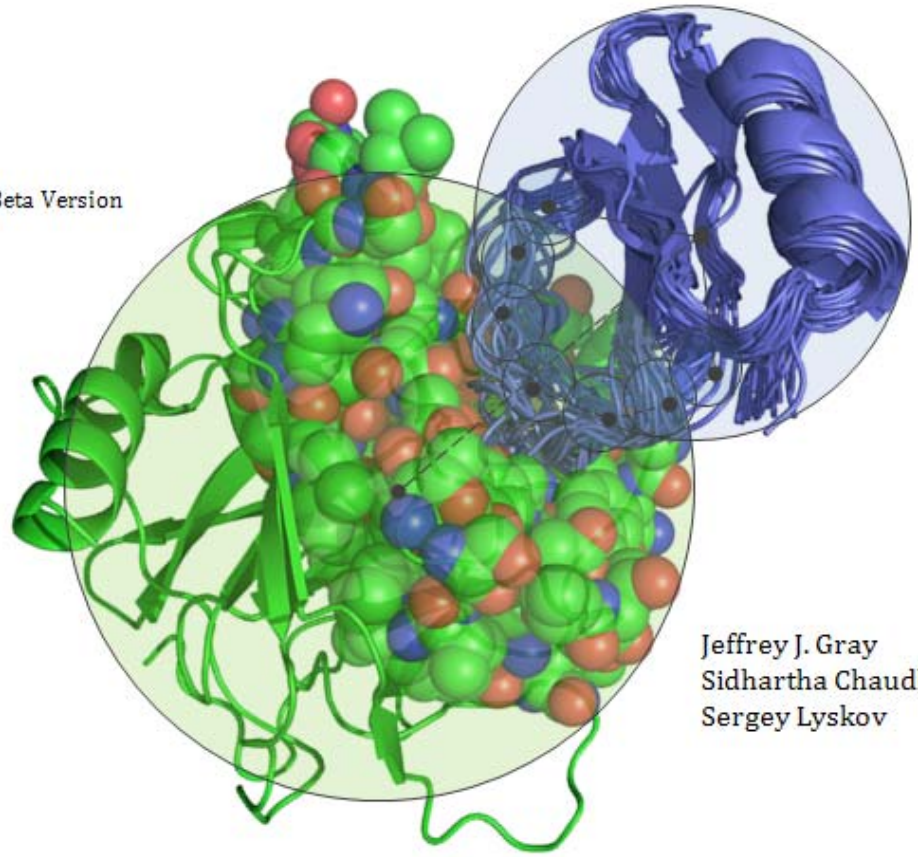
PyRosetta

Interactive Molecular Modeling for Proteins

User's Manual

Beta Version



Jeffrey J. Gray
Sidhartha Chaudhury
Sergey Lyskov

Johns Hopkins University, Baltimore, MD

Table of Contents

Preface

Structures of proteins and protein complexes help explain biomolecular function, and computational methods provide an inexpensive way to predict unknown structures, manipulate behavior, or design new proteins or functions. The protein structure prediction program Rosetta, developed by a consortium of laboratories in the Rosetta Commons, has an unmatched variety of functionalities and is one of the most accurate protein structure prediction and design approaches (Das & Baker Ann Rev Biochem 2008; Gray Curr Op Struct Biol 2006). To make the Rosetta approaches broadly accessible to biologists and biomolecular engineers with varied backgrounds, we developed PyRosetta, a Python-based interactive platform for accessing the objects and algorithms within the Rosetta protein structure prediction suite. In PyRosetta, users can measure and manipulate protein conformations, calculate energies in low- and high-resolution representations, fold proteins from sequence, model variable regions of proteins (loops), dock proteins or small molecules, and design protein sequences. Furthermore, with access to the primary Rosetta optimization objects, users can build custom protocols for operations tailored to particular biomolecular applications. Since the Python-based program can be run within the visualization software PyMol, search algorithms can be viewed on-screen in real time.

In this book, we go over the fundamentals of molecular modeling in PyRosetta, providing both a cursory background into the science behind various modeling strategies as well as instruction on how to use the PyRosetta objects and functions. Each unit covers a single topic in the field and walks the reader through the basic operations. Interactive exercises are incorporated so that the reader gains hands-on experience using the variety of commands available in the toolkit. The text is arranged progressively, beginning with the fundamentals of protein structure and energetics, and then progressing through the applications of protein folding, refinement, packing, design, docking, and loop modeling. A set of tables are provided at the end of the book as a reference of the available commands.

Additional resources on the Rosetta program are available online. The PyRosetta web site, pyrosetta.org, includes additional example and application scripts. A web-based user forum is in development and we hope that the PyRosetta community will share their experiences as well as useful scripts so that we build a repository of useful functions. For the expert, documentation on the underlying C++ code is available at www.rosettacommons.org under the TikiWiki application (www.rosettacommons.org/tiki/tiki-index.php). PyRosetta is built upon the Rosetta 3 platform, so objects available in PyRosetta will have the same underlying data structures and functionality.

Unit 1: Introduction to Molecular Modeling in Rosetta

1.1 Protein Structure and Function

Proteins are one of the five major biological macromolecules, they are responsible for a variety of biochemical processes from structure, to signaling, to catalyzing essential biochemical reactions. Proteins are polymers of amino acids that are encoded by genes. In the processes of *translation*, a mRNA transcript from the nucleus is used to create the protein chain in the ribosome. After translation, this amino acid polymer (also known as a polypeptide chain) adopts the lowest free-energy conformation in solution, through a process called ***protein folding***. Most naturally occurring proteins fold into a very specific shape or structure, and their function is directly a result of this structure. A typical folding energy for a protein is -10 kcal/mol, meaning that well over 99% of the protein in solution adopts this lowest-free energy conformation.

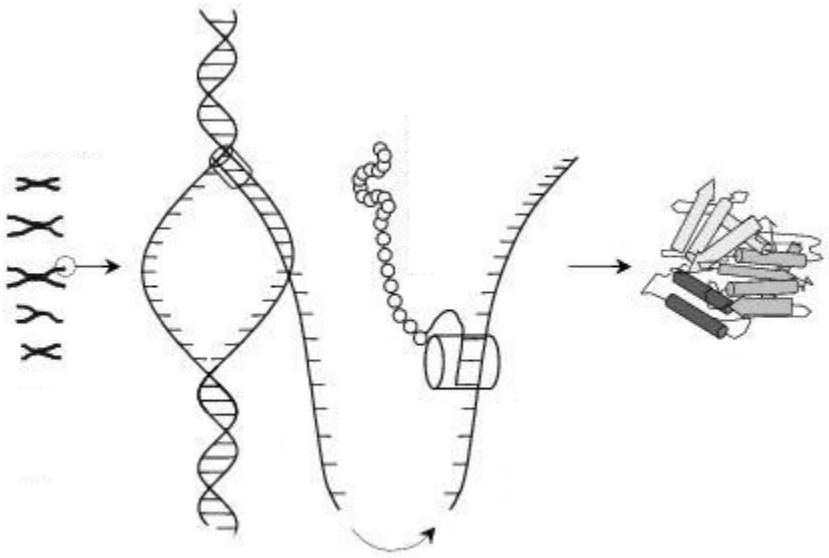


Fig 1.1.1 Protein synthesis

Like protein folding, almost all protein functions occur as a result of this basic thermodynamic principle: the protein will adopt the lowest energy conformation in a given environment. In almost all cases, this lowest energy conformation is a biologically-evolved, highly specific structure. For example, in protein-protein binding, in which two proteins are free in solution, the lowest energy conformation will be a highly specific complex between the two partners. The energy of a given conformation is a function of various molecular forces that act on that conformation, including electrostatics, Van der Waals interactions, hydrogen-bonding, and solvation energies.

Since its inception in 1998, the Rosetta molecular modeling software has been designed for accurate protein structure prediction and design. Fundamentally, its algorithms use this same

basic thermodynamic principle: it explores many conformations of a protein given a set of pre-defined constraints, under an approximate free energy function, searching for the lowest-energy, and consequently biologically-relevant, conformation. For each algorithm, from docking, to folding, to design, it is essential to understand three things:

1. What are conformational constraints to the system?
2. What is conformational sampling strategy within these predefined constraints?
3. What is the energy function/score function that is being used to identify the lowest-energy conformation?

1.2 Introduction to Protein Structure

The monomers of a polypeptide chain, called ***residues***, form a polymer through peptide bonds. The polypeptide chain itself (hereafter referred to as the ***backbone***) is made up entirely of an N-Cα bond, a Cα-C bond, and a C-N bond, repeated for each amino acid. The structure of this polymer can be described in two ways: first, as a set of Cartesian coordinates (***x, y, z***) that describe the position of every atom in the polymer, and second, as a ***internal coordinates***, as a set of torsion angle values along all bonds (**φ,ψ,ω**) for each amino acid in the polymer.

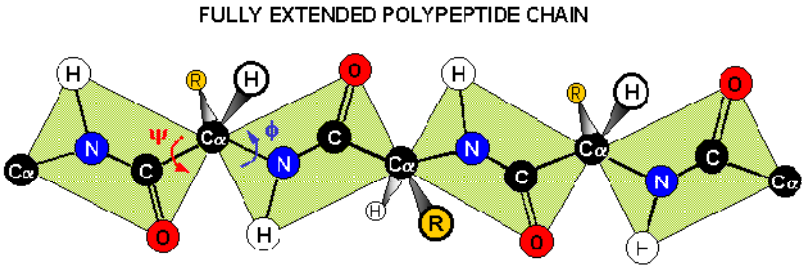


Fig 1.2.1 the polypeptide chain

The specificity of protein structure and function arises from its amino acid sequence. Although all backbone atoms for all residues are identical, each residue in the polypeptide chain is one of 20 amino acids, shown as R-groups in the Fig 1.2.1. Amino acids differ from one another through their ***side-chains***, defined as those non-backbone atoms bonded to the Cα atom. The twenty amino acids and their side-chains are shown in Fig 1.2.2.

The structure of the residue side-chains can be described either in the Cartesian coordinates (*x*,*y*,*z*) of each side-chain atom, or as the torsion angle values of each rotatable bond in the side-chain (χ_1, χ_2, χ_3…). Note that because different amino acids have different side-chains, they have a varying number of torsion angles, from no torsion angles in glycine, to four torsion angles in lysine.

