the interaction-dependent coefficients.

Column	Type	Description
param1	INTEGER	1st entry in **nonbonded_param** table
param2	INTEGER	2nd entry in **nonbonded_param** table
coeff1	FLOAT	first combined coefficient
		other combined coefficients…

Table 22: Schema for the **vdw_12_6** nonbonded type

name	type	description
sigma	FLOAT	VdW radius in LENGTH
epsilon	FLOAT	VdW energy in ENERGY

The functional form is $V = a_{ij}/|r|^{12} + b_{ij}/|r|^6$, where a_{ij} and b_{ij} are computed by applying either the combining rule from `nonbonded_info` or the value from `nonbonded_combined_param` to obtain σ and ϵ, then computing $a_{ij} = 4\epsilon\sigma^{12}$ and $b_{ij} = -4\epsilon\sigma^6$.

7.4 Alchemical systems

Methods for calculating relative free energies or energies of solvation using free energy perturbation (FEP) involve mutating one or more chemical entities from a reference state, labeled ‘A’, to a new state, labeled ‘B’. DMS treats FEP calculations as just another set of interactions with an extended functional form. In order to permit multiple independent mutations to be carried out in the same simulation, a ‘moiety’ label is applied to each mutating particle and bonded term.

Any particle whose charge or nonbonded parameters changes in going from state A to state B, is considered to be an alchemical particle and must have a moiety assigned to it. The set of distinct moieties should begin at 0 and increase consecutively. The set of alchemical particles, if any, should be provided in a table called **alchemical_particle** shown below:

Column	Type	Description
p0	INTEGER	alchemical particle id
moiety	INTEGER	moiety assignment
nbtypeA	INTEGER	entry in nonbonded_param for A state
nbtypeB	INTEGER	entry in nonbonded_param for B state
chargeA	FLOAT	charge in the A state
chargeB	FLOAT	charge in the B state

Alchemical bonded terms can be treated by creating a table analogous to the non-alchemical version, but replacing each interaction parameter with an ‘A’ and a ‘B’ version. As a naming convention, the string *alchemical_* should be prepended to the name of the table. An example is given below for **alchemical_stretch_harm** records, corresponding to alchemical harmonic stretch terms with a functional form given by interpolating between the parameters for states A and B.

Column	Type	Description
r0A	FLOAT	equilibrium separation in A state
fcA	FLOAT	force constant in A state
r0B	FLOAT	equilibrium separation in B state
fcB	FLOAT	force constant in B state
p0	INTEGER	1st particle
p1	INTEGER	2nd particle
moiety	INTEGER	chemical group

7.5 References

CHAPTER

EIGHT

RELEASE NOTES

```
1.7.301 Fix dms-select

1.7.300 Add a default 60s timeout to AssignBondOrderAndFormalCharges
        No provenance in etr metadata.
        Add scripts for building binary wheel files.
        Detect when clone() breaks bonds.
        Make protein" selection include ACE and NMA.

1.7.299 Support numpy integers in System.clone()
        Don't write zero-length metadata frames.

1.7.298 Fix for MAE export of empty m_depend fields.

1.7.297 New use_index keyword for clone().

1.7.296 Skip mae ct properties with conflicting type.

1.7.295 Add compare_frames tool.

1.7.294 Don't fail on empty stk.
        Remove phi0 from dihedral_fourier schema.

1.7.293 Add rudimentary psf export.
        Drop support for ddparams.
        Maintain atom order when writing pdb files.

1.7.292 Added softened_stretch_harm schema.
        Improve speed of reading stk cache files
        Isotope support for sdf and to/from oechem.

1.7.291 Fix bug in ct property handling introduced in 1.7.284.

1.7.290 Fix reading of stks with empty framesets.

1.7.289 Drop python2 from build and tests, again.

1.7.288 Restore the python2 build, for now.

1.7.287 Docstring fix for findContactIds.
        Fix very old bug in SSSR which may not have affected anyone.
        Added dms-hmr.
        Add metadata keyword to DtrWriter constructor.
        Use bytearray when unicode conversion fails reading dtr keyvals.
```

(continues on next page)

(continued from previous page)

```
1.7.284 Add msys.ConvertFromRdkit.
        Better handling of ct properties in dms files.

1.7.283 Bring tests/files back into the main repo.
        Fixes and notes for external builds.
        Handle empty string in mae vdwtype.
        Add testModifiedInteractionConsistency to validate.
        Expose metadata keyvals from Python via DtrReader.metadata.
        Add benchmark program.
        Add dms-restrain-dihedrals script.

1.7.282 Support for compressed positions in dtr frames
        New dms-randomized-atoms and dms-diff-ff scripts
        Fix for ConvertToRdkit when vsites are present.

1.7.281 canonicalize atoms and bonds before calculating
          bond orders and formal charges
        bugfix for usage of msys.* in MatchFragments python code

1.7.280 Handle UNREAD as jobstep_id in stk cache.

1.7.279 Ensure ConvertToOEChem gives deterministic order of bonds
        Add documentation on extracting force-field parameters
        Fixed docs for improper definition (DESRESCode4243)
        Drop python2 from build and tests.
        Better error message when addProp fails.

1.7.278 Silence some compile warnings.
        Fix for ParamTable::compare.
        Routine to build amber-style pairs and exclusions.
        Make json format use full precision parsing.
        Make json format handle aux tables.
        Skip writing of blank provenance entries.
        Add a close() method to DtrWriter.
        Handle mae export when ct fields contain whitespace.
        Bump python version.
        Don't prereq a desres-python from lib-python.

1.7.277 Make System.hash sort terms by default.
        dtr-validate: require constraints only if there are hydrogens.
                      Run anton2 tests by default.
        Drop #include of boost/thread.
        Initial implementation of experimental json format.

1.7.276 Store atom name in ConvertToRdkit.
        DtrReader.keyvals() automatically unpacks ETR frames
        Allow ETRs to be written by DtrWriters

1.7.275 Fix stk cache file writing.

1.7.274 Fix dms-tile.
        Fix dms-sequence.
        Bump python versions.

1.7.273 Fix dms-neutralize for neutral systems.
        Support reading mol2 files with no residues.
```

(continues on next page)

(continued from previous page)

```
1.7.272 Support reading bond order 0 in sdf.

1.7.271 Remove old desmond validation routines.
        Python3 fix for dms-select.

1.7.270 Propagate return code in dms-diff.
        Added System.save().
        Fix structure-only for dms-neutralize.
        Documention DMS functional forms.
        Load mathjax in docs.
        Use python3 as executable in tools.

1.7.266 Turn off water-pad by default in Neutralize.
        Fix for within selections when atoms have been deleted.

1.7.265 Support molecular ionic species in Neutralize.
        Add support for VMD-style sequence selection.

1.7.264 Conda support.
        Replace some test files with synthetic data.
        desres-python/3.7.2-08.

1.7.262 Fix dms-neutralize and add --structure-only option.

1.7.261 Added neutralize.Neutralize and dms-neutralize.
        Replace spaces with '_' in mol2 names and resnames.

1.7.259 New FindDistinctFragments interface.

1.7.258 Fix broken dms-info caused by remove of msys.vdw.

1.7.257 Remove dms-scale-vdw and dms-override-vdw.
        Make dms-find-knot agree with dms-validate.
        Added msys.MatchFragments.

1.7.256 Fix for Graph to support viparr-style graph matching.

1.7.255 desres-python/3.7.2-04
        Fix for writing dms files with tables containing more than 10 atoms.

1.7.254 desres-python/3.7.2-02.

1.7.253 Bump desres-python/3.7.0 to 3.7.2.
        Support the != operator in atomsel.
        Fix documentation in molfile.

1.7.252 Fix ParseSDF for python3.

1.7.251 Added vtfplugin.

1.7.250 msys.ConvertFromOEChem now preserves chain and residue information.

1.7.249 Make InChI ignore vsites.

1.7.248 Add consider_stereo flag to FindDistinctFragments.
        Fix missing call to updateFragids in ImportSdf.
```

(continues on next page)

(continued from previous page)

```
1.7.247 Include pseudos when validating against nonbonded contacts [DESRESCode#4042]
        Fix missing shlib version on msys.so.

1.7.246 Fix msys.Load(jobid) for both return types of yas_job.input_ark
        Read CRYSIN records from mol2 files.
        Wrap ImportPDBUnitCell.
        Added periodic option to System.findBonds.
        Add docs on msys.sequence and a few other tools.
        Added color option to Graph.

1.7.245 Use bytearray for dtr unsigned char fields.
        New "degree" selection keyword for bonds between real atoms.

1.7.244 Make "ion" selection atomic number and bond based.
        Change old "ion" selection to "legacy_ion".
        Handle ComputeTopologicalIds for systems with crazy pseudos.
        Handle nonconsecutive resids in mol2 files.
        Avoid crashing in find-contacts for unreasonable coordinates.
        Fix for undecodeable bytes in DtrReader.keyvals [DESRESCode#4002].

1.7.243 Python3.7.

1.7.242 Atomselection bug for 'all'.

1.7.241 Fix for dms-dump.
          Docs for molfile trajectory writing.
          Triclinic support for pfx.
          Add dms-validate check for virtuals in multiple virtual tables.

1.7.240 Added ReplaceTableWithSortedTerms.
          Implement dms-diff in python instead of shell.
          Fix the Graph.atoms method.
          Use inchi/1.05.
          Make msys.Graph check for duplicates.
          Tweaks to the build for external users.

1.7.239 DESRESCode#3974: Fix some broken tool scripts.

1.7.238 Fix broken lib-python3 from 1.7.237.
          DESRESCode#3964: fix mol2 reading and writing.

1.7.237 Use new sconsutils; should affect developers only.

1.7.235 Add a 'moe' option to msys.SaveMol2 to output guanidinium
          groups MOE can will parse correctly. This option is turned
          on by default as it seems _most_ tools handle this variant
          fine.

1.7.233 Fix dms-dump (broken since 1.7.230).
          Added SpatialHash::findPairlist.
          Better verbosity API for validate.

1.7.232 Use python/2.7.15 and 3.6.6 megamodules.
          Update boost from 1.63 to 1.67.
          Add bindings for System.addProvenance.

1.7.230 Added virtual_lc1 schema
```