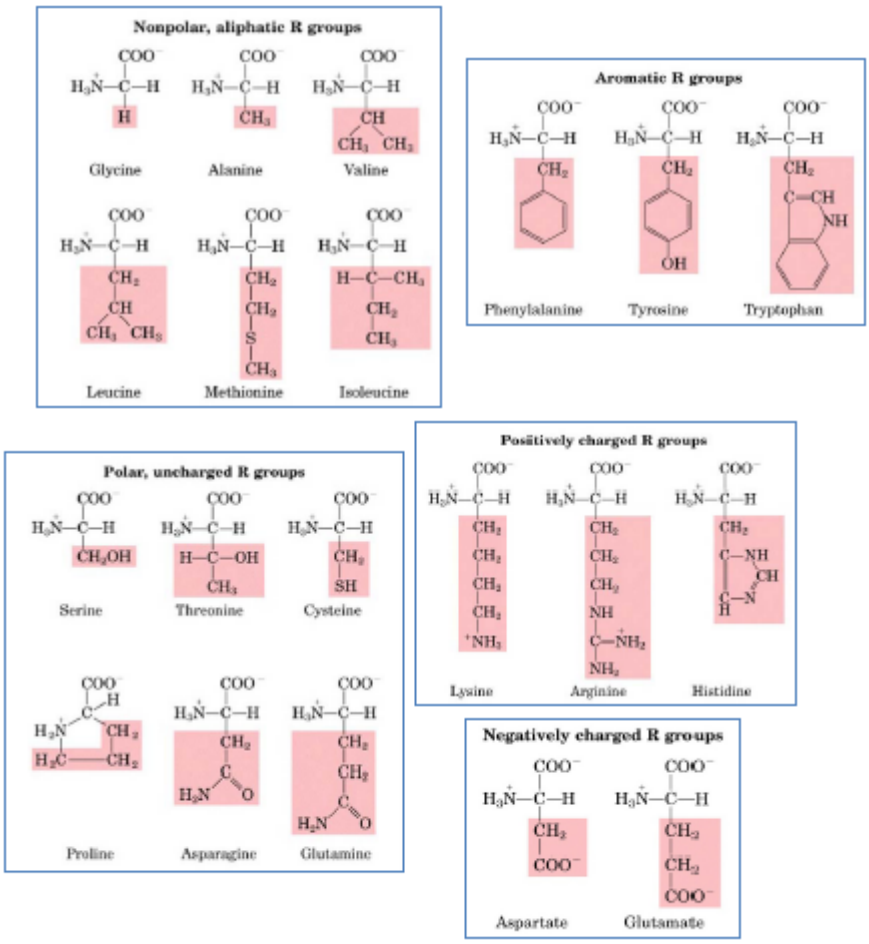


Fig 1.2.2 Amino Acid types

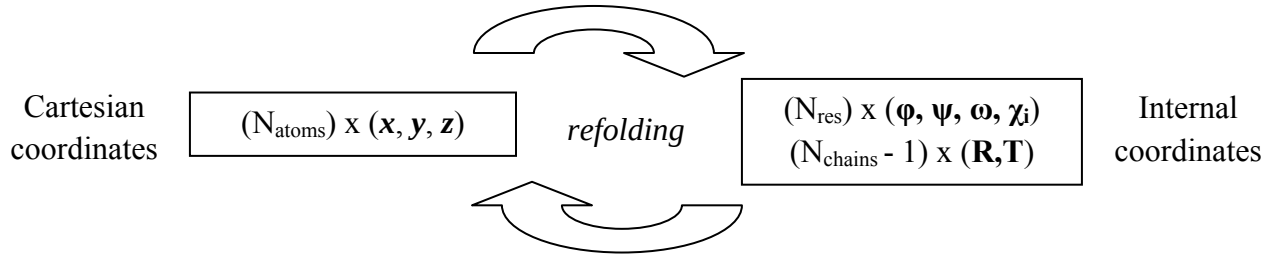
Finally, many protein systems in biology include more than a single protein chain. A protein complex, for example, by definition, involves the interaction of two proteins, or two protein chains. The relative spatial position and orientation of one polypeptide chain relative to another can be described exactly using three dimensions of translation (**T**) and three dimensions of rotation (**R**), based on a given fixed spatial reference frame.

1.3 Introduction to Molecular Energies

The energy of a given conformation is a result of molecular forces that act on and between all non-bonded atoms in that conformation. In protein biophysics, the most significant energy contributions that favor one conformation over another are Van der Waals energies, steric repulsions, electrostatic energies, solvation energies, and hydrogen bonding energies. These component energies themselves functions of the atomic positions and atom types of the atoms in the protein structure.

1.4 Protein Structure and Energies in Rosetta

In Rosetta, a protein system is represented by an object called a ***pose***. The pose contains all the structural information necessary to completely define the system in both spatial coordinates and internal coordinates.



Within the pose object, the Cartesian coordinates and internal coordinates are synchronized, both are used in Rosetta. Sampling is done primarily through manipulation of the internal coordinates (for example, perturbing a φ or ψ torsion angle), while the energy is calculated primarily from the Cartesian coordinates (for example, calculating the electrostatic interaction between two atoms, using their respective spatial coordinates). In a process called "refolding", new Cartesian coordinates are generated by 'rebuilding' the polypeptide chain, side-chain conformation, and rigid-body orientations, from the internal coordinates. Likewise, when a pose is first instantiated, it is typically created from a PDB file, which contains only spatial coordinates, and the internal coordinates are generated from the spatial coordinates automatically. At all times, the spatial coordinates and internal coordinates are synchronized within the pose.

In Rosetta, most of the energetic components are functions of the atom types and spatial positions of atoms that comprise the system. Van der Waals interactions and Pauli Exclusion forces are modeled using the Lennard-Jones potential, using the Cartesian coordinates of all non-bonded atoms. Electrostatics are modeled using a simple distance-dependent dialectric, with distance derived from the coordinates of charged atoms. Solvation is modeled using the Lazardis-Karplus Gaussian solvent-exclusion model using the pair-wise distances of all atoms in the system. There are scoring components that use the internal coordinates as well. Hydrogen-bonding is modeled using a backbone-dependent, orientation-dependent empirical formula that uses both Cartesian coordinates and bond geometries of donors and acceptors as well as the φ and ψ of the hydrogen-bonding residues. Finally, there are internal energy terms, such as the Dunbrack side-chain energy, or the Ramachandran backbone energy, that are derived purely from the internal coordinates (χ and φ,ψ, respectively) directly.

1.5 Getting started in PyRosetta

PyRosetta is written as a python front-end to the Rosetta molecular modeling suite. It allows the user to access Rosetta functions and classes using the Python scripting language. To get started you have to import all Rosetta functions and initialize the Rosetta database files:

```
>>from rosetta import *
>>init()
```

PyRosetta has primarily two modes of use: interactive mode and script mode. Interactive-mode uses IPython and allows the user to type in PyRosetta commands in real-time. It is excellent for learning to use PyRosetta or running simple commands quickly. Simply go to your PyRosetta directory and type:

```
>>python ipython.py
```

Once the interactive mode has loaded up, type in the previous two commands into the Python terminal to import the Rosetta library and initialize PyRosetta, and you are ready to go. Additionally, there are two help features in interactive mode: tab-completion and the 'help' command. Double-tapping the 'tab' button will show a list of possible functions that match what has been entered in the Python terminal so far. The help function will take in any PyRosetta object/function name and output a text description of what that function does as well as usage information, for example how it is constructed or called:

```
>>help(Pose)
<< Help on class Pose in module rosetta.core.pose._rosetta_core_pose:

class Pose(Boost.Python.instance)
 |  The Pose class represents a molecular system (proteindnaligand...)
 |  as a container of librosetta Residue objects together with
 |  an atom-tree structure that defines how internal coordinate changes
 |  propagate through the system and an Energies object that stores
 |  information from the last energy evaluation.
 |  The two main ingredients are a Conformation object and an Energies
 |  object.
 |  The main responsibilities of the pose are
 |  ii  Kinematic:
 |  (a) to update the xyz coordinates in response to changes to internal
 |  degrees of freedom, and
 |  (b) to update  internal coordinates when the user modifes the xyz
 |  (cartesian) coords,
 |  ii  Scoring:
 |  (a) to keep track of what parts of the structure have changed since
 |  the last score evaluation, and
 |  (b) to cache rsd and rsd-pair energies for efficient re-use
 |  ii As a container:
 |  The pose should provide a useful single object for passing
 |  a molecular system and support copying of entire molecules
```

```
|  or stretches of molecules from one Pose object into another.
|  thats way more comments than I have ever written in a single
|  stretch before...
|
|  Methods defined here:
|  __init__(...)
|      __init__( (object)arg1) -> None :
|          default constructor
|...
```

The second mode of PyRosetta is script-mode. This is the mode where most writing and development of PyRosetta protocols is done. Simply add the lines import the Rosetta library and initializing PyRosetta to the beginning of the Python script to access the Rosetta functions and objects. PyRosetta scripts can be easily viewed and edited through any major Text Editor, and shared, through the website at http://www.pyrosetta.org. Sample scripts can be downloaded from the 'PyRosetta Scripts' section of the website.

Unit 2: Protein structure in PyRosetta

The pose object contains all the necessary information to completely define a protein system, including the amino acid sequence, the Cartesian coordinates, and internal coordinates of all proteins in the system. The Protein Databank File (PDB) is the standardized file-format for storing the Cartesian-coordinates for each atom in a protein, and is used in PyRosetta to input or output a protein structure.