(continues on next page)

(continued from previous page)

```
          Fix for dms-dump with .sqliterc files
          Unused param fix for v2 compatibility

1.7.229 Added dtr_frame_{from,as}_bytes.

1.7.228 Added msys.FormatDMS.
          Pickled systems use zlib; 3-20x smaller pickles.
          FindDistinctFragment returns full map of equivalent fragments.

1.7.227 Support K+ as an ion residue name.
          Include tempered_nonbonded and modified_interaction in dms-dump
          Add table schemas for constraint_ahnNR

1.7.226 Fix molfile.StkFile for python3
          Use new megamodules (-03c7).
          Check for atom count consistency in stks.
          dtr-validate: add check for constrained hydrogens.
          Add LoadMany for concatenated dms files.

1.7.225 Use 1/1024 fs as the time quantum for dtr trajectory processing.
          Make msys.SaveDMS(..., unbuffered=True) around 10x faster.
          Remove vestigial ExportDMSMany reference.
          Make SystemImporter python bindings work.

1.7.224 msys.ConvertFromOEChem will no longer complain about
          molecules not having hydrogens that don't actually have
          hydrogens, e.g., O=S=O.
          msys.ConvertToOEChem will always set the dimension of the
          molecule to 3D as msys really only traffics in 3D molecules.

1.7.223 Yet another fix for double precision in timekeys files.
          Expose error message from numpy version clashes

1.7.222 msys.ConvertToOEChem can take a list of atoms to convert to convert a subset␣
↪much faster.

1.7.221 msys.Convert(To/From)OEChem will preserve the molecule title
          Fix SaveDMS for unbuffered=True.
          Added (..., sanitize=True) option to msys.ConvertToRdkit.

1.7.220 Added msys.ConvertFromOEChem (DESRESCode#3785).

1.7.219 Report strerror(errno) when reading dms files (DESRESInfra#39849)

1.7.218 Write .dms.gz files using gzip.  DESRESCode#3771.
          Added rigid_explicitN schema.
          All dms-tools and msys.Load support jobids as arguments.
          Sort atoms and bonds by atom ids.
          More precise handling of timekeys in large stks.

1.7.216 Fixing msys.ConvertToRdkit to work with RDKit 2017.09.
          Sanity check added to Chem.DetectBondStereoChemistry was
          being tripped and needed to be worked around.

          Refactored test cases and making sure to run RDKit and
          OEChem conversion test cases.
```

(continues on next page)

(continued from previous page)

```
1.7.215 Removed v7 stk cache code and added utility to rename
          a dtr in an stk cache file.

1.7.214 Make msys.InChI include stereo by default (SNon=True).
          Fix mol2 handling of resids (DESRESCode#3747).

1.7.213 Bump megamodule versions.
          Don't filter metals in bond order assigner (DESRESCode#3713).
          Ignore excluded knots in dms-validate (DESRESCode#3566).
          Added an ignore_excluded option to findContactIds()
          Fix altloc writing (DESRESCode#3626).
          Fix pdb download.
          Expose SystemImporter from python.

1.7.212 Added msys.ConvertToOEChem.
          Added --without-pos option to dms-dump.

1.7.211 Optionally compute and store resonant charge and bond order.

1.7.210 python3 support.

1.7.208 Added molfile.list_fields()
          Make DtrReader.frame(...) not panic if the FORMAT is empty.
          Fixed reading of dms files with table properties.
          Use new sphinx-rtd-theme for fixed doc search.
          Avoid a deadly exception when writing a dms with missing nbtypes.
          Handle variable-sized frames in etr files.

1.7.207 DESRESCode#3590 - changes from schrodinger.

1.7.206 Bring back GuessHydrogenPositions
          New dms-solvate for those who can't use viparr.
          dms-tile checks for reasonable box size.
          DESRESCode#3571 : Added dms-validate case for massless atoms
          DESRES_ROOT is deprecated, have the cache query DESRES_LOCATION
          Bugfix in bond order assigner
          Make old, non-monotonic stks only generate warning.

1.7.205 Added PyCapsule interface for easier pybind11 integration.
          Added msys.System.hash() function to replace dms-hash.

1.7.204 Added Cl- to 'ion' atom selection.
          Using cent7 garden modules.

1.7.203 Fix for dms-frame broken by DESRESCode#3486 in previous release.
        Double precision support for SpatialHash in C++.
        Eliminate dms-hash.

1.7.202 DESRESCode#3486 - changes from schrodinger.
        Added SpatialHash::findContactsReuseVoxels.
        dms-validate skips knot check when system is alchemical.
        New libmsys-core and versioned libmsys.
        gcc/6.3 portability.
        Remove cealign.

1.7.157 DESRESCode#3438 - fix reading sdf files without trailing delim.
        DESRESCode#3472 - fix msys.System.guessBonds.
```

(continues on next page)

(continued from previous page)

```
        DESRESCode#3450 - pickle support for residues and chains.
        DESRESCode#3461 - fix ComputeTopologicalIds bindings.
        DESRESCode#3452 - use garden flex and bison for CentOS7 support.
        use desres-python/2.7.13-02.

1.7.156 Fix build of docs.

1.7.155 Python/2.7.12-03st.

1.7.154 Python/2.7.12-02st, gcc/5.2.0-33x, boost/1.59-06.

1.7.153 Fixed reading of dms files with table properties.
        Experimental new msys to rdkit converter.
        Experimental new IndexedFileLoader.
        Support for ETR writing.
        Added ElectronegativityForElement.
        Don't write mae files with bad key names.
        Rip out resonant_charge and resonant_order.
        Removed msys2mol2 - use dms-select instead.
        Don't grow the SpatialHash grid too large.
        SpatialHash input checks.
        Fixes for smiles charge parsing.
        Fix LoadDMS(buffer).

1.7.149 Pass --reorder to dms-dump in dms-diff.
        Handle coordinate-only Amber crd files. (DESRESCode#3183).
        Fix dms-frame (DESRESCode#3205).

1.7.148 Fixed typo in dtr-validate.

1.7.147 C++: TermTable::atomsFast() and paramFAST().
        Pickle support for Systems (DESRESCode#3148).
        Include all of msys module in the docs.
        Fixes for Mac build/test.

1.7.145 Use just SSE4_1 for findContacts since drdb has only that.

1.7.144 C++: Replace boost::shared_ptr with std::shared_ptr.
        Better smiles support (DESRESCode#3109)
        SSE4 acceleration of findContacts.
        Fix AnnotatedSystem.repr (DESRESCode#3108)
        Added optional weights argument to pfx.aligned_rmsd.
        Throw exception on empty atom selections.

1.7.143 C++: Spatial hash doesn't compute the square root anymore.
        C++: Fix installation of atomsel headers.
        Skip over newlines and tabs in atom selections.

1.7.142 NEW: msys-fetch-pdb tool
        NEW: SpatialHash.findContacts (supports minimum image checks)
        FIXED: parsing of negative integers in atom selections
        FIXED: dms-grease periodic wrapping.
        IMPROVED: error reporting in atom selections.
        Tracking ticket: DESRESCode#3040.

1.7.141 Fix incorrect check for leading zero in json int.
        Disable multithreaded molfile.read_frames.
```

(continues on next page)

(continued from previous page)

```
        Handle negative integers in atom selections.
        Report line number in sdf import errors.
        Fix empty atom selections in System.atomsel.
        Fix preservation of absolute stereo in sdf I/O.
        Move alchemical system generation to sw/libs/atommatch.

1.7.140 Using Python/2.7.11-03st
        DESRESCode#2857: Fix msys.Term.system property.
        DESRESCode#2779: dms-replicate sets resid, chain or ct as needed.
        C++: Move ThreeRoe out of dtrplugin header.

1.7.139 Make molfile use sequential access for dtr.
        Fix bug with ETRs if no TSS values followed FORMAT.

1.7.138 mol2 processing fixes: preserve sybyl_type; fix aromatic bond type.
        Make dms-frame complain about -t or -n without -i.
        Added dtr-validate to tools documentation.

1.7.137 SDF file processing just uses LoadMany now.
        ScanSDF is gone; use LoadMany, to read, FormatSDF to write.

1.7.136 Fix bug in System.atomsel when alignment produces reflections.

1.7.135 System.selectIds takes optional pos and box.
        The 'fragment' atom selection keyword is now a synonym for 'fragid'.
        dtr-validate now takes only a single path.
        Molfile handles single-row seqfiles.
        Make dms-validate handle systems with virtuals more intelligently.

1.7.134 Double the speed of within searches for small target selections.

1.7.133 Don't put inchi strings in alchemical dms.

1.7.132 rst7 support in dms-frame.
        dms-frame no longer wraps by default.
        Added MakeAlchemical python bindings.
        Build with latest OS X compiler.

1.7.131 Added near-workalike fstime and fsdump.
        Added DtrReader.keyvals().
        C++: AnnotatedSystem no longer holds a SystemPtr, and is itself not
        a shared_ptr.
        Made smarts matching around 3x faster.
        Better error checking on archive file writing.
        Added etr as valid extension type for dtr.

1.7.130 Support for alchemical ffio_morsebonds.
        Handle NULL chains, segids, etc. in dms files.
        Allow extra blank line after M END in ScanSDF.
        C++: make pfx::Graph constructible directly from SystemPtr.
        Fix a memory leak affecting all dms file reading (!).

1.7.129 Support for ffio_morsebonds in mae files.
        More efficient term table indexing.
        Support for .gz files in ScanSDF.

1.7.128 Handle stk files containing empty dtrs (DESRESCode#2406).
```

(continues on next page)

(continued from previous page)

```
1.7.127 Added --energy option to dtr-validate.

1.7.126 Alchemical fix for multiple dihedrals.

1.7.125 Fix reading of stk files whose parent directory changed.

1.7.124 Added AllowBadCharges option to AnnotatedSystem.
        Support more valences for heavy atoms.
        Fix writing of sdf files with formal charge=4.
        Fix pfx for very extended system.
        Use pandas.read_csv for eneseq files when available.
        Accomodate empty eneseq files.

1.7.123 New stk caching ready to roll out.
        Added SmartsPattern.match for Python.

1.7.122 Handle long title lines in ScanSDF.

1.7.121 Fixed charge parsing in ScanSDF.
        Added --add-hydrogens and --guess-hydrogen-positions to dms-select.

1.7.120 Added --sorted option to dms-select.
        Fixed .gro box size unit conversion.
        Fixed bond order assignment for systems with bonds to pseudos.

1.7.119 Experimental ETR support and new STK caching - do not use!

1.7.118 Added paramtype atom selection keyword.

1.7.117 More ScanSDF robustness improvements.

1.7.116 Improvements to ScanSDF and FormatSDF.

1.7.115 Add checkpoint option to dtr-validate to turn off delta_t checks.

1.7.114 Make dtr-validate perform some additional checks on timekeys.

1.7.113 dtr-validate backwards compatibility.
        Fix sdf import bug handling entries containing doublet radical.
        Start using the new stk cache.

1.7.112 Disregard difference in type and memo in identifying alchemical terms.
        Accommodate missing tables in both A and B states in alchemical mapping.

1.7.111 Support for gromacs files (gro, g96, trr, trj, xtc).
        Build with C++11.
        Better error handling in dms export.

1.7.110 Stk cache files are placed alongside original stk.
        DtrReader class supports sequential reading for stk.

1.7.109 Revert the 1.7.103 changes to structure_only.  structure_only now
        excludes pseudos.  Also excluded in mae files (previously wasn't).

        Added new without_tables option to Load(), which causes term tables
        to be ignored during loading but otherwise gives you all particles.
```

(continues on next page)

(continued from previous page)

```
1.7.108 Another shot at build fixes.

1.7.107 Build fixes.

1.7.106 Added --unrotate option to dms-frame.

1.7.105 Smiles functionality factored out into its own garden module.

1.7.104 Experimental SMILES parsing.
        Added LoadSDF function.
        The mol2 exporter mistakenly assigned C.cat atom type in some cases.

1.7.103 Loading structure-only now includes pseudos.
        Fix a bug in import where residues were sometimes erroneously split.

1.7.100 No more versioned libmsys.so.

1.7.99 Create a versioned libmsys.so.
       Fix molfile-based reading of dms with pseudos.
       More HBond finder improvements.
       Added raw dtr I/O support for Python.

1.7.98 Better xyz, dx, and Amber prm/rst7 file support.
       HBondFinder allows multiple donor hydrogens.
       Eliminate primegen dependency.

1.7.97 Fix mol2 bond type output.
       Fix molfile xyz plugin.
       Added dx plugin.

1.7.96 Fix mol2 atom type output.

1.7.95 Fix the /lib submodule, which behaved differently under garden.

1.7.94 Bump lp_solve version to get tighter symbol visibility.
       Fix race condition in build script.
       mol2 exporter now always assigns Sybyl atom types.

1.7.93 Hydrogen bond finder.

1.7.92 Added weights option to Pfx.align.
       Fix OSX compilation issues.
       Enhance molfile.Timekeys Python interface.

1.7.91 Added ostream operator to SmallString.

1.7.90 Incorporated molfile into msys.  Use 'from msys import molfile'.
       pbwithin selection now handles rotated unit cell correctly.
       [pb]nearest now handles empty subselection.
       Slight change to SpatialHash interface.

1.7.88 Added SmartsPattern::match().

1.7.87 Use Python/2.7.9-05st.
       Added boost::serialization-based archive format (.msys extension).
```

(continues on next page)

(continued from previous page)

```
1.7.86 Bump to numpy/1.6.2-35A
       Fix tools scripts so they use the right version of numpy and molfile.
       Make the rms routines more general.

1.7.85 Fixed construction of provenance from Python.
       Improvements to mmod_type handling in dms2mae.
       Table properties in TermTable.  New msys_table_properties in dms.
       Incorporate fastjson and pfx into msys.

1.7.84 IMPROVED: Alchemical mapping now merges dummy fragments connected to the same␣
↪real atom into one fragment, and thus retains more permissible terms.

1.7.83 System.topology now returns a pfx.Graph object.

1.7.82 Fixed reading and writing of large (>2GB) dms files.

1.7.80 Fix memory leak in msys.pfx.aligned_rmsd.
       Added Python bindings for CalcDihedralGeometry.

1.7.79 Revert boost version change.

1.7.78 C++ header fixes for spatial_hash.
       C++ - Bump boost and zlib versions.

1.7.77 FIXED: dms-alchemical typos.

1.7.76 NEW: SpatialHash Python class for efficient findWithin and findNearest.
       NEW: System.selectArr to return atom selections as numpy array of ids.
       NEW: C++ within.hxx renamed to spatial_hash.hxx.

1.7.75 FIXED: Allow structure-only saves with partial nonbonded params.
       FIXED: Use sse2 rather than sse4 in new FindWithin code.
       FIXED: FindWithin crashed on empty subselection.
       NEW: keep-none for dms-alchemical.
       NEW: Write msys_file_offset in mol2 files.

1.7.74 FIXED: xyz files now written with element name rather than number.
       FIXED: better bond guessing for systems with virtuals.
       NEW: distance based atom selections got a lot faster.

1.7.72 FIXED: bfactor and occupancy output in pdb files wasn't quite working.
       FIXED: System.append() and PDB importer now always preserve atom order.
       FIXED: permissions on dms-fix-water-residues.

1.7.71 FIXED: write bfactor and occupancy to pdb files.
       NEW: C++ system.hxx no longer includes {term,param}_table.hxx.

1.7.70 FIXED: handling of element names with extraneous characters
       FIXED: addTable should be no-op when table already exists.
       NEW: dms-set takes any supported file type.

1.7.69 FIXED: Fix error handling in certain Python bindings.
       FIXED: dms-validate for check-gruops.
       FIXED: validate output formatting
       FIXED: read/write PDB space group
       NEW: Saving to stdout.sdf writes to stdout.
       NEW: API change for msys.groups
```

(continues on next page)

(continued from previous page)

```
1.7.68 NEW: dms-validate --anton checks for contacts shorter than 1A,
            including those that cross periodic boundaries.
       FIXED: The bond guesser (used during loading of pdb files) sometimes
              handled hydrogens bonded to multiple atoms incorrectly.
       FIXED: System.findContactIds should be stable now.
       FIXED: System.topology was too slow.

1.7.67 FIXED: AppendSystem is smarter about merging global cells.
       FIXED: PDB readers weren't storing the global cell.
       FIXED: pfx.apply() wasn't returning the correct global cell,
              and the Python bindings for rsmd() failed for double
              precision inputs.

1.7.66 FIXED: msys.pfx.rmsd() was invalid when given double precision
              coordinates, and pfx.apply() was not returning the transformed
              unit cell when doing an alignment.

        MODULE SPLITS: atommatch now lives in the atommatch garden module.
                       psfgen now lives in the desres-psfgen garden module.

1.7.65 IMPROVED: dms-validate checks that each water molecule has
                 its own resid and lives in its own residue
       NEW: dms-fix-water-residues for putting each water molecule in
            its own residue.
       FIXED: appending systems now correctly merges nonbonded_info.

1.7.64 FIXED: SDF writer checks for isfinite() before calling floatify.

1.7.63 IMPROVED: SDF reader is more robust and handles multi-line values.

1.7.62 NEW: Bond.other(atom) from Python.
       NEW: C++ interface for FindWithin.

1.7.61 IMPROVED: Across-the-board speedups for all atom selections.
       IMPROVED: Notable improvements to "nearest" selections.
       NEW: "pbnearest" selection.
       NEW: lc5-l schemas.
       FIXED: Loading webpdb crashed in 1.7.60.

1.7.60 NEW: dms-find-knot has a --untie option to remove knots.
       NEW: LoadMany support for PDB files.
       IMPROVED: dms-find-knot has a --ignore-excluded-knots that uses
                 the exclusion table
       IMPROVED: "nearest k to subsel" is much faster.
       IMPROVED: Delete property in ct block using python del operator.

1.7.59 FIXED: mol2 importer choked on resid 0.
       IMPROVED: read and writes TER records between chains in pdb files
       NEW: System.getTable() returns None on missing table.
       FIXED: (C++) don't link against libmolfile or libpython.

1.7.58 IMPROVED: dms-tile permits multiple input files, and requires -o.

1.7.57 IMPROVED: mol2 export includes original resids.

1.7.55 FIXED: dms-thermalize with --remove-drift doesn't accelerate pseudos.
```

(continues on next page)