2.1 Exploring the Pose object

To load a PDB file you have to first create an empty pose object, and then assign a PDB structure to that pose:

```
>>my_pose = Pose()
>>pose_from_pdb( my_pose, “test.pdb” )
```

You can access a summary of the pose through:

```
>>print my_pose
PDB file name: test.pdb
Total residues:116
Sequence: DAITHSILDWIDNLESPLSEKVSERSGYSKWHLQRMFKKETGHSLGQYRSRK…
Fold tree:
FOLD_TREE EDGE 1 116 -1
```

This summary contains the pdb file name that the pose originated from, the total number of residues in the pose, the amino acid sequence, and the *fold tree*. The fold tree (described in Section 7.2) describes the path of refolding of the pose. This information can also be accessed individually:

```
>>print my_pose.total_residue()
116
>>print my_pose.sequence()
DAITHSILDWIDNLESPLSEKVSERSGYSKWHLQRMFKKETGHSLGQYRSRK…
>>print my_pose.fold_tree()
FOLD_TREE EDGE 1 116 -1
```

Each residue in a pose is represented as a *residue* object within the pose. It can be accessed as (residue #2 as an example) below. Printing the residue displays the residue type, residue number (in pose numbering), all the atoms that make up the residue and their respective Cartesian coordinates. Again, this information can be accessed individually as well.

```
>>print p.residue(2)
ALA 2:
 N  : 0.764 37.858 76.239
```

```
 CA : 0.778 39.078 77.028
 C  : -0.624 39.484 77.368
 O  : -0.925 39.776 78.527
 CB : 1.474 40.216 76.301
 H  : 1.417 37.762 75.474
 HA : 1.307 38.885 77.961
1HB : 1.461 41.108 76.927
2HB : 2.506 39.936 76.09
3HB : 0.955 40.422 75.367

>>print p.residue(2).name()
ALA
```

Residue chemical properties that have been defined in the parameter files (see Section 2.4) can also be accessed:

```
>>p.residue(2).is_polymer()
>>p.residue(2).is_protein()
>>p.residue(2).is_DNA()
>>p.residue(2).is_RNA()
>>p.residue(2).is_NA()
>>p.residue(2).is_ligand()
>>p.residue(2).is_polar()
>>p.residue(2).is_aromatic()
```

The residue numbering in a pose object is always consecutive, beginning at residue 1. This is in contrast to PDB files, in which each residue has both a residue number and chain letter. PyRosetta will read a PDB file and sequentially number the residues in the file as they are listed. The conversion between pose residue numbering and PDB residue number and chain letter can be access as:

```
>>print my_pose.pdb_info().pdb2pose('I',8)
239
>>print my_pose.pdb_info().pose2pdb(239)
8  I
```

2.2 Accessing and manipulation protein geometry through the pose

The pose object stores the protein structure as both Cartesian coordinates and internal coordinates. We can access the internal coordinates for each residue as follows:

```
>>res_num = 5
>>print my_pose.phi(res_num)
>>print my_pose.psi(res_num)
>>print my_pose.chi(chi_num, res_num)
```

In the same way, new internal coordinates can be assigned at each residue:

```
>>my_pose.set_phi(res_num, new_angle)
```

```
>>my_pose.set_psi(res_num, new_angle)
>>my_pose.set_chi(chi_num, res_num, new_angle)
```

The bond lengths and angles, almost always held fixed in Rosetta algorithms, can also be manipulated. Since this is a less-used function, it is not as easy to use and utilizes **`AtomID`** objects based on atom numbers and residue numbers, as listed in the PDB file:

```
>>N = AtomID(1,res_num)
>>CA = AtomID(2, res_num)
>>C = AtomID(3, res_num)
```

From these **`AtomID`** objects, the bond length and angles can be accessed and manipulated for the pose object:

```
>>print my_pose.conformation().bond_length(N, CA)
>>print my_pose.conformation().bond_angle(N, CA, C)
>>print my_pose.conformation().set_bond_length(N, CA, new_bond_length)
>>print my_pose.conformation().set_bond_angle(N, CA, C, new_bond_angle)
```

Finally, the atomic coordinates of each atom in the pose is accessible through the *residue* object. The atomic coordinates are stored in atom order, with the first four atoms as N, CA, C, O. The atom order and atom numbers can be accessed using the atom_index function:

```
>>print my_pose.residue(5).atom_index(CA)
2
>>print my_pose.residue(5).xyz(2)
0.010                         35.998      80.515
>>print my_pose.residue(5).xyz(my_pose.residue(5).atom_index(CA))
0.010                         35.998      80.515
```

The coordinates are returned as **`numeric.xyzVector`**'s and can be used with Python Math functions.

2.3 Centroid and Full-atom representations of protein structure

In Rosetta there are two main 'modes' of representation of protein structure, centroid-mode and full-atom mode. In full-atom mode, for each residue, all backbone and side-chain atoms are explicitly modeled. In centroid-mode, all backbone atoms are explicitly modeled, but the side-chain atoms are replaced with a single pseudo-atom, called a 'centroid'. The position of the centroid is based on the average center-of-mass of that side-chain among the low-energy side-chain conformations; the size (radius) of the centroid is related to the average size of that side-chain.

The advantage of centroid-mode is that it allows for much faster energy calculations because the number of atoms in the simulation falls by significant amount, allowing for rapid searches through large areas of conformational space during Monte Carlo simulations. The disadvantage is that interactions involving the side-chain, such as side-chain hydrogen bonds, or side-chain Van der Waals interactions, cannot be captured explicitly because doing so would require all atoms of the side-chain to be modeled. Instead these interactions are captured implicitly using pair-wise statistical potentials and VdW sphere approximations. These implicit measurements can be significantly less accurate, which is why most Rosetta protocols have a low-resolution or coarse-grain phase using centroid mode with the main responsibility of *sampling* a diverse range of conformations *quickly*, and a high-resolution refinement phase, with the main responsibility of assessing an *accurate* energy for a given structure.

The example below shows a full-atom and centroid-mode representation of a Tryptophan residue:

```
 N  : -7.438 37.691 87.434
 CA : -8.307 36.699 87.982
 C  : -7.701 36.101 89.223
 O  : -8.432 35.736 90.14
 CB : -8.62 35.581 87
 CG : -9.535 34.551 87.595
 CD1: -10.902 34.633 87.706
 CD2: -9.17 33.258 88.106
 NE1: -11.406 33.47 88.256
 CE2: -10.362 32.614 88.513
 CE3: -7.959 32.591 88.27
 CZ2: -10.372 31.333 89.063
 CZ3: -7.97 31.314 88.814
 CH2: -9.171 30.699 89.203
 H  : -7.073 37.611 86.496
 HA : -9.252 37.156 88.276
1HB : -9.113 35.983 86.114
2HB : -7.702 35.072 86.705
 HD1: -11.37 35.554 87.363
 HE1: -12.381 33.28 88.44
 HE3: -7 33.024 87.988
 HZ2: -11.326 30.893 89.354
 HZ3: -7.016 30.8 88.932
 HH2: -9.142 29.695 89.626
```

```
N  : -7.438 37.691 87.434
CA : -8.307 36.699 87.982
C  : -7.701 36.101 89.223
O  : -8.432 35.736 90.14
CB : -8.59316 35.6086 86.9479
CEN: -9.89581 34.8776 86.0296
H  : -7.073 37.611 86.496
```

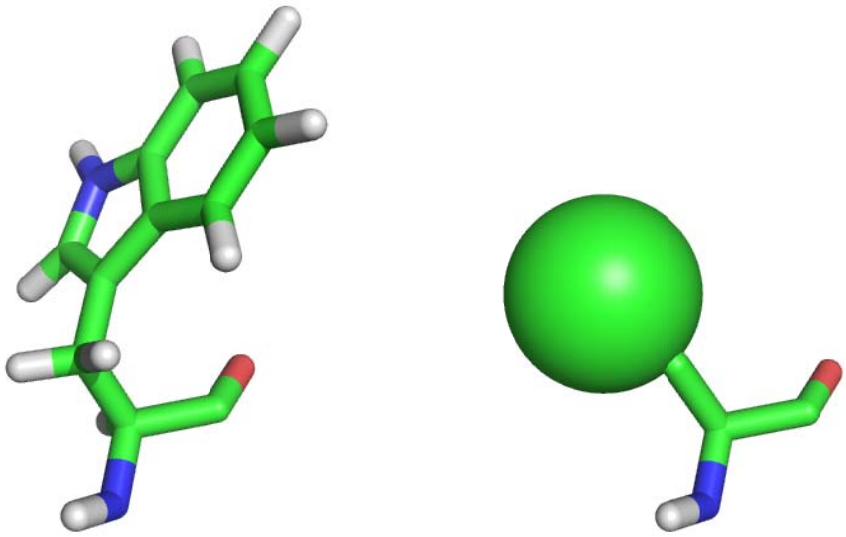


Figure 5.6. Full-atom (left) and centroid (right) representations for Tryptophan

Section 5.6 describes how to convert a protein between full-atom and centroid mode. A Pose can be queried as to whether it is in full-atom mode using:

```
>>pose.is_fullatom()
<<True
```

2.4 Parameter files for residues in PyRosetta

Parameter files describing the chemical and structural properties of each residue is found in the `minirosetta_database/chemical/residue_type_sets` directory in the PyRosetta directory.

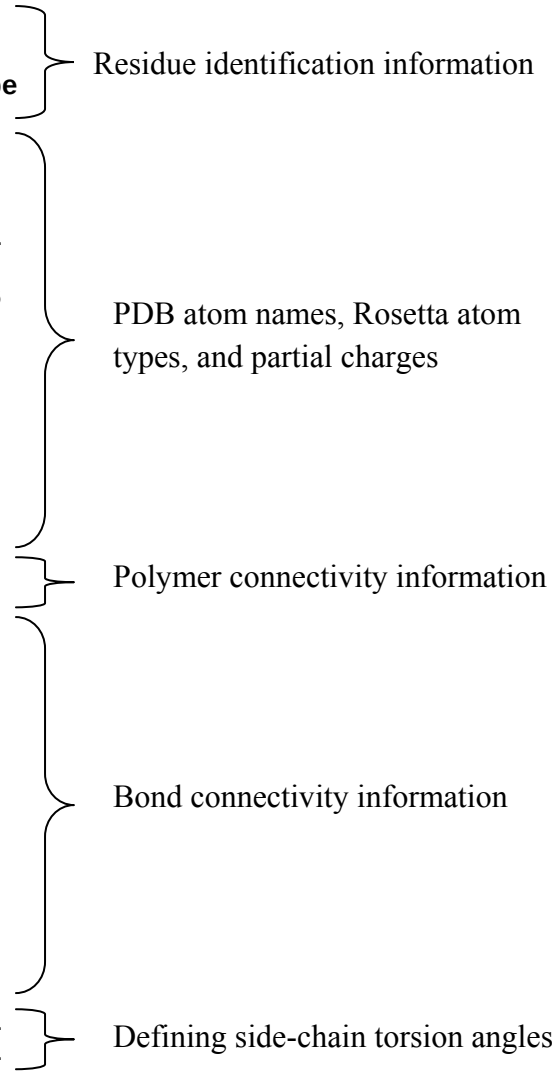
The full-atom residue parameters are stored in the `/fa_standard/residue_types` directory.

As an example, the parameter file for Threonine is shown below:

```
NAME THR
IO_STRING THR T
TYPE POLYMER #residue type
AA THR
ATOM  N   Nbb  NH1  -0.47
ATOM  CA  CAbb CT1  0.07
ATOM  C   CObb C    0.51
ATOM  O   OCbb O    -0.51
ATOM  CB  CH1  CT1  0.14
ATOM  OG1 OH   OH1  -0.66
ATOM  CG2 CH3  CT3  -0.27
ATOM  H   HNbb H    0.31
ATOM  HG1 Hpol H    0.43
ATOM  HA  Hapo HB   0.09
ATOM  HB  Hapo HA   0.09
ATOM 1HG2 Hapo HA   0.09
ATOM 2HG2 Hapo HA   0.09
ATOM 3HG2 Hapo HA   0.09
LOWER_CONNECT N
UPPER_CONNECT C
BOND  N    CA
BOND  N    H
BOND  CA   C
BOND  CA   CB
BOND  CA   HA
BOND  C    O
BOND  CB   OG1
BOND  CB   CG2
BOND  CB   HB
BOND  OG1  HG1
BOND  CG2 1HG2
BOND  CG2 2HG2
BOND  CG2 3HG2
CHI 1  N    CA   CB   OG1
CHI 2  CA   CB   OG1  HG1

PROTON_CHI 2 SAMPLES 3 60 -60 180 EXTRA 1 20

PROPERTIES PROTEIN POLAR
NBR_ATOM CB
NBR_RADIUS 3.4473
FIRST_SIDECHAIN_ATOM CB
ACT_COORD_ATOMS OG1 END
ICOOR_INTERNAL    N      0.000000    0.000000    0.000000   N     CA    C
ICOOR_INTERNAL    CA     0.000000  180.000000    1.458001   N     CA    C
ICOOR_INTERNAL    C      0.000000   68.800049    1.523257   CA    N     C
ICOOR_INTERNAL  UPPER  149.999954   63.800026    1.328685   C     CA    N
ICOOR_INTERNAL    O    180.000000   59.199905    1.231016   C     CA  UPPER
ICOOR_INTERNAL    CB  -121.983574   68.467087    1.539922   CA    N     C
ICOOR_INTERNAL    OG1   -0.000077   70.419235    1.433545   CB    CA    N
ICOOR_INTERNAL    HG1    0.000034   70.573135    0.960297   OG1   CB    CA
ICOOR_INTERNAL    CG2 -120.544136   69.469185    1.520992   CB    CA    OG1
ICOOR_INTERNAL   1HG2 -179.978256   70.557961    1.089826   CG2   CB    CA
ICOOR_INTERNAL   2HG2  120.032188   70.525108    1.089862   CG2   CB   1HG2
ICOOR_INTERNAL   3HG2  119.987984   70.541740    1.089241   CG2   CB   2HG2
ICOOR_INTERNAL    HB  -120.292923   71.020676    1.089822   CB    CA    CG2
```



```
ICOOR_INTERNAL    HA  -120.513664   70.221680    1.090258   CA    N     CB
ICOOR_INTERNAL  LOWER -149.999969   58.300030    1.328684   N     CA    C
ICOOR_INTERNAL    H    180.000000   60.849979    1.010000   N     CA  LOWER
```

Residue structure defined in internal coordinates

The centroid residue parameters can be found in `/centroid/residue_types` directory. As an example, the centroid parameter file for Threonine is shown below:

```
NAME THR
IO_STRING THR T
TYPE POLYMER #residue type
AA THR
ATOM  N   Nbb  NH1  -0.47
ATOM  CA  CAbb CT1  0.07
ATOM  C   CObb C    0.51
ATOM  O   OCbb O    -0.51
ATOM  CB  CB    CT1  0.14
ATOM  H   HNbb H    0.31
LOWER_CONNECT N
UPPER_CONNECT C
BOND  N    CA
BOND  N    H
BOND  CA   C
BOND  CA   CB
BOND  C    O
PROPERTIES PROTEIN POLAR
NBR_ATOM CEN
NBR_RADIUS 3.025
FIRST_SIDECHAIN_ATOM CB
ICOOR_INTERNAL    N      0.000000    0.000000    0.000000   N     CA    C
ICOOR_INTERNAL    CA     0.000000  180.000000    1.458001   N     CA    C
ICOOR_INTERNAL    C      0.000000   68.800049    1.523257   CA    N     C
ICOOR_INTERNAL  UPPER  149.999954   63.800026    1.328685   C     CA    N
ICOOR_INTERNAL    O    180.000000   59.199905    1.231016   C     CA  UPPER
ICOOR_INTERNAL    CB  -121.983574   68.467087    1.539922   CA    N     C
ICOOR_INTERNAL  LOWER -149.999969   58.300030    1.328684   N     CA    C
ICOOR_INTERNAL    H    180.000000   60.849979    1.010000   N     CA  LOWER
```

Residue identification information

PDB atom names, Rosetta atom types, and partial charges

Polymer connectivity information

Bond connectivity information

Residue properties

Defining parameters for neighbor calculations

Residue structure defined in internal coordinates

```
##centroid-specific
ATOM  CEN CEN_THR H 0.0
BOND CA CEN
ICOOR_INTERNAL    CEN -128.951279   72.516479    2.072556    CA     N     C
```

Centroid-specific information

Unit 3: Calculating energies in PyRosetta

The pose object gives us a framework for manipulating the protein structure, sampling different conformations. The PyRosetta score functions allow us to evaluate the approximate free energy of each conformation. In general form, the score function takes in a pose object and outputs a score that represents its energy.

3.1 Introduction to the score functions

There are over 30 different components to the score function, and each algorithm, from docking to loop modeling typically uses its own combination of components and component weights. A comprehensive table of all scoring components and their abbreviations can be found in the index. The 'standard' score function can be loaded up as follows:

```
>>name = 'standard'
>>my_scorefxn = create_score_function( name )
>>print my_scorefxn
ScoreFunction::show()
weights: (fa_atr 0.8) (fa_rep 0.44) (fa_sol 0.65) (fa_intra_rep 0.004)…
```

Printing the score function information for the standard score function lists, among other things, all non-zero weights and their respective components.

To calculate the energy of a given pose under the score function, simply pass the pose object into the score function:

```
>>my_scorefxn(my_pose)
465.88104573
```

There are primarily two forms of representing a protein in Rosetta: full-atom and centroid mode. Some scoring components require full-atom mode, others centroid mode, and others work for both modes. When scoring a Pose, the centroid/full-atom representation of that Pose needs to be compatible with *all* scoring components in the ScoreFunction.

The standard centroid-mode scoring function is:

```
>>scorefxn = create_score_function('cen_std')
>>print scorefxn
<< ScoreFunction::show():
weights: (vdw 1) (pair 1) (env 1) (cbeta 1)
```

A list of the available scoring function weight sets can be found in `/minirosetta_database/scoring/weights/` in the PyRosetta directory. They can be opened, viewed, and edited using any Text Editor and they simply list the components and their weights. There are two types files in this folder: weight sets and patches. Weight sets can be used as described above to create a `ScoreFunction` from scratch. Patches are applied to a weight set and modify or add additional components to that weight set. In the below example, the score12 weight patch (`score12.wts_patch`) is applied to the standard weight set (`standard.wts`):

```
>>scorefxn = create_score_function_ws_patch('standard', 'score12')
```

The patches allow for additional flexibility when setting up a score function. For example, the `score4L` patch adds additional components for scoring loops models, such as chain-break penalties.

3.3 Scoring components

The most common score function components are:

Rosetta Full-atom Scoring Functions		
Van der Waals net attractive energy	**FA**	**fa_atr**
Van der Waals net repulsive energy	**FA**	**fa_rep**
Hydrogen bonds, short and long-range, (backbone)	**FA/CEN**	**hbond_sr_bb, hbond_lr_bb**
Hydrogen bonds, short and long-range, (side-chain)	**FA**	**hbond_sc, hbond_bb_sc**
Solvation (Lazaridis-Karplus)	**FA**	**fa_sol**
Dunbrack rotamer probability	**FA**	**fa_dun**
Statistical residue-residue pair potential	**FA**	**fa_pair**
Intra-residue repulsive Van der Waals	**FA**	**fa_intra_rep**
Electrostatic potential	**FA**	**hack_elec**
Disulfide statistical energies (S-S distance, etc.)	**FA**	**dslf_ss_dst, dslf_cs_ang, dslf_ss_dih, dslf_ca_dih**
Amino acid reference energy (chemical potential)	**FA/CEN**	**ref**
Statistical backbone torsion potential	**FA/CEN**	**rama**
Van der Waals "bumps"	**CEN**	**vdw**
Statistical environment potential	**CEN**	**env**
Statistical residue-residue pair potential (centroid)	**CEN**	**pair**
Cb		**cbeta**

Note that a number of scoring components are compatible with both full-atom and centroid mode.

3.3 Creating or editing a score function

Instead of using a pre-made score function, such as 'standard' or 'docking', you can also create your own score function from scratch using the various scoring components available in Rosetta. By definition, an empty score function is a score function in which all component weights are set to zero. To add a component, simply set the weight of the desired component to a non-zero number. In the following example, we build a simple score function that includes only the repulsive LJ-potential and hydrogen bonding.

```
>>my_scorefxn = ScoreFunction()
>>my_scorefxn.set_weight(fa_rep, 1.0)
>>my_scorefxn.set_weight(hbond_lr_bb, 1.0)
>>my_scorefxn.set_weight(hbond_sr_bb, 1.0)
```

3.4 Accessing more detailed scoring information

Beyond simply returning the total energy of a given pose, you can access a comprehensive breakdown of the contribution of each scoring component to the total score.

```
>>my_scorefxn.show(my_pose)
-----------------------------------------------------------
 Scores                       Weight   Raw Score Wghtd.Score
-----------------------------------------------------------
 fa_rep                       1.000     981.311     981.311
 hbond_sr_bb                  1.000     -56.655     -56.655
 hbond_lr_bb                  1.000    -103.050    -103.050
-----------------------------------------------------
 Total weighted score:                       821.606
```

The pose object also stores the latest energy calculations, and you can access this information through the **`energies()`** object. Through **`energies()`**, you can access a further breakdown of the scoring information on a residue by residue basis.

```
>>res_num = 5
>>my_pose.energies().show(res_num)
E       fa_rep  hbond_sr_bb     hbond_lr_bb
E(i)   5     0.53     0.00     0.00
```

You can also access individual scoring components from individual residues directly:

```
>>my_pose.energies().residue_total_energies(res_num)[fa_rep]
0.532552907292
```

Unit 4: Simple simulations in PyRosetta

In molecular modeling using PyRosetta we are generally searching a conformational space under a given energy function for the global minimum. The predominant sampling strategy used to search this conformational space is Monte Carlo-based sampling using a large number of short trajectories or paths. The lowest energy structure accessed in each trajectory is stored as a 'decoy'. Theoretically, assuming adequate sampling and discrimination, the lowest energy decoy corresponds to the global minimum.

4.1 Introduction to the Monte Carlo sampling algorithm

In Monte Carlo-base sampling, random perturbations, or 'moves' are made to a starting structure and those moves are either accepted or rejected based on the resulting change of energy due to that move.