(continued from previous page)

```
       FIXED: addTableWithSchema accepts user-supplied name.
       IMPROVED: dms-frame gets otuput format from path name.
       NEW: Use pfx instead of periodicfix in dms-frame and other places.

1.7.54 FIXED: pdb export includes insertion code.

1.7.53 FIXED: dms-tile sets nonbonded_info and cell.
       IMPROVED: dms-info shows counts of pseudos and nucleic.
       NEW: Load handles PSF files.
       NEW: ReadPDBCoordinates() function.

1.7.52 CentOS6 portability.

1.7.49 FIXED: mol2 export was missing residues not in the first chain.
       IMPROVED: mol2 export now includes chain name.

1.7.48 FIXED: sdf export writes bond type 4 for resonant bond.
       FIXED: clone() and append() preserve bond resonant_order.

1.7.47 IMPROVED: psfgen.Psfgen got setCoordinates.
       IMPROVED: update to molfile/1.11.16.
       FIXED: psfgen.write() to dms was writing uninitialized global cell.

1.7.43 FIXED: System.name wasn't set when loading a pdb.
       FIXED: The segid atom selection keyword was missing.
       FIXED: The web pdb access code regex was too restrictive.
       FIXED: dms-reorder atoms: reference selection applies to all atoms.
       IMPROVED: load mae files with multiple consecutive m2io_version blocks.
       IMPROVED: Better error reporting of truncated mae files.

1.7.42 FIXED: Added POPS to the lipid atom selection macro.
       FIXED: The structure analyzer recognized waters with pseudos now.
       FIXED: Allow writing mae files with multi-line ct properties.
       IMPROVED: Added share_params option to clone().
       NEW: Can load from the pdb databank by specifying an accession code.

1.7.41 IMPROVED: SaveDMS accepts structure_only flag.
       IMPROVED: dms-validate --desmond runs the dms-check-groups test.
       FIXED: System.delXXX was confused by empty selections.

1.7.40 FIXED: build of command line tool documentation.
       IMPROVED: error recovery in msys.LoadMany (yield None on bad entry)
       IMPROVED: dms-validate checks for nonzero volume.
       IMPROVED: dms-thermalize has a --remove-drift option, on by default.

1.7.39 treetop/57-Anton2

1.7.38 IMPROVED: dms-select replaced its --append option with support for
                 multiple input files.
       IMPROVED: Compilation improvements.

1.7.37 IMPROVED: PDB reader guesses atomic number form name if element is
                 missing.  END and ENDMDL records split PDB into cts.
       IMPROVED: PDB reader and writer handle formal charge in columns 79-80.
       IMPROVED: dms-select has -A/--assign option to assign formal charge
                 and bond order.
       NEW: dms-tile and dms-replicate scripts.
```

(continues on next page)

(continued from previous page)

```
1.7.35 FIXED: dms-dump truncated its fields to 10 characters; now output
              is separated by '|' rather than being formatted into columns.

1.7.34 IMPROVED: annotated system is now handles radicals
                 (bond order assigner still does not)

1.7.33 FIXED: SDF counts line specification

1.7.32 FIXED: SDF exporter writes ct key-values.

1.7.31 FIXED: Read MAE files with fields containing hash symbols.
       NEW: Atommatch takes optional pair of atoms which must be matched.

1.7.30 FIXED: Another maegz fix.

1.7.29 FIXED: MAE export was confused by bonds to pseudos.
       FIXED: Bad image filename for latex2pdf documentation.

1.7.28 FIXED: Handle MAE files with m2io_version block at the end.

1.7.27 FIXED: Documentation of command line tools matches result of -h.
       FIXED: SaveMAE supports lists of Systems again.
       FIXED: dms-find-knot finds knots crossing periodic boundaries.
       IMPROVED: dms-find-knot is much faster.
       NEW: SerializeMAE instead of SaveMAE(path=None) for MAE contents.
       NEW: XYZ file export.
       NEW: dms-posre takes --max-energy and --max-distance options.
       FIXED: dms-grease puts lipids in their own ct, leaving chains unchanged.

1.7.24 IMPROVED: Documentation
       REMOVED: ExportMAE() no longer takes a list argument; use append=True.
       NEW: ExportMAE() takes path=None to return MAE file contents.
       FIXED: Type checks for delAtom, etc.
       FIXED: Don't consider saturated elements for hydrogen addition.

1.7.23 FIXED: bond order assignments for disulfides and hypervalent halogens

1.7.22 FIXED: Handle maegz files containing multiple zlib streams.

1.7.21 NEW: Add dms-atommatch program and msys.atommatch Python library
       FIXED: dms-alchemical keeps more bonded terms involving dummy atoms

1.7.19 FIXED: dms-alchemical handles constraint tables with extra
              informative parameter columns.

1.7.18 FIXED: dms-alchemical handles more complex topologies.
       FIXED: dms-alchemical keeps dihedral terms, not just bonds and angles.

1.7.17 NEW: Python bindings for ComputeTopologicalIds.
       NEW: dms-check-groups to check for valid initial conformation.
       NEW: AddHydrogens routine.
       NEW: Can append to a specific ct, rather than always create new cts.
       IMPROVED: GuessHydrogenPositions works much better now.
       IMPROVED: dms-fix-mass checks for consistency of atomic number and mass.
       FIXED: MAE writer now handles negative atomic numbers.
```

(continues on next page)

(continued from previous page)

```
1.7.16 NEW: faster InChI c interface added
       IMPROVED: bond order assignment speed (dimension of ILP reduced)
       OTHER: resonance bond order/formal charge generation disabled

1.7.15 FIXED: Make dms-select use desres-cleanenv.

1.7.14 NEW: msys.Save() function understands filename extensions.
       FIXED: Better hybridization detection.
       FIXED: SaveMAE preserves atom order (broken since 1.7.0).
       C++: load.hxx was renamed to io.hxx.

1.7.13 NEW: Python bindings for {Period,Group}ForElement.
       NEW: Python bindings for Calc{Distance,Angle,Dihedral,Planarity}.

1.7.12 NEW: The GlobalCell class is gone; System.cell is now represented
            a 3x3 NumPy array.  getCell() still returns a copy of the cell
            as before.  Small API change but no change in behavior of the
            existing interfaces.
       NEW: The Vec3 class was removed from the Python interface; it was there
             only to support the GlobalCell class.

1.7.11 NEW: AbbreviationForElement in Python interface.
       FIXED: ct properties were being improperly shared.
       FIXED: mol2 and sdf import/export better supports multiple entries.
       FIXED: dms-dump looks for more than six cmap tables.

1.7.10 FIXED: Atom selections with names beginning with digits were broken
              since 1.7.7.

1.7.9  FIXED: Amber prm files with nontrivial dihedral phases could be misread.

1.7.8  FIXED: Amber prm file should still add bonds when structure_only=True.
       FIXED: Amber prm file was sometimes incorrectly rejected.
       FIXED: GuessAtomicNumber for python.

1.7.7  NEW: New atom selection parser based on flex/yacc.
       NEW: smarts atom selection keyword.
       NEW: ParamTable.addParam() takes keyword arguments.
       FIXED: msys.AssignBondOrderAndFormalCharges(mol) is now much faster.
       FIXED: dms-validate has more useful options.
       FIXED: dms-alchemical does a sanity check.

1.7.6  FIXED: msys.knots was broken, which broke dms-validate.
       FIXED: dms-validate didn't properly handle systems with pseudos.
       FIXED: Read occupancy and bfactor from PDB files.
       FIXED: Bring back msys.GetSSSR.

1.7.5  FIXED: SaveMAE puts quotes around strings containing braces.
       FIXED: Catch misuse of System.chain(<string>), etc.

1.7.3  FIXED: maedms didn't apply LJ scale to pair terms with vdw overrides.

1.7.2  NEW:   System.clone() accepts a list of atom ids.
       NEW:   Recipes section in the documentation.
       FIXED: Better bond order assigner.
       FIXED: Read proper_harm in MAE files.
```

(continues on next page)

(continued from previous page)

```
1.7.1  NEW: dms-select learned --coalesce.
       NEW: inchi.InChI function.
       FIXED: SDF reader handles formal charge and some keyvals
       FIXED: bond order assigner gives up instead of hanging
       FIXED: Improved aromaticity and hybridization detection.

1.7.0  NEW: Ct class for multiple components and system attributes.
       NEW: ctnumber atom selection keyword (matches VMD's).
       NEW: dms_version in the DMS file.
       FIXED: Load[Many] failed on some compressed MAE files.
       DEPRECATED: all glue functionality (modern periodicfix doesn't need it).

1.6.35 NEW: Class AnnotatedSystem to cache rings and aromaticity
       FIXED: Aromaticity detection now operates on entire ring systems
       REMOVED: SSSR/ring aromaticity global functions in Python interface
       REMOVED: SSSR cache and AllRelevantSSSR function in System

1.6.33 NEW: NBFix support in ExportMAE.
       NEW: ExportMAEMany.  msys.SaveMAE accepts list of Systems.
       NEW: msys_provenance and msys_forcefield blocks in exported MAE systems.
       NEW: msys.LoadMany works with all filetypes.
       NEW: msys.Graph.matchAll can do substructure searches.

1.6.32 NEW: msys.LoadMany and C++ LoadIterator interface for working with
            multi-structure files.
       NEW: msys.Atomsel class with RMSD calculation and superposition.
       FIXED: dms-set was broken.

1.6.31 FIXED: Read m_grow_name in ImportMAE.

1.6.30 FIXED: dms-frame was broken.
       FIXED: protein backbone must have minimally correct bond topology.

1.6.28 NEW: dms-sequence tool and msys.sequence.Sequences() function.
       NEW: dms-reorder-atoms
       FIXED: dms-scale-vdw now requires --scale options.
       FIXED: Bond guesser keeps only the shortest bond to hydrogens.
       FIXED: Make PDB output more conformant.
       FIXED: ExportMAE no longer writes 0-length chain names.

1.6.25 REMOVED: msys.glue module and dms-glue program.
       FIXED: No more MSYS_CLEANENV.

1.6.24 REMOVED: mview is now in its own mview module.

1.6.23 NEW: inchi.Strings python interface.
       FIXED: mae and dms read and write insertion code.

1.6.21 NEW: Print count of override pairs in dms-info.
       NEW: mview tool for viewing dms file contents.
       NEW: Added es_funct field to nonbonded_info.

1.6.20 NEW: Enable support for hybridization matching in SMARTS matcher.
       FIXED: Deleted printf chatter.

1.6.19 NEW: Load() has a structure_only option.
```

(continues on next page)

(continued from previous page)

```
1.6.18 FIXED: SmartsMatcher for systems with pseudos.
       FIXED: TermTable find methods don't return duplicate terms when
              the terms contain duplicate atoms.

1.6.17 NEW: class SmartsPattern.
       NEW: MassForElement() function.
       FIXED: dms-dump handles systems with missing expected tables.

1.6.15 NEW: Added 'insertion' as a residue property.
       NEW: PDB reader reads 'altloc' as extra atom property.
       NEW: Added System.atomsGroupedBy(prop) convenience method.
       NEW: dms-posre learned --reference-structure and --reference-selection
       C++: aromatic.hxx is visible.

1.6.14 NEW: Using gcc/4.7.2, Python/2.7.3, and numpy/1.6.2.

1.6.13 FIXED: dms-alchemical handles hoh constraints in the atom map.

1.6.12 FIXED: TermTable.findOnly() makes its input sorted and unique.
       NEW: Param.keys() and Term.keys()

1.6.11 NEW: SDF export.
       NEW: RadiusForElement() function.
       NEW: FindDistinctFragments() function.
       NEW: dms-inchi tool.
       FIXED: LoadMAE handles forcefield provenance better.

1.6.10 FIXED: CreateSystem() is thread-safe again.

1.6.9  FIXED: System.atoms is now (amortized) constant time.
       NEW: dms-select now reads any msys-supported file type, not just dms.
       NEW: mae2dms supports fbhw terms.

1.6.8  FIXED: Bug in sidechain detection for residues with non-unique names.

1.6.7  FIXED: FindKnot uses within selections instead of boxing.
       FIXED: Restored the max-cycle option to FindKnot.

1.6.6  NEW: 'sidechain' atom selection.
       FIXED: bad int and float literals in atom selections are caught.
       FIXED: FindKnot uses accurate ring finder from GetSSSR.
       REMOVED: stretch_quartic schema.

1.6.5  NEW:   stretch_quartic schema.

1.6.4  FIXED: dms-alchemical checks for valid atom maps.
       FIXED: dms-alchemical avoid writing unnecessary alchemical terms.
       NEW: TermTable.findWithOnly.

1.6.3  FIXED: SavePDB writes element determined from atomic number.

1.6.2  NEW: Heuristic for guessing bonds.  From Python: System.guessBonds()
       NEW: xyz importer.
       NEW: xyz and pdb importers guess bonds.
       NEW: pdb importer reads and writes unit cell.

1.6.1  NEW: mol2 import/export support.
```

(continues on next page)

(continued from previous page)

```
       NEW: AssignBondOrderAndFormalCharge function (was in Viparr)
       NEW: GetSSSR (smallest set of smallest rings) (was in Viparr)

1.6.0  NEW: The Python atom selection handles large selections much more
            efficiently.
       NEW: Added System.selectIds(), which is more efficient than
            System.select() if all you need are the ids.
       NEW: C++ interface for working with override tables.
       NEW: Experimental support for multiple nonbonded tables.  When present,
            a new metatable in the DMS file will be created.

1.5.14 NEW: protein, backbone, and water analysis is done on load instead of
            being repeated for each selection.  This is faster and allows
            these selections to work on cloned subsets of the original system.
       FIXED: dms-uncharge works again now.
       FIXED: examples/alchemical_pair.py (for Schrodinger).
       FIXED: Python findWithAny, findWithAll, findExact for empty input.
       FIXED: __eq__ and __ne__ methods of System and ParamTable.
       FIXED: dms-dump raises an exception on sqlite errors.

1.5.13 NEW: Psfgen.read() and Psfgen.write() handle unit cell.
       FIXED: findWith{Any,All} on empty tables.

1.5.12 NEW: Psfgen.applyResidue does point mutations.
       NEW: C++ interface for filtered bonds.
       NEW: TermTable::resetParams(); in Python TermTable.params is settable.
       FIXED: deleting terms is faster now.
       FIXED: Psfgen.read() works now.
       FIXED: TermTable::hasTerm().
       FIXED: Destroyed TermTables appear empty.
       FIXED: Document dms-posre better.

1.5.11 NEW: Slightly optimized within selections.
       FIXED: dms-alchemical for noncontiguous alchemical ranges.

1.5.8  NEW: dms-fix-mass for fixing inconsitent masses.
       NEW: dms-find-knot for finding bonds passing through rings.
       NEW: dms-validate runs the knot check.
       FIXED: dms-alchemical handles alchemical_improper_harm .

1.5.7  FIXED: dms-alchemical was confused by extra properties in the B state.

1.5.6  FIXED: Memory leak in setPositions and setVelocities.

1.5.5  NEW: {get,set}Positions is ~100x faster due to NumPy integration.
       NEW: {get,set}Velocities is ~500x faster due to NumPy integration.
       NEW: System.center and Residue.center use NumPy.
       NEW: Low-level SystemPtr positions and velocities accessors take
            optional ids argument to get/set values for specified atoms.
       NEW: C++ interface for TermTable provides iterator for efficiency.

1.5.4  FIXED: SaveMAE reports error on failure.
       NEW: SaveMAE and SaveDMS print errno information on failure.

1.5.3  FIXED: Support SC{EE,NB}_SCALE_FACTOR in Amber PrmTop importer.

1.5.2  FIXED: System.setCell() converts its input to the correct type.
```

(continues on next page)

(continued from previous page)

```
1.5.1  FIXED: dms-frame reads in double precision if possible.
       FIXED: Better error messages when accessing nonexistent Term properties.
       NEW: Added getPositions(), getVelocities(), and getCell() to System.

1.5.0  REMOVED: dms-neutralize, dms-solvate
       REMOVED: Undocumented "builder" module.
       FIXED: Documentation for dms-override-vdw, dms-scale-vdw
       NEW: dms-frame tool for extracting coordinates and doing periodic fix.
       NEW: C++ interface for guessing file type.

1.4.8  NEW dms-frame script for frame extraction with centering and wrapping.

1.4.7  NEW: ParamTable::find and TermTable::find{WithAny,WithAll,Exact}.
       NEW: msys.SavePDB.
       NEW: better logging in dms-override-vdw and dms-scale-vdw.
       FIXED: reading of alchemical dms files.
       FIXED: LoadPDB now uses correct chain/segid logic.
       FIXED: dms-dump prints nonbonded_combined information.

1.4.6  Portability (build without C++0x, and on OS X).

1.4.5  Renamed dms-scale-vdw --scale option to --scale-sigma and added
       --scale-epsilon.

1.4.4  Add minimal build configuration options.
       DMS import no longer panics on empty particle table.

1.4.3  Added dms-scale-vdw.

1.4.2  Make ImportDMS thread-safe.

1.4.1  Another way to handle nonbonded_combined_param tables.  We now
       store the 2d table directly, so that we don't choke on systems
       whose entire vdw table has overrides.  API changes at the Python
       and C++ level; dms-override-vdw is unchanged.

1.4.0  The nonbonded_combined_param is now loaded into a TermTable
       called nonbonded_combined which has a new category called override.
       No change to the dms representation, but code that uses msys must
       use the new API in override.hxx to handle vdw overrides.

       The clone() method now copies provenance.

       Reimplemented dms-combine-vdw correctly and renamed it to
       dms-override-vdw.

1.3.7  Hide functions that can conflict with other shared libraries.

1.3.6  Added executable path to provenance.

1.3.5  Internal changes to facilitate inclusion in larger builds.

1.3.4  DEPRECATED: dms-neutralize, dms-solvate.
       Fixed typo in dms-neutralize.
       Fixed equality checks between msys handles of different type.
       Fixed hashing of Term and Param handles.
```

(continues on next page)

(continued from previous page)

```
       Fixed dms-dump nonbonded output for nonbonded param.
       mae2dms and dms2mae now try to preserve ffio_vdwtype names.

1.3.3  Added dms-combine-vdw.

1.3.2  Using 73-Desmond modules.

1.3.1  Fix ImportMAE handling of virtuals (broken in 1.3.0).
       Turn off type checking of particle and bond tables.
       Fix GuessAtomicNumber (affects ReadParmTop only).

1.3.0  Alchemical terms now appear in their own TermTable, and the paramB
       interfaces have been removed.

       Alchemical particles are now listed in an "alchemical_nonbonded" table;
       the chargeB, moiety, and alchemical properties have been removed.

       Alchemical particles can have arbitrary term properties, including
       "chargeC".

       Msys can now detect sharing of parameters by TermTables that share
       a ParamTable, not just within a single TermTable.

1.2.5  Export/import formal_charge to/from dms.

1.2.4  Avoid exporting std::vector of primitive types to python.
       Prmtop import bugfix for when torsions contain negative force constants

1.2.3  Export version info to python.

1.2.2  Module file now adds the -lmsys to DESRES_MODULE_LDLIBS

1.2.1  Much improved support for glue, including in clone() and append()
       operations.  The glue table has been renamed to msys_glue, though
       msys will still read from both tables and merge the two.

       dms-thermalize: the random seed is now 1 by default so that results
       are reproducible.

1.2.0  segid is now a property of Chain.
       New tools: dms-set and dms-macro

1.1.1  dms-neutralize: fix for positive solute charge (DESRESCode#1352)
       dms-neutralize: remove ions if needed to achieve specified concentration
       Fix bug in atom selections on systems with deleted atoms

1.1.0  Added user-defined atom selection macros.
       New atom selection documentation.
       Bumped the minor version since selection macros are visible in the
       dms file.

1.0.13 Disambiguate residues by segid (DESRESCode#1345,1346)

1.0.12 dms-builder fixes

1.0.11 Fixed a memory corruption bug in the dms reading code.
```