The most common basis for accepting or rejecting a move is through the *Metropolis Criterion*. The Metropolis criterion states that if the change in energy (ΔE) is less than zero, that is the move *decreased* the energy, then always accept the move. If the change in energy is greater than zero, then accept that move only some of the time. The probability of accepting that move is a function of how much it increased the energy by:

$$P(\Delta E) = e^{-\Delta E/kT}$$

In practical terms this means:

1. A move is made to the structure
2. The energy is calculated and compared with the previous energy -> ΔE
3. If ΔE < 0 the move is accepted
4. If ΔE > 0, then:
    - The probability (*P*) of the move is calculated based on ΔE
    - A random number, *i*, is generated from 0 to 1
    - If $i < P$, then the move is accepted, otherwise the move is rejected

Note the role of *kT* in the calculation of the probability of acceptance. For a given ΔE, as *kT* increases, the probability of acceptance increases. At higher values of *kT*, the structure can more easily escape local minima, but the average energy of the structure is higher; at lower values of kT, the structure is more likely to get 'stuck' in a local minima, but will generally have a lower energy. In simulated annealing, *kT* starts at a high value and either linearly or geometrically decreases to a final value through the course of the simulation. This allows the structure to both escape local minima and also settle into a global minimum.

4.2 Monte Carlo Object

The **MonteCarlo** object keeps track of all variables necessary to run Monte Carlo simulations in Rosetta and also applies the Metropolis criterion. The **MonteCarlo** object is initialized with a score function to calculate the energy, a pose object to serve as a reference structure, and the temperature, which is used in the Metropolis Criterion:

```
>>kT = 1.0
>>mc = MonteCarlo(scorefxn, pose, kT)
```

Following a move to the pose object, the Metropolis Criterion is applied using:

```
>>mc.boltzmann(pose)
```

Within this function, the energy of inputted pose is calculated using the score function and compared to the energy of the last accepted pose object. The Metropolis criterion is applied to the pose; if the move was accepted than the inputted pose remains unchanged, and the last accepted pose, within the MonteCarlo object, is updated. If the move was rejected, the inputted pose is switched to the last accepted pose, and the last accepted pose is unchanged.

The lowest energy structure assessed by the MonteCarlo object can be accessed as well. The lowest energy structure is not only recovered at the end of the simulation, but often intermittently throughout the simulation as well.

```
>>mc.recover_low(pose)
```

For simulated annealing, the temperature (or kT) is decremented throughout the simulation. This can be done by changing the temperature that the MonteCarlo object uses for evaluating the Metropolis Criterion:

```
>>kT = 2.0
>>mc.set_temperature(kT)
```

In addition to applying the Metropolis Criterion, the Monte Carlo object stores a variety of information on acceptance and rejection:

```
>>mc.show_scores()
<<protocols.moves.MonteCarlo: MonteCarlo:: last_accepted_score,lowest_score:
   -8.02917 -8.02917

>>mc.show_counters()
<<protocols.moves.MonteCarlo:              unk trials=  60000;  accepts=
   0.6766;  energy_drop/trial=   0.00465

>>mc.show_state()
<<protocols.moves.MonteCarlo: MC: 1  -8.02917  -8.02917  -8.02917  -8.02917
   0  0  0  2
```

```
<<protocols.moves.MonteCarlo:              unk trials=  60000;  accepts=
   0.6766;  energy_drop/trial=   0.00465
-8.02916688224
```

Finally, the MonteCarlo object can be reset with a new pose. This will wipe clean all information about previous scores, acceptance rates, or last-accepted and lowest-energy poses.

```
>>mc.reset(pose)
```

4.3 A simple Monte Carlo simulation for peptide folding

Here is a simple Monte Carlo algorithm that folds a small polyalanine peptide from an extended strand to an α-helix. The score function consists purely of Hydrogen bonding and Van der Waals terms. Perturbations are made to the structure by randomly selecting a residue and then perturbing its φ and ψ by a random magnitude, from -25˚ to 25˚. The Monte Carlo object evaluates the Metropolis Criterion after each application of the perturbation.

```
p=Pose()
pose_from_pdb(p, "mc_initial.pdb")

#set up score function
scorefxn = ScoreFunction()
scorefxn.set_weight(hbond_sr_bb,1.0)
scorefxn.set_weight(vdw, 1.0)

#set up MonteCarlo object
mc = MonteCarlo(p, scorefxn, 1.0)

#set up mover
def perturb_bb = function(pose):
   resnum = randint(i, pose.total_residue())
   pose.set_phi(resnum, pose.phi(resnum)-25+random()*50)
   pose.set_psi(resnum, pose.psi(resnum)-25+random()*50)
   return pose

#set up protocol
def my_protocol = function(pose)
   mc.reset(pose)
   for i in range(1,60000):
      perturb_bb(pose)
      mc.boltzmann(p)

      if (i%1000 == 0):
          mc.recover_low(p)

   #output lowest-energy structure
   mc.recover_low(p)
   return pose

my_protocol(pose)
dump_pdb(p, "mc_final.pdb")
```

The above script starts with the starting structure (mc_initial.pdb), shown on the lower left. The lowest energy structure that is recovered (mc_final.pdb) is shown on the right:

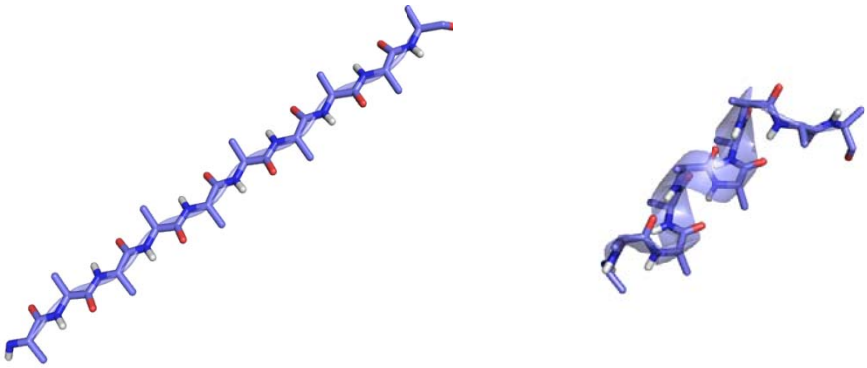


Figure 4.3 The initial extended strand structure (left) and the final helical structure (right) from the simple Monte Carlo simulation.

4.4 Job Distributor

Unlike in the previous example of peptide folding, in which a single trajectory lead to a folded solution, Rosetta protocols typically require a large number of short *independent* trajectories. The lowest-energy structure of each trajectory is stored as a 'decoy'. The lowest energy decoy generally represents the global minimum. The Job Distributor object is designed to facilitate the generation and storage of a large number of decoys and allow for the use of parallel processing to generate decoys.

The Job Distributor object directs decoy generation, outputs each decoy as a PDB file, and stores the energy and other metrics of each decoy in a score file. It is initialized with a file tag line that accompanies all files output by the job distributor, the number of decoys desired, and the score function that all decoys will be evaluated in for the score file. Optionally, a reference structure can be inputted, from which an RMSD to the decoy is calculated and recorded in the scorefile.

```
>>jd = PyJobDistributor("test_output", 5, scorefxn_high)
>>jd.native_pose = starting_pose
```

Once the Job Distributor is set up, it is used in the following way with a protocol (using the previous protocol as an example):

```
>>while (jd.job_complete == False):
   pose.assign(starting_pose)
   my_protocol( pose )
   jd.output_decoy(pose)
```

While a decoy is being generated, the Job Distributor will create a temporary file called **`test_output_0001.pdb.in_progress`**. Once that decoy is complete, it will be renamed **`test_output_0001.pdb`**, for decoy #1. Through the use of these temporary files, the script can be run multiple times for multiple processors all working on the same pool of decoys. In addition to the decoy structures, a score file is generated that lists each decoy, its RMSD to a reference structure and a break-down of its score. This scorefile is stored as **`test_output.fasc`** for all-atom structures and **`test_output.sc`** for centroid structures.

Additional information, such as particular measurements like a loop RMSD, or a specific residue-residue distance can be stored as an additional line in the scorefile can be added with the following line, just prior to outputting the decoy.

```
>>jd.additional_decoy_info("loop_RMSD" + str(loop_rmsd)" + "res49A_res20B" +
   str(res_dist)")
```

Note that if a MonteCarlo object is used in **`my_protocol`**, in the above example, it must be reset each time the function is called. Otherwise information from the previous decoy will be retained and recovered in the Monte Carlo object. This will lead to trajectories that are *not* independent.

Unit 5: Conformational sampling in PyRosetta – Movers

Rosetta uses a large variety of structural perturbations, or 'moves' that are specifically designed for efficient conformational sampling for proteins. These include moves that alter backbone torsion angles, optimize side chain conformations, manipulate rigid-body positions of multiple protein chains. While in theory all these moves can be enacted using the functions that manipulate protein geometry that we've already learned (i.e. `pose.residue().set_phi()`), in practice, these moves are often very specific, complex, combinations of smaller perturbations, that have been designed to search the conformation space in a *computationally efficient* manner. These moves, called `Movers`, are among the most powerful features of Rosetta and have been rigorously benchmarked and tested on protocols published in scientific literature.

5.1 Introduction to the Mover base class

`Movers` are one of the main archetypal classes in PyRosetta. After construction, their basic function is to be 'applied' to a pose, which, for most traditional movers, means perturbing the structure in some way. There are a large variety of `Movers` in PyRosetta. Learning to use them requires understanding three things:

1. `Mover` construction – what is needed to construct the mover? Some Movers are very simple and require almost nothing for construction. Others require many other objects that define how the Mover is implemented.
2. `Mover` options – what options exist to manipulate the Mover? In additional to options in during constructions, Movers often have a large number of variables that can be altered from the default settings.
3. `Mover.apply(pose)` – What does the mover *do* when you apply it to a Pose? The Mover will use all the instructions given to it during construction and option-setting to modify how the mover is implemented on a Pose object.

Just as the `ScoreFunction` object is principally passed a Pose object and returns its energy with `ScoreFunction(pose)`, a `Mover` is implemented on a Pose object with `Mover.apply(pose)`.

5.2 Limiting the search space with MoveMap

In Rosetta, we typically try to define the limits of the conformation space for a particular molecular modeling problem in terms of which degrees of freedom (in internal coordinates) we allow to be flexible, and which degrees of freedom we want to remain fixed. The `MoveMap` gives us a way to implement that with the `Movers`.