(continues on next page)

(continued from previous page)

```
1.0.10 neutralize: set the residue name of ions equal to the atom name.
       builder: preliminary support for charmm36.
       builder: guess atom type based on name if not provided in top file.

1.0.9  msys.Load() supports .maeff and .maeff.gz now.
       Documentation for DMS files converted from Desmond documentation.
       dms-validate --desmond to check for bonds between nonbonded atoms.

1.0.8  gzipped dms files are detected and handled automatically.
       Added msys.Load() for guessing file type from filename.
       dms-info works now on dms, mae, and prmtop files.
       Documentation for dms-validate.

1.0.7  Support for importing Amber prmtop and crd files.
       Update fastjson dependency.
       Documentation fixes.

1.0.6  Anton version: aligning dependencies with treetop/43-Anton.
       Blame scarpazz.

1.0.5  System.positions property.

1.0.4  Typo in dms-validate.

1.0.3  Bugfix: clone() and append() did not copy auxiliary tables (e.g. cmap).

1.0.2  Internal: fix symbol conflicts with other packages.

1.0.1  Documentation updates.
       Added dms-posre.

1.0.0  First stable release.
```

BIBLIOGRAPHY

[Bro-2004] C. L. Brooks III, A. D. MacKerell Jr., M. Feig, “Extending the treatment of backbone energetics in protein force fields: limitations of gas-phase quantum mechanics in reproducing protein conformational distributions in molecular dynamics simulations”, *J. Comput. Chem.*, 25:1400–1415, 2004.

[Gun-1984] W. F. van Gunsteren H. J. C Berendsen, “Molecular dynamics simulations: Techniques and approaches”, In A. J. Barnes et al., editor, *Molecular Liquids: Dynamics and Interactions*, NATO ASI C 135, pages 475–500. Reidel Dordrecht, The Netherlands, 1984.

PYTHON MODULE INDEX

m

`msys`, 17
`msys.atomsel`, 39
`msys.grease`, 58
`msys.knot`, 59
`msys.molfile`, 46
`msys.pfx`, 43
`msys.posre`, 59
`msys.sequence`, 57
`msys.thermalize`, 58
`msys.update`, 58

INDEX

Symbols

`__contains__()` (*msys.Atom method*), 18
`__contains__()` (*msys.Bond method*), 19
`__delitem__()` (*msys.Ct method*), 22
`__eq__()` (*msys.ParamTable method*), 27
`__eq__()` (*msys.System method*), 31
`__eq__()` (*msys.TermTable method*), 37
`__getinitargs__()` (*msys.System method*), 31
`__getitem__()` (*msys.Atom method*), 18
`__getitem__()` (*msys.Bond method*), 19
`__getitem__()` (*msys.Ct method*), 22
`__getitem__()` (*msys.IndexedFileLoader method*), 25
`__getitem__()` (*msys.Param method*), 27
`__getitem__()` (*msys.Term method*), 37
`__hash__()` (*msys.ParamTable method*), 27
`__hash__()` (*msys.System method*), 31
`__hash__()` (*msys.TermTable method*), 37
`__init__()` (*msys.AnnotatedSystem method*), 17
`__init__()` (*msys.Graph method*), 23
`__init__()` (*msys.HydrogenBondFinder method*), 24
`__init__()` (*msys.InChI method*), 25
`__init__()` (*msys.IndexedFileLoader method*), 25
`__init__()` (*msys.ParamTable method*), 27
`__init__()` (*msys.SmartsPattern method*), 30
`__init__()` (*msys.SpatialHash method*), 30
`__init__()` (*msys.System method*), 31
`__init__()` (*msys.SystemImporter method*), 36
`__init__()` (*msys.TermTable method*), 38
`__init__()` (*msys.atomsel.Atomsel method*), 39
`__len__()` (*msys.IndexedFileLoader method*), 25
`__len__()` (*msys.atomsel.Atomsel method*), 39
`__lt__()` (*msys.Atom method*), 18
`__lt__()` (*msys.Bond method*), 20
`__ne__()` (*msys.ParamTable method*), 27
`__ne__()` (*msys.System method*), 31
`__ne__()` (*msys.TermTable method*), 38
`__repr__()` (*msys.AnnotatedSystem method*), 17
`__repr__()` (*msys.Atom method*), 18
`__repr__()` (*msys.Bond method*), 20
`__repr__()` (*msys.Chain method*), 20
`__repr__()` (*msys.Param method*), 27
`__repr__()` (*msys.Residue method*), 28
`__repr__()` (*msys.SmartsPattern method*), 30
`__repr__()` (*msys.System method*), 31
`__repr__()` (*msys.Term method*), 37
`__repr__()` (*msys.TermTable method*), 38
`__repr__()` (*msys.atomsel.Atomsel method*), 39
`__setitem__()` (*msys.Atom method*), 19
`__setitem__()` (*msys.Bond method*), 20
`__setitem__()` (*msys.Ct method*), 22
`__setitem__()` (*msys.Param method*), 27
`__setitem__()` (*msys.Term method*), 37
`__str__()` (*msys.InChI method*), 25
`__str__()` (*msys.atomsel.Atomsel method*), 39
`__weakref__` (*msys.AnnotatedSystem attribute*), 17
`__weakref__` (*msys.BrokenBondsError attribute*), 20
`__weakref__` (*msys.Graph attribute*), 24
`__weakref__` (*msys.HydrogenBondFinder attribute*), 24
`__weakref__` (*msys.InChI attribute*), 25
`__weakref__` (*msys.IndexedFileLoader attribute*), 25
`__weakref__` (*msys.SmartsPattern attribute*), 30
`__weakref__` (*msys.SpatialHash attribute*), 30
`__weakref__` (*msys.SystemImporter attribute*), 36

A

`addAtom()` (*msys.Residue method*), 28
`addAtom()` (*msys.System method*), 31
`addAtom()` (*msys.SystemImporter method*), 36
`addAtomProp()` (*msys.System method*), 31
`addAuxTable()` (*msys.System method*), 32
`addBond()` (*msys.Atom method*), 19
`addbond()` (*msys.molfile.Atom method*), 48
`addBondProp()` (*msys.System method*), 32
`addChain()` (*msys.Ct method*), 22
`addChain()` (*msys.System method*), 32
`addCt()` (*msys.System method*), 32
`addNonbondedFromSchema()` (*msys.System method*), 32
`addParam()` (*msys.ParamTable method*), 27
`addProp()` (*msys.ParamTable method*), 28
`addResidue()` (*msys.Chain method*), 20
`addResidue()` (*msys.System method*), 32

`addTable()` (*msys.System method*), 32
`addTableFromSchema()` (*msys.System method*), 32
`addTerm()` (*msys.TermTable method*), 38, 41
`addTermProp()` (*msys.TermTable method*), 38, 41
`alignCoordinates()` (*msys.atomsel.Atomsel method*), 39
`aligned_rmsd()` (*in module msys.pfx*), 43
`alignedRMSD()` (*msys.atomsel.Atomsel method*), 39
`altloc` (*msys.molfile.Atom attribute*), 48
`analyze()` (*msys.System method*), 32
`AnnotatedSystem` (*class in msys*), 17, 51
`anum` (*msys.molfile.Atom attribute*), 48
`append()` (*msys.Ct method*), 22
`append()` (*msys.System method*), 32
`apply()` (*in module msys.posre*), 59
`apply()` (*in module msys.thermalize*), 58
`ApplyDihedralGeometry()` (*in module msys*), 18
`aromatic()` (*msys.AnnotatedSystem method*), 18, 52
`aromatic()` (*msys.Atom property*), 19
`asCapsule()` (*msys.System method*), 32
`AssignBondOrderAndFormalCharge()` (*in module msys*), 18
`at_time_ge()` (*msys.molfile.Reader method*), 50
`at_time_gt()` (*msys.molfile.Reader method*), 50
`at_time_le()` (*msys.molfile.Reader method*), 50
`at_time_lt()` (*msys.molfile.Reader method*), 50
`at_time_near()` (*msys.molfile.Reader method*), 50
`at_time_near()` (*msys.molfile.SeqFile.Reader method*), 49
`Atom` (*class in msys*), 18
`Atom` (*class in msys.molfile*), 48
`atom()` (*msys.System method*), 32
`atom_props()` (*msys.System property*), 32
`atomic_number()` (*msys.Atom property*), 19
`atomPropType()` (*msys.System method*), 32
`atoms()` (*msys.Bond property*), 20
`atoms()` (*msys.Ct property*), 22
`atoms()` (*msys.Graph method*), 24
`atoms()` (*msys.molfile.Reader property*), 50
`atoms()` (*msys.Residue property*), 28
`atoms()` (*msys.System property*), 32
`atoms()` (*msys.Term property*), 37
`Atomsel` (*class in msys.atomsel*), 39
`atomsel()` (*msys.System method*), 33
`atomsGroupedBy()` (*msys.System method*), 32
`auxinfo()` (*msys.InChI property*), 25
`auxtable()` (*msys.System method*), 33
`auxtable_names()` (*msys.System property*), 33
`auxtables()` (*msys.System property*), 33
`axis()` (*msys.molfile.Grid property*), 50

B

`bfactor` (*msys.molfile.Atom attribute*), 48
`Bond` (*class in msys*), 19
`bond()` (*msys.System method*), 33
`bond_props()` (*msys.System property*), 33
`bonded_atoms()` (*msys.Atom property*), 19
`bondorders()` (*msys.molfile.Reader property*), 50
`bondPropType()` (*msys.System method*), 33
`bonds` (*msys.molfile.Atom attribute*), 48
`bonds()` (*msys.Atom property*), 19
`bonds()` (*msys.Ct property*), 22
`bonds()` (*msys.System property*), 33
`box()` (*msys.molfile.Frame property*), 48
`BrokenBondsError`, 20

C

`CalcAngle()` (*in module msys*), 20
`CalcDihedral()` (*in module msys*), 20
`CalcDistance()` (*in module msys*), 20
`CalcPlanarity()` (*in module msys*), 20
`can_read()` (*msys.molfile.Plugin property*), 47
`can_write()` (*msys.molfile.Plugin property*), 47
`category()` (*msys.TermTable property*), 38, 41
`cell()` (*msys.System property*), 33
`center()` (*msys.Residue property*), 28
`center()` (*msys.System property*), 33
`Chain` (*class in msys*), 20
`chain` (*msys.molfile.Atom attribute*), 49
`chain()` (*msys.Residue property*), 28
`chain()` (*msys.System method*), 33
`chains()` (*msys.Ct property*), 22
`chains()` (*msys.System property*), 33
`charge` (*msys.molfile.Atom attribute*), 49
`charge()` (*msys.Atom property*), 19
`clone()` (*msys.System method*), 33
`CloneSystem()` (*in module msys*), 21
`close()` (*msys.molfile.Writer method*), 51
`coalesce()` (*msys.TermTable method*), 38, 41
`coalesceTables()` (*msys.System method*), 33
`ComputeTopologicalIds()` (*in module msys*), 21
`ConvertFromOEChem()` (*in module msys*), 21
`ConvertFromRdkit()` (*in module msys*), 21
`ConvertToOEChem()` (*in module msys*), 21
`ConvertToRdkit()` (*in module msys*), 21
`count_overrides()` (*msys.TermTable method*), 38, 41
`CreateParamTable()` (*in module msys*), 22
`CreateSystem()` (*in module msys*), 22
`Ct` (*class in msys*), 22
`ct()` (*msys.Chain property*), 20
`ct()` (*msys.System method*), 33
`cts()` (*msys.System property*), 33
`currentRMSD()` (*msys.atomsel.Atomsel method*), 39

D

`data()` (*msys.molfile.Grid property*), 50
`degree()` (*msys.AnnotatedSystem method*), 18, 52

`delAtomProp()` (*msys.System method*), 34
`delAtoms()` (*msys.System method*), 34
`delAuxTable()` (*msys.System method*), 34
`delbond()` (*msys.molfile.Atom method*), 49
`delBondProp()` (*msys.System method*), 34
`delBonds()` (*msys.System method*), 34
`delChains()` (*msys.System method*), 34
`delProp()` (*msys.ParamTable method*), 28
`delResidues()` (*msys.System method*), 34
`delTermProp()` (*msys.TermTable method*), 38, 41
`delTermsWithAtom()` (*msys.TermTable method*), 38, 41
`dpos()` (*msys.molfile.Frame property*), 48
`DtrReader` (*class in msys.molfile*), 47
`duplicate()` (*msys.Param method*), 27
`dvel()` (*msys.molfile.Frame property*), 48

E

`errors()` (*msys.AnnotatedSystem property*), 18, 52
`es_funct()` (*msys.NonbondedInfo property*), 56
`extended_energy()` (*msys.molfile.Frame property*), 48

F

`fileinfo()` (*msys.molfile.DtrReader method*), 47
`filename_extensions` (*msys.molfile.SeqFile attribute*), 49
`filename_extensions()` (*msys.molfile.Plugin property*), 47
`find()` (*msys.HydrogenBondFinder method*), 25
`find()` (*msys.ParamTable method*), 28
`findBond()` (*msys.Atom method*), 19
`findBond()` (*msys.System method*), 34
`findContactIds()` (*msys.System method*), 34
`findContacts()` (*msys.SpatialHash method*), 30
`FindDistinctFragments()` (*in module msys*), 22
`findExact()` (*msys.TermTable method*), 38, 41
`findMatches()` (*msys.SmartsPattern method*), 30, 53
`findNearest()` (*msys.SpatialHash method*), 31
`findWithAll()` (*msys.TermTable method*), 38, 41
`findWithAny()` (*msys.TermTable method*), 38, 41
`findWithin()` (*msys.SpatialHash method*), 31
`findWithOnly()` (*msys.TermTable method*), 38, 41
`first()` (*msys.Bond property*), 20
`formal_charge()` (*msys.Atom property*), 19
`FormatDMS()` (*in module msys*), 23
`FormatJson()` (*in module msys*), 23
`FormatSDF()` (*in module msys*), 23
`fpos()` (*msys.molfile.Frame property*), 48
`fragid()` (*msys.Atom property*), 19
`Frame` (*class in msys.molfile*), 48
`frame()` (*msys.molfile.DtrReader method*), 47
`frame()` (*msys.molfile.Reader method*), 50
`frame()` (*msys.molfile.SeqFile.Reader method*), 49
`frame()` (*msys.molfile.Writer method*), 51
`frames()` (*msys.molfile.Reader method*), 50
`frames()` (*msys.molfile.SeqFile.Reader method*), 49
`frameset_is_compact()` (*msys.molfile.DtrReader method*), 47
`frameset_path()` (*msys.molfile.DtrReader method*), 47
`frameset_size()` (*msys.molfile.DtrReader method*), 47
`fromCapsule()` (*msys.System class method*), 34
`FromSmilesString()` (*in module msys*), 23
`fromTimekeys()` (*msys.molfile.DtrReader static method*), 47
`fvel()` (*msys.molfile.Frame property*), 48

G

`get()` (*msys.Ct method*), 22
`get_prop()` (*msys.molfile.SeqFile.Reader method*), 49
`getbondorder()` (*msys.molfile.Atom method*), 49
`GetBondsAnglesDihedrals()` (*in module msys*), 23
`getCell()` (*msys.System method*), 34
`getOverride()` (*msys.TermTable method*), 38, 41
`getPositions()` (*msys.System method*), 34
`GetRingSystems()` (*in module msys*), 23
`GetSSSR()` (*in module msys*), 23
`getTable()` (*msys.System method*), 34
`getVelocities()` (*msys.System method*), 34
`Graph` (*class in msys*), 23
`Grease()` (*in module msys.grease*), 58
`Grid` (*class in msys.molfile*), 50
`grid()` (*msys.molfile.Reader method*), 50
`grid()` (*msys.molfile.Writer method*), 51
`grid_data()` (*msys.molfile.Reader method*), 50
`grid_meta()` (*msys.molfile.Reader method*), 50
`guessBonds()` (*msys.System method*), 34
`GuessHydrogenPositions()` (*in module msys*), 24

H

`has_velocities()` (*msys.molfile.Reader property*), 50
`hash()` (*msys.Graph method*), 24
`hash()` (*msys.System method*), 34
`hash_atoms()` (*msys.Graph static method*), 24
`hasTerm()` (*msys.TermTable method*), 38, 41
`hcount()` (*msys.AnnotatedSystem method*), 18, 52
`hybridization()` (*msys.AnnotatedSystem method*), 18, 52
`HydrogenBondFinder` (*class in msys*), 24

I

`id()` (*msys.Param property*), 27
`id()` (*msys.Term property*), 37
`ids()` (*msys.atomsel.Atomsel property*), 40

`InChI` (*class in msys*), 25
`index_ge()` (*msys.molfile.DtrReader method*), 47
`index_gt()` (*msys.molfile.DtrReader method*), 47
`index_le()` (*msys.molfile.DtrReader method*), 47
`index_lt()` (*msys.molfile.DtrReader method*), 47
`index_near()` (*msys.molfile.DtrReader method*), 47
`IndexedFileLoader` (*class in msys*), 25
`initialize()` (*msys.SystemImporter method*), 36
`insertion` (*msys.molfile.Atom attribute*), 49
`insertion()` (*msys.Residue property*), 28
`inverse_3x3()` (*in module msys.pfx*), 43

K

`key()` (*msys.InChI property*), 25
`keys()` (*msys.Ct method*), 22
`keys()` (*msys.Param method*), 27
`keys()` (*msys.Term method*), 37
`keyvals()` (*msys.molfile.DtrReader method*), 47
`kinetic_energy()` (*msys.molfile.Frame property*), 48

L

`LineIntersectsTriangle()` (*in module msys*), 25
`Load()` (*in module msys*), 26
`LoadDMS()` (*in module msys*), 26
`LoadMAE()` (*in module msys*), 26
`LoadMany()` (*in module msys*), 26
`LoadMol2()` (*in module msys*), 26
`LoadPDB()` (*in module msys*), 26
`LoadPrmTop()` (*in module msys*), 26
`LoadXYZ()` (*in module msys*), 26
`loneelectrons()` (*msys.AnnotatedSystem method*), 18, 52

M

`mass` (*msys.molfile.Atom attribute*), 49
`mass()` (*msys.Atom property*), 19
`match()` (*msys.Graph method*), 24
`match()` (*msys.SmartsPattern method*), 30, 53
`matchAll()` (*msys.Graph method*), 24
`MatchFragments()` (*in module msys*), 26
`message()` (*msys.InChI property*), 25
`metadata()` (*msys.molfile.DtrReader property*), 47
`module`
    `msys`, 17
    `msys.atomsel`, 39
    `msys.grease`, 58
    `msys.knot`, 59
    `msys.molfile`, 46
    `msys.pfx`, 43
    `msys.posre`, 59
    `msys.sequence`, 57
    `msys.thermalize`, 58
    `msys.update`, 58
`moveby()` (*msys.molfile.Frame method*), 48
`msys`
    `module`, 17
`msys.atomsel`
    `module`, 39
`msys.grease`
    `module`, 58
`msys.knot`
    `module`, 59
`msys.molfile`
    `module`, 46
`msys.pfx`
    `module`, 43
`msys.posre`
    `module`, 59
`msys.sequence`
    `module`, 57
`msys.thermalize`
    `module`, 58
`msys.update`
    `module`, 58