Movers typically apply changes on a protein structure by perturbing its internal coordinates, backbone torsion angles, side-chain torsion angles, and rigid-body jumps (see Unit 1). The `MoveMap` contains instructions about internal coordinates are allowed to move (be flexible) and which ones are to remain fixed. The `MoveMap` is a generic object that can be applied used with any Pose, but it is typically used for the specific application of a `Mover` or set of `Movers`. Construct a `MoveMap` as follows:

```
>>movemap = MoveMap()
```

On construction, all degrees of freedom are set to `False`, indicating that neither backbone torsion angles, nor side-chain torsion angles are allowed to move. We can set specific degrees of freedom to `True`, for example for residue 5. We can also set a range of residues to be true for backbone torsion angles:

```
>>movemap.set_bb(2, True)
>>movemap.set_chi(2,True)
>>movemap.set_bb_true_range(5,10)
```

We can allow the rigid-body orientation of one protein chain relative to another to be altered by allowing the jump (jump #1 in the example) that defines the orientation to be flexible:

```
>>movemap.set_jump(1, True)
```

Finally, you can view the instructions for the `MoveMap`:

```
>>movemap.show(10)
<<
  resnum       BB      CHI
     001    FALSE    FALSE
     002    TRUE     TRUE
     003    FALSE    FALSE
     004    FALSE    FALSE
     005    TRUE     FALSE
     006    TRUE     FALSE
     007    TRUE     FALSE
     008    TRUE     FALSE
     009    TRUE     FALSE
     010    TRUE     FALSE
```

5.3 Backbone Movers: SmallMover and ShearMover

The simplest movers that exist in Rosetta are the backbone movers `SmallMover` and `ShearMover`. They are used frequently to make small perturbations to the backbone structure for structural refinement and relaxation simulations.

SmallMover

The `SmallMover` makes small individual random perturbations on the φ and ψ backbone torsion angles of *n* residues among the all residues that are allowed to move. By default, *n* = 1. A `MoveMap` defines which residues are allowed to move. In making the perturbation, the `SmallMover` does the following each time:

1. Select random residue, *i*
2. For φ_i, select a perturbation magnitude randomly between (0 and `max_angle`)
3. new_φ_i = old_ φ_i ± perturbation
4. Do the same as Step 2-3 for ψ_i
5. accept or reject (new_ φ_i, new_ ψ_i) based on Metropolis Criterion, where kT is user inputted, and ΔE = `rama`(new_ φ_i, new_ ψ_i) – `rama(`old_ φ_i, old_ ψ_i). If move is accepted, apply move and continue. If move is rejected, go back to Step 2.

In the above description, rama is the Ramachandran score component (`rama`), based on the statistical probability of observing a given (φ,ψ) for a given residue type. This biases backbone torsion angle sampling towards allowable regions of the Ramachandran space and ensures that the simulation isn't wasting time sampling in disallowed regions.

The construction of a `SmallMover` requires a `MoveMap`, the number of moves the `SmallMover` should make for each `SmallMover.apply(pose)`, and the temperature that the SmallMover should use when applying the Metropolis Criterion during torsion angle selection. Additionally, the `max_angle` can be set for all residues, both in a secondary structure-specific, and secondary-structure independent manner:

```
>>movemap = MoveMap()
>>movemap.set_bb(True)
>>n_moves = 5
>>kT = 1.0
>>smallmover = SmallMover(movemap, n_moves, kT)
>>smallmover.angle_max(10)
>>smallmover.angle_max('E', 5)      #beta-strand residues
>>smallmover.angle_max('H', 10)     #helix residues
>>smallmover.angle_max('L', 20)     #loop residues
```

Finally, as with all Movers, a SmallMover is applied to a Pose with:

```
>>smallmover.apply(pose)
```

The `SmallMover` can be used to replace the `perturb_BB` function in the example peptide folding script in Section 4.3.

ShearMover

Most backbone movers in Rosetta sample backbone conformations by perturbing the internal coordinates of the protein. One drawback to this approach is that, in a continuous polypeptide chain, even small torsion angle perturbations in the beginning or middle of the chain can have large downstream consequences on the protein structure in Cartesian coordinates once the chain is 'refolded' with the new torsion angles. To accommodate this, Rosetta has a number of backbone movers that are designed to allow *local* perturbations of backbone conformation while minimizing *global* changes in the protein structure as a consequence. The most simple of these movers is the `ShearMover`.

In many respects the `ShearMover` is similar to the `SmallMover`. They both use the same arguments in construction. The difference is that while the `SmallMover` perturbs φ_i and ψ_i, the `ShearMover` perturbs, φ_i and ψ_{i-1}. The reason for this is a mathematical relationship that allows one of those two torsion angles to be perturbed and the other to be perturbed in such a way as to partially eliminate the downstream effect of the torsion angle perturbation, thus altering the torsion angle while minimizing the changes to the overall global structure. The syntax and usage of the ShearMover is identical to the SmallMover:

```
>>shearmover = ShearMover(movemap, n_moves, kT)
>>shearmover.angle_max(10)
>>shearmover.angle_max('E', 5)      #beta-strand residues
>>shearmover.angle_max('H', 10)     #helix residues
>>shearmover.angle_max('L', 20)     #loop residues
>>shearmover.apply(pose)
```

5.4 Backbone Movers: FragmentMovers

The `SmallMover` and `ShearMover` perturb the existing backbone torsion angles by a random amount. There is a second type of backbone mover that changes the torsion angle not by perturbing the original torsion angle of a single residue by some small amount, but by replacing the torsion angles for a set of consecutive residues, known as a'*fragment*' with a new set of torsion angles for those residues (a new fragment) derived from a database of low-energy fragments for that sequence of residues, known as a *fragment library*.

This method, known as *fragment insertion* is critical to Rosetta's *ab initio* structure prediction and loop modeling methods and allows for a fast, efficient, search of a much wide range of conformational space than SmallMovers and ShearMovers. In Rosetta we primarily use two lengths of fragments, 3mer fragments and 9mer fragments. 3mer fragments are predominantly used for most modeling applications as 9mer fragment insertion is generally too disruptive for all applications but protein folding.

A fragment library can be generated for a given protein sequence by going to Robetta website (http://robetta.bakerlab.org/fragmentsubmit.jsp) and submitting the desired sequence in FASTA format. The method for generating a fragment library involves searching a non-redundant subset of the Protein Data Bank for the 100 highest-frequency fragments that contain a similar sequence profile to the input sequence, in window-lengths of the same size as the fragment size. Theoretically, this method relies on the observation that local low-energy backbone conformations are partially a result of the local sequence. The fragment insertion method biases backbone sampling towards known low-energy conformations for a given local sequence.

A `FragmentMover` requires a fragment library (a `FragSet`) from which to select fragments, and a `MoveMap`, which specifies which residues are allowed to be altered. A fragment library can be read in as follows, in this example, for a 3mer fragment library file named `test_in_3mer.frag`:

```
>>fragset = ConstantLengthFragSet(3)
>>fragset.read_fragment_file("test_in_3mer.frag")

>>movemap = MoveMap()
>>movemap.set_bb(True)
>>frag_mover = ClassicFragmentMover(fragset, movemap)
>>frag_mover.apply(pose)
```

Like the `SmallMover`, standard fragment insertion that replace a 3-residue or 9-residue window of backbone torsion angles can have large downstream effects on the protein structure during the re-folding step, leading to drastic changes in the global structure. To address this, there is a fragment mover called `SmoothFragmentMover`, which selects fragments that minimize downstream effects. This leads to sampling of diverse local conformations without massively altering the global structure, making it ideal for structural refinement or relaxation.

```
>>frag_mover = SmoothFragmentMover(fragset, movemap)
>>frag_mover.apply(pose)
```

5.5 Energy Minimization

Minimizing the energy with respect to certain flexible degrees of freedom is a quick and easy way to lower the energy of a given structure without altering its structure substantially. Often significant decreases in energy can be achieved with minute changes in backbone or side-chain torsions. Energy minimization is common in Rosetta and often follows explicit perturbations and precedes a Monte Carlo Metropolis Criterion step. Essentially, it provides the lowest energy structure in the immediate local vicinity of a conformation just after an explicit perturbation step.

Energy Minimization in Rosetta is carried out primarily through the MinMover. In the construction of the MinMover, one mainly needs to supply the energy function that is to be minimized and a MoveMap which defines which conformational degrees of freedom to minimize over. Additionally, one can provide the minimization type used and the threshold for minimization (i.e. when minimization is considered complete), also known as the 'tolerance'.

```
>>movemap = MoveMap()
>>movemap.set_bb(True)
>>scorefxn = create_score_function('standard')
>>tolerance = 0.01
>>min_type = "dfp_min"

>>minmover = MinMover(movemap, scorefxn, min_type, tolerance, True)
>>minmover.apply(pose)
```

The MinMover can also be constructed with default settings where only specific options are changed later:

```
>>minmover = MinMover()
>>minmover.score_function(scorefxn)
>>minmover.movemap(movemap)
>>minmover.tolerance(tolerance)
```

There are primarily two minimization methods used in Rosetta: Linear minimization, or steepest-descent minimization (`linmin`), and Davidson-Fletcher-Powell minimization (`dfp_min`). Linmin is computational cheaper than dfp_min, but generally minimizes less well.

5.6 Other types of movers

In addition to the traditional movers that directly perturb the protein structure by altering their internal coordinates, there are other types of movers as well. Combination movers, such as the SequenceMover and RepeatMover, are essentially 'mover containers' that execute the mover(s) within them with some instruction. SequenceMover is a mover with a list of movers within it; when applied, it applies all the movers within it consecutively.

```
>>sequence_mover = SequenceMover()
>>sequence_mover.add_mover(small_mover)
>>sequence_mover.add_mover(minmover)
>>sequence_mover.apply(pose)
```

In the above example, when `sequence_mover.apply(pose)` is called, the SequenceMover will apply `small_mover` and then `minmover`.