N

`name` (*msys.molfile.Atom attribute*), 49
`name` (*msys.molfile.SeqFile attribute*), 49
`name()` (*msys.Atom property*), 19
`name()` (*msys.Chain property*), 20
`name()` (*msys.Ct property*), 22
`name()` (*msys.molfile.Grid property*), 50
`name()` (*msys.molfile.Plugin property*), 47
`name()` (*msys.Residue property*), 28
`name()` (*msys.System property*), 34
`name()` (*msys.TermTable property*), 38, 41
`natoms()` (*msys.Ct property*), 22
`natoms()` (*msys.molfile.DtrReader property*), 47
`natoms()` (*msys.molfile.Reader property*), 50
`natoms()` (*msys.molfile.SeqFile.Reader property*), 49
`natoms()` (*msys.Residue property*), 28
`natoms()` (*msys.SmartsPattern property*), 30, 53
`natoms()` (*msys.System property*), 35
`natoms()` (*msys.TermTable property*), 38, 41
`nbonds()` (*msys.Atom property*), 19
`nbonds()` (*msys.System property*), 35
`nchains()` (*msys.Ct property*), 22
`nchains()` (*msys.System property*), 35
`ncts()` (*msys.System property*), 35
`next()` (*msys.molfile.Reader method*), 50
`nframes()` (*msys.molfile.DtrReader property*), 47
`nframes()` (*msys.molfile.Reader property*), 50
`nframes()` (*msys.molfile.SeqFile.Reader property*), 49
`nframesets()` (*msys.molfile.DtrReader property*), 48
`ngrids()` (*msys.molfile.Reader property*), 51
`nhydrogens()` (*msys.Atom property*), 19
`nonbonded_info()` (*msys.System property*), 35

`NonbondedInfo` (*class in msys*), 56
`NonbondedSchemas()` (*in module msys*), 27
`noverrides()` (*msys.TermTable property*), 38, 41
`nparams()` (*msys.ParamTable property*), 28
`nprops()` (*msys.ParamTable property*), 28
`nresidues()` (*msys.Chain property*), 20
`nresidues()` (*msys.System property*), 35
`nterms()` (*msys.TermTable property*), 38, 41

O

`occupancy` (*msys.molfile.Atom attribute*), 49
`ok()` (*msys.InChI property*), 25
`order()` (*msys.Bond property*), 20
`origin()` (*msys.molfile.Grid property*), 50
`other()` (*msys.Bond method*), 20
`override_params()` (*msys.TermTable property*), 38, 42
`overrides()` (*msys.TermTable method*), 39, 42

P

`Param` (*class in msys*), 27
`param()` (*msys.ParamTable method*), 28
`param()` (*msys.Term property*), 37
`params()` (*msys.ParamTable property*), 28
`params()` (*msys.TermTable property*), 39, 42
`ParamTable` (*class in msys*), 27
`ParseSDF()` (*in module msys*), 28
`path()` (*msys.IndexedFileLoader property*), 25
`path()` (*msys.molfile.DtrReader property*), 48
`pattern()` (*msys.SmartsPattern property*), 30, 53
`Plugin` (*class in msys.molfile*), 47
`pos()` (*msys.Atom property*), 19
`pos()` (*msys.molfile.Frame property*), 48
`position()` (*msys.molfile.Frame property*), 48
`positions()` (*msys.System property*), 35
`potential_energy()` (*msys.molfile.Frame property*), 48
`pressure()` (*msys.molfile.Frame property*), 48
`pressure_tensor()` (*msys.molfile.Frame property*), 48
`prettyname()` (*msys.molfile.Plugin property*), 47
`props()` (*msys.ParamTable property*), 28
`props()` (*msys.TermTable property*), 39, 42
`propType()` (*msys.ParamTable method*), 28
`provenance()` (*msys.System property*), 35

R

`radius` (*msys.molfile.Atom attribute*), 49
`raw_alignment()` (*msys.atomsel.Atomsel method*), 40
`read()` (*msys.molfile.Plugin method*), 47
`read()` (*msys.molfile.SeqFile class method*), 49
`ReadCrdCoordinates()` (*in module msys*), 28
`Reader` (*class in msys.molfile*), 50
`ReadPDBCoordinates()` (*in module msys*), 28
`reload()` (*msys.molfile.DtrReader method*), 48
`remove()` (*msys.Atom method*), 19
`remove()` (*msys.Bond method*), 20
`remove()` (*msys.Chain method*), 20
`remove()` (*msys.Ct method*), 22
`remove()` (*msys.Residue method*), 28
`remove()` (*msys.Term method*), 37
`remove()` (*msys.TermTable method*), 39, 42
`remove_drift()` (*in module msys.thermalize*), 58
`reopen()` (*msys.molfile.Reader method*), 51
`replaceWithSortedTerms()` (*msys.TermTable method*), 39, 42
`resid` (*msys.molfile.Atom attribute*), 49
`resid()` (*msys.Residue property*), 29
`Residue` (*class in msys*), 28
`residue()` (*msys.System method*), 35
`residues()` (*msys.Chain property*), 21
`residues()` (*msys.System property*), 35
`resname` (*msys.molfile.Atom attribute*), 49
`ringbondcount()` (*msys.AnnotatedSystem method*), 18, 52

S

`Save()` (*in module msys*), 29
`save()` (*msys.System method*), 35
`SaveDMS()` (*in module msys*), 29
`SaveMAE()` (*in module msys*), 29
`SaveMol2()` (*in module msys*), 29
`SavePDB()` (*in module msys*), 29
`second()` (*msys.Bond property*), 20
`segid` (*msys.molfile.Atom attribute*), 49
`segid()` (*msys.Chain property*), 21
`select()` (*msys.molfile.Frame method*), 48
`select()` (*msys.System method*), 35
`selectArr()` (*msys.System method*), 35
`selectAtom()` (*msys.Residue method*), 29
`selectChain()` (*msys.System method*), 35
`selectCt()` (*msys.System method*), 35
`selectIds()` (*msys.System method*), 35
`selectResidue()` (*msys.Chain method*), 21
`SeqFile` (*class in msys.molfile*), 49
`SeqFile.Reader` (*class in msys.molfile*), 49
`Sequences()` (*in module msys.sequence*), 57
`SerializeMAE()` (*in module msys*), 29
`setbondorder()` (*msys.molfile.Atom method*), 49
`setCell()` (*msys.System method*), 35
`setOverride()` (*msys.TermTable method*), 39, 42
`setPositions()` (*msys.System method*), 36
`setVelocities()` (*msys.System method*), 36
`size()` (*msys.Graph method*), 24
`skip()` (*msys.molfile.Reader method*), 51
`SmartsPattern` (*class in msys*), 29, 52
`sorted()` (*msys.System method*), 36

SpatialHash (*class in msys*), 30
string() (*msys.InChI property*), 25
svd_3x3() (*in module msys.pfx*), 43
sync() (*msys.molfile.Writer method*), 51
System (*class in msys*), 31
system() (*msys.atomsel.Atomsel property*), 40
system() (*msys.Param property*), 27
system() (*msys.SystemImporter property*), 36
system() (*msys.Term property*), 37
system() (*msys.TermTable property*), 39, 42
SystemImporter (*class in msys*), 36

T

table() (*msys.Param property*), 27
table() (*msys.System method*), 36
table() (*msys.Term property*), 37
table_names() (*msys.System property*), 36
tables() (*msys.System property*), 36
TableSchemas() (*in module msys*), 36
temperature() (*msys.molfile.Frame property*), 48
Term (*class in msys*), 36
term() (*msys.TermTable method*), 39, 42
term_props() (*msys.TermTable property*), 39, 42
termPropType() (*msys.TermTable method*), 39, 42
terms() (*msys.TermTable property*), 39, 42
TermTable (*class in msys*), 37, 40
time() (*msys.molfile.Frame property*), 48
times() (*msys.molfile.DtrReader method*), 48
times() (*msys.molfile.Reader property*), 51
topology() (*msys.molfile.Reader property*), 51
topology() (*msys.System property*), 36
total_bytes() (*msys.molfile.DtrReader method*), 48
total_energy() (*msys.molfile.Frame property*), 48
translate() (*msys.System method*), 36
truncate() (*msys.molfile.Writer method*), 51
type (*msys.molfile.Atom attribute*), 49

U

Update() (*in module msys.update*), 58
updateFragids() (*msys.System method*), 36

V

valence() (*msys.AnnotatedSystem method*), 18, 52
valence() (*msys.Atom property*), 19
vdw_funct() (*msys.NonbondedInfo property*), 56
vdw_rule() (*msys.NonbondedInfo property*), 56
vel() (*msys.Atom property*), 19
vel() (*msys.molfile.Frame property*), 48
velocity() (*msys.molfile.Frame property*), 48
version() (*msys.molfile.Plugin property*), 47
virial_tensor() (*msys.molfile.Frame property*), 48
voxelize() (*msys.SpatialHash method*), 31
vx() (*msys.Atom property*), 19
vy() (*msys.Atom property*), 19
vz() (*msys.Atom property*), 19

W

warnings() (*msys.SmartsPattern property*), 30, 53
write() (*msys.molfile.Plugin method*), 47
Writer (*class in msys.molfile*), 51

X

x() (*msys.Atom property*), 19

Y

y() (*msys.Atom property*), 19

Z

z() (*msys.Atom property*), 19