`RepeatMover` is a mover that repeats the mover within it a user-specified number of times, in this case it will repeat the small_mover 5 times when it is applied to the pose.

```
>>repeats = 5
>>repeat_mover = RepeatMover(small_mover, repeats)
>>repeat_mover.apply(pose)
```

The most important of these 'container' movers is the `TrialMover`. The `TrialMover` contains a user-inputted `Mover` and `MonteCarlo` object. On `TrialMover.apply(pose)`, it executes the mover contained within it, and then applies `MonteCarlo.boltzmann()` on the resulting pose, accepting or rejecting that move based on the Metropolis Criterion.

```
>>mc = MonteCarlo(scorefxn, pose, kT)
>>trial_mover = TrialMover(small_mover, mc)
>>trial_mover.apply(pose)
```

The most common trial movers used in Rosetta involve using a sequence mover that makes one or more explicit perturbations, followed by energy minimization, followed by the `TrialMover`'s Metropolis Criterion:

```
>>smallmin = SequenceMover()
>>smallmin.add_mover(small_mover)
>>smallmin.add_mover(minmover)
>>smallmintrial = TrialMover(smallmin, mc)
>>smallmintrial.apply(pose)
```

Finally, we have a mover that switches a Pose between full-atom and centroid-mode representation. This is useful during multi-scale protocols in which a protein starts out in centroid-mode for most of the conformational search and then is converted into full-atom mode for refinement.

```
>>to_centroid = SwitchResidueTypeSetMover('centroid')
>>to_centroid.apply(pose)

>>to_fullatom = SwitchResidueTypeSetMover('fa_standard')
>>to_fullatom.apply(pose)
```

A pose that has just been converted into a full-atom pose has coordinates for all atoms at all side-chains but the torsion values for the side-chains is initialized to '0'. From here either the side-chain torsion angle conformations can be optimized to sensible values using side-chain packing (Unit 6), or they can be recovered from a reference full-atom structure, such as a starting structure (in the example below, `starting_pose`) using the `ReturnSidechainMover`:

```
>>recover_sidechains = ReturnSidechainMover(starting_pose)
>>recover_sidechains.apply( pose )
```

Unit 6: Side-chain Packing and Design

The main internal coordinates that define a protein structure are the backbone φ and ψ angles and the side-chain torsion angles χ_n. In Rosetta, the optimization of side-chain conformations is primarily handled through *side-chain packing*. The 'packer' is a self-contained algorithm that is called by many protocols and functions within Rosetta. The packer takes an input protein backbone structure, and uses a simulated-annealing Monte Carlo algorithm to identify the lowest energy side-chain conformations for each residue. Sampling in the packer is carried out by selecting '*rotamers*' or distinct low-energy side-chain conformations, from a *rotamer library*. Like the `MoveMap` that the backbone movers used in the previous unit, the side-chain packer movers use an object called a `PackerTask`, to provide instructions to the packer about what residues are allowed to move.

For a given residue, the side-chain packer selects the optimum rotamer from a set of rotamers for that residue, based its interactions with surrounding residues. In the case of simply optimizing the side-chain conformation, this rotamer set comprises exclusively of low-energy side-chain conformations for that residue type. In the case of protein design, however, the rotamer set used for packing will include the rotamers for all residue-types allowed in that position by the design instructions (again provided by the `PackerTask`). In this manner, the same algorithm is used for side-chain packing and protein design.

6.1 The PackRotamersMover

Side-chain packing in Rosetta is primarily carried out using a mover called the PackRotamersMover. Construction of a PackRotamersMover requires a score function that will be used in the side-chain packing algorithm within the packer, and a PackerTask, which specifies which residues are allowed to move. We will go over the PackerTask in the following section.

```
>>packer_task = standard_packer_task(pose)
>>scorefxn = create_score_function('standard')
>>pack_mover = PackRotamersMover(scorefxn, packer_task)
>>pack_mover.apply(pose)
```

On 'apply', the PackRotamersMover will optimize the side-chain conformations in the inputted pose while following any instructions from the PackerTask.

6.2 The PackerTask

The PackerTask provides the packer with restrictions by defining the rotamer set allowed for packing for each residue in the Pose. The standard_packer_task function creates a standard packer task that can be subsequently manipulated:

```
>>packer_task = standard_packer_task(pose)
>>print packer_task
<<
#Packer_Task

resid   pack?   design?  allowed_aas
1       1       1       ALA,CYS,ASP,GLU,PHE,GLY,HIS,HIS_D,ILE,LYS,LEU,MET...
2       1       1       ALA,CYS,ASP,GLU,PHE,GLY,HIS,HIS_D,ILE,LYS,LEU,MET...
3       1       1       ALA,CYS,ASP,GLU,PHE,GLY,HIS,HIS_D,ILE,LYS,LEU,MET...
4       1       1       ALA,CYS,ASP,GLU,PHE,GLY,HIS,HIS_D,ILE,LYS,LEU,MET...
5       1       1       ALA,CYS,ASP,GLU,PHE,GLY,HIS,HIS_D,ILE,LYS,LEU,MET...
6       1       1       ALA,CYS,ASP,GLU,PHE,GLY,HIS,HIS_D,ILE,LYS,LEU,MET...
….
```

Printing the PackerTask shows the instructions that are currently contained within the task. For each residue in the Pose (`resid`), it shows whether that residue is allowed to be packed, and allowed to be designed. A residue that is forbidden from being packed will maintain its original rotamer throughout the packing. A residue that is allowed to be packed, but not allowed to be designed, will maintain its original residue-type, and the packer will use a rotamer set containing only rotamers for that residue-type. A residue that is allowed to be packed and designed will be optimized for both residue-type and side-chain conformation, the packer will use a rotamer set containing rotamers from all allow-able residue-types. The allowable residue-types for each residue position are displayed as `allowed_aas`.

The ‘standard’ task has essentially no restrictions – it will allow both repacking and redesign of all residues in the Pose to any residue-type. This is the default PackerTask, all additional instructions serve to *restrict* the task in specific ways, such as by holding certain residues fixed. There are a number of functions that manipulate the PackerTask:

```
>>packer_task.restrict_to_repacking()
>>print packer_task
<<
resid   pack?   design? allowed_aas
1       1       0       ALA_p:NtermProteinFull,
2       1       0       GLU,
3       1       0       ALA,
4       1       0       LYS,
…
```

Restrict_to_repacking forbids all residues from being designed, thus allowing only the original residue-type for each residue.

```
>>packer_task.temporarily_fix_everything()
>>packer_task.temporarily_set_pack_residue(7, True)
>>packer_task.temporarily_set_pack_residue(8, True)
>>print packer_task
<<
resid   pack?   design? allowed_aas
```

```
...
5       0       0       SER
6       0       0       GLY
7       1       0       CYS
8       1       0       THR
9       0       0       ASP
...
```

The above functions fix *all* residues to their original rotamer, and allow repacking of only specific residues.

6.3 The Resfile

One way to input specific, custom, instructions for side-chain packing is by using an input file called the *resfile*. The resfile contains the same information as the pack_task and allows the user to directly manipulate each residue. A resfile can be generated for a given pose or PDB file as follows:

```
>>generate_resfile_from_pose( pose, "test.resfile")
>>generate_resfile_from_pdb( pdb_file, "test.resfile")
```

In order to edit the resfile you will have to open it with a Text Editor of your choice:

```
start
   1  A  NATRO
   2  A  NATRO
   3  A  NATRO
   4  A  NATRO
   5  A  NATRO
   6  A  NATRO
   7  A  NATRO
   8  A  NATRO
```

The first column is the residue number, the second column is the chain letter, and the third column contains instructions for that residue:

`NATRO` – instructs the packer to keep the sidechain conformation fixed. The rotamer set for that residue contains only the native rotamer.
`NATAA` – instructs the packer to maintain the original residue type, but allow the sidechain conformation of that residue to be optimized by the packer. The rotamer set for that residue contains all the rotamers from the rotamer library for that residue-type only.
`PIKAA XXXX` – instructs the packer to allow the side-chain to be mutated into any one of the proceeding amino acids (one-letter codes). The rotamer set for this residue contains all the rotamers from the rotamer library for all of the allowable residue-types.
`ALLAA` – instructs the packer to allow the side-chain to be mutated into any of the 20 amino acids. The rotamer set contains all of the rotamers in the rotamer library for all 20 amino acids.

A task can be created from the resfile using:

```
>>packer_task = standard_packer_task( pose )
>>packer_task.read_resfile("test.resfile")
>>packer = PackRotamersMover(scorefxn, packer_task)
>>packer.apply(pose)
```

Now, when you print the packer_task, it will reflect all of the instructions from the resfile.

6.4 TaskFactory

A `PackerTask` is fairly limited. It applies only to the `Pose` it was constructed for and is difficult to modify on the fly. If the `Pose` amino-acid sequence is changed, the original `PackerTask` that was constructed with it becomes obsolete. In some cases the exact residues that are to be packed varies as the structure changes. For example, in protein-protein docking, only the interface residues are supposed to be packed, but the residues that make up the interface change throughout the docking process. In these cases, a `TaskFactory` can be used, to create a task on the fly, each time `PackRotamers.apply(pose)` is called.

A `TaskFactory`'s primary job is to create a `PackerTask` based on a list of instructions, called `TaskOperations`, that given to it. Most instructions that can be given to a `PackerTask`, have analogous instructions for the `TaskFactory`.

```
>>tf = standard_task_factory()
>>tf.push_back(RestrictToRepacking())
>>tf.push_back(ReadResfile("test.resfile")
>>tf.push_back(RestrictToInterface( jump_num )
>>packer_task = tf.create_task_and_apply_taskoperations( pose )
```

In the above example, the `TaskFactory` will generate a task that restricts all residues to repacking, puts in additional restrictions based on a user-inputted `resfile`, and finally restricts packing to residues at the interface defined by the `Jump` 'jump_num'.

Additionally, the `TaskFactory` can be sent to a `PackRotamersMover`. The `Mover` will then create a new `PackerTask` each time `PackRotamers` is applied:

```
>>packer_mover = PackRotamersMover(scorefxn)
>>packer_mover.task_factory(tf)
>>packer_mover.apply(pose)
```

A list of some of the `TaskOperations` available include:

TaskFactory TaskOperations	
`tf = standard_task_factory()`	Creates a default TaskFactory
`tf.push_back(IncludeCurrent())`	includes the current rotamers in the Pose to the rotamer sets used for packing. Defaulted on
`tf.push_back(ReadResFile("test.resfile"))`	applies instructions from the resfile when creating a PackerTask
`tf.push_back(NoRepackDisulfides())`	holds disulfide bond cysteine side-chains fixed. Defaulted on.
`tf.push_back(RestrictToInterface(1))`	allows repacking only at the interface defined by the jump number (in example jump# 1)

`pr = PreventRepacking()`
`  pr.include_residue( 5 )`
`  tf.push_back(pr)`
	turns off repacking for specified residues (residue #5 in example)

`rr = RestrictResidueToRepacking()`
`rr.include_residue(5)`
`tf.push_back(rr)`
	turns on repacking for specified residues (residue #5 in example)

6.5 Other side-chain movers

Besides side-chain packing there are two other side-chain movers. The first is the RotamerTrials mover. This mover acts as a 'cheap' version of the standard packer. It's much faster than standard packing and quickly finds local minima in side-chain conformation space. The standard `PackRotamersMover` is much better at finding the global minimum in side-chain conformation space. `RotamerTrialsMinMover` is a variation of the `RotamerTrialsMover` that is uses energy minimization to minimize the torsion angles while selecting rotamers. It is the only mover capable of going off-rotamer in search of low-energy side-chain conformations.

```
>>rot_trials = RotamerTrialsMover(scorefxn, tf)
>>rot_trials.apply( pose )
>>rt_min = RotamerTrialsMinMover(scorefxn, tf)
>>rt_min.apply( pose )
```

Finally, there are a number of wrapper functions that use the previous movers in highly specific, but commonly used ways. For example, the `mutate_residue` function creates a point mutation at a user-specified position on the pose. Below, residue #5 in the `Pose` is mutated to a Serine.

```
>>mutate_residue( pose, 5, 'S')
```

Unit 7: Methods and Protocols

There are a number of standardized methods and protocols that use various movers to accomplish certain modeling objectives. These protocols, such as loop modeling, protein-protein docking, or protein design, are published in the scientific literature, rigorously benchmarked, and already in frequent use by many in the molecular modeling community. Here we will introduce some of the protocols that are available in PyRosetta. Please see the PyRosetta scripts section of the PyRosetta website (http://www.pyrosetta.org) an updated list of available protocols and a sample script demonstrating their usage.

7.1 Loop Modeling

In loop modeling, the objective is to sample low-energy conformations over a defined loop, or range of residues. The challenge in this method is the loop residues are being sampled by altering their backbone torsion angles, but the overall global structure of the protein (the non-loop residues) are to remain fixed in space. The range of torsion angles in the loop residues that satisfies this constraint is significantly smaller than the entire range of space accessible in the allowable region of Ramachandran space. Any local torsional perturbation, mid-loop, implemented by a `SmallMover` or `FragmentMover`, will alter the downstream global structure of the protein during refolding. Rosetta addresses this by allowing chain-breaks to form in the loop during loop modeling that allow local structures to be sampled while maintaining the global structure of the protein. Then it uses a second algorithm to mend the chain-breaks, leading to structures with varying low-energy, closed-loop conformations, that all have the same global structure.

For a protein of length *n*, Rosetta defines a 'loop' between residues *i* and *j*, a 'jump' connecting residues *i* and *j*, and a 'cutpoint' within the loop, in between residues *i* and *j* where the chain-break will occur (see Figure 7.1). As was mentioned in Unit 2, the jump defines the position of the second jump point in rigid body space relative to the first jump point. During refolding, the Pose is refolded via its φ,ψ,ω values, placing residues in Cartesian space residue-by-residue, in the standard N→C direction starting from residue 1 until it hits a cutpoint. Once it arrives at the cutpoint residue, it uses the jump to place the position of residue *j* in Cartesian space, relative to residue *i*, which has already been placed in Cartesian space through refolding. Then it refolds from residue *j*, to the cutpoint, in the C→N direction, and refolds from residue *j* to residue *n*, in the N→C direction. In this manner, φ/ψ perturbations between residues *i* and *j* will affect only the structure of residues between *i* and *j* without altering the rest of the structure. The downstream impact of torsional perturbations is eliminated by the presence of a 'chain-break'.

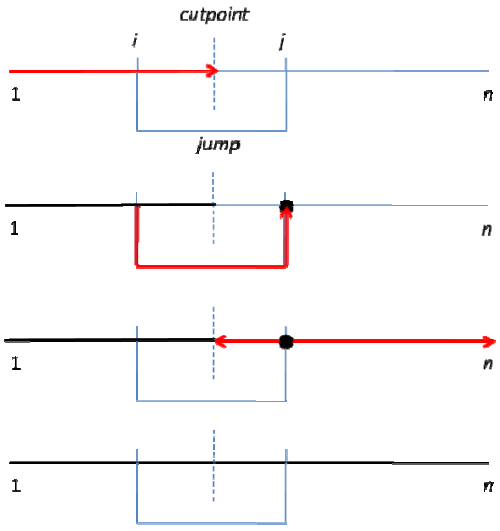


Figure 7.1: The refolding process using a loop modeling fold tree

In Rosetta, by convention, a loop between residues *a* and *b*, is defined by residues *i* and *j*, where *i* = *a* - 2 and *j* = *b* + 2. The `Loop` object in Rosetta is defined by the residues that define the loop (*i* and *j*), and the cutpoint between the loop (a residue in between *i* and *j*).

```
>>i = 70
>>j = 80
>>cutpoint = 75
>>n = pose.total_residue()
>>loop1 = Loop(i, j, cutpoint)
```

Additionally, a set of loops can be specified by the `Loops` object. Most Loop protocol movers use a Loops object as an input and can be used for single loops or multiple loops:

```
>>loop_list = Loops()
>>loop_list.add_loop(loop1)
```

7.2 Loop-modeling Fold Tree

In Rosetta, a `FoldTree` is used to define instructions for refolding. A `FoldTree` consists of a list of 'edges' that can each be of one of two types, a 'peptide edge' (type -1), or a jump edge, (type *n*). A `Pose` always contains a `FoldTree` directing its refolding which can be accessed:

```
>>print pose.foldtree()
<<
```

A new foldtree can be constructed and then passed in to the Pose object. Below is a construction of the fold tree described in Figure 7.1. The syntax for adding an edge is:

`FoldTree().add_edge(node1, node2, edge_type)`. Note that all 'peptide' edges have an type of -1, while the jump has an edge of type that is a positive integer, corresponding to the jump number, in this case, 1. Also note that the direction of folding (N→C or C→N) is described by the order of nodes used to define the edge. For N→C folding, `node1` < `node2`, for C→N folding, `node1` > `node2`.

```
>>ft = FoldTree()
>>ft.add_edge(1,i,-1)
>>ft.add_edge(i,cutpoint,-1)
>>ft.add_edge(i,j,1)
>>ft.add_edge(cutpoint, j, -1)
>>ft.add_edge(j,n)

>>pose.foldtree( ft )
```

Finally you can check if a FoldTree is valid:

```
>>ft.check_fold_tree()
<<True
```

In the above example, we have defined a FoldTree manually and entered it into the Pose. For most PyRosetta protocols, functions exist to define the fold tree for that particular protocol automatically. A user would only manually define a fold tree for highly customized algorithms where automatically generated, protocol-specific fold trees cannot be used.

A single loop fold tree can be defined using the Loop object for a Pose:

```
>>set_single_loop_fold_tree(pose, loop1)
>>print p.foldtree()
<<
```

7.3 Loop Modeling protocol movers

Cyclic-Coordinate Descent (CCD) Loop Closure

During sampling, using `SmallMovers`, `ShearMovers`, and/or `FragmentMovers` on loop residues on a Pose with a loop-modeling FoldTree, a chain break will form at the cutpoint. Rosetta primarily uses CCD loop closure to close that chain-break to recover a continuous chain along the polypeptide, from the N to the C terminus. CCD loop closure is carried out using the `CcdLoopClosureMover`. It uses Cyclic-coordinate descent to sample the torsion angles of the loop residues in a way that minimizes the chain-break. On construction it requires a `Loop` object, and a `MoveMap` that allows the loop residues to move:

```
>>movemap = MoveMap()
>>movemap.set_bb_true_range(i,j)
>>ccd = CcdLoopClosureMover(loop1, movemap)
```

```
>>ccd.apply(pose)
```

Low-resolution Loop Modeling

The low-resolution phase of loop-modeling, described in (Rohl *et al.*) can be accessed using the LoopMover_Perturb_CCD. This mover automatically sets up a fold tree based on an input pose and the Loops object. On construction it requires a `Loops` object to define the loops, a `FragSet` that is used by Fragment movers during sampling, and the `ScoreFunction` under which sampling is carried out.

```
>> scorefxn = create_score_function_ws_patch('cen_std', 'score4L')
>>fragset = ConstantLengthFragSet(3,"test_in_3mer.frag")

>>loop_perturb = LoopMover_Perturb_CCD(loop_list, scorefxn, frag_set)
>>loop_perturb.apply( pose )
```

Additionally a number of options can be set, including randomizing the input loop conformation:

```
>>loop_perturb.randomize_loop(True)
```

Note: The `LoopMover_Perturb_CCD` mover does not work in PyRosetta v1.0. Please see the loop modeling example on the website (http://www.pyrosetta.org/scripts.html) for how to model loops without this mover.

High-resolution Loop Refinement

The high-resolution phase of loop modeling is carried out using the `LoopMover_Refine_CCD` mover. On construction it requires a `Loops` object to define the loops, and a `ScoreFunction` under which sampling is carried out.

```
>>scorefxn = create_score_function('standard', 'score12')
>>loop_refine = LoopMover_Refine_CCD(loop_list, scorefxn)
>>loop_refine.apply( pose )
```

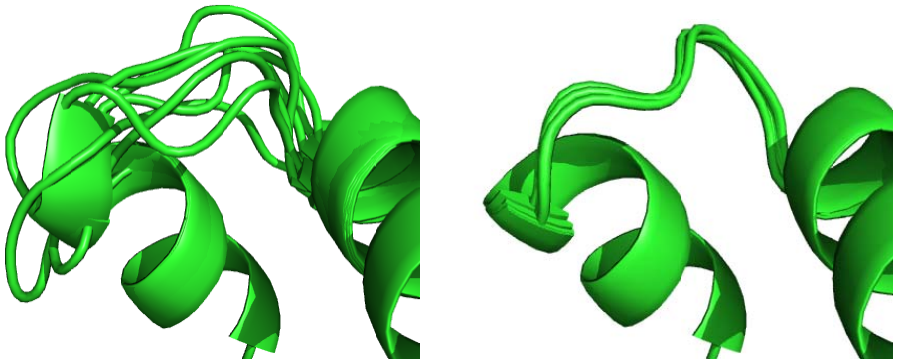


Figure 7.3 Five structures generated from the same starting structure using the LoopMover_Perturb_CCD mover (left) after randomizing the starting loop conformation, and the LoopMover_Refine_CCD mover (right).

7.4 Protein-protein docking

Protein-protein docking is used to predict the structure of protein complexes starting from their unbound components. In docking, different docked configurations, or rigid-body orientations of the two partners are sampled, and, theoretically, the lowest-energy docked configuration corresponds to a near-native protein complex structure. Like in Loop Modeling, there is a low-resolution perturbation/search phase and a high-resolution refinement phase.

The FoldTree used in docking is identical to the one used in loop modeling except that the jump points are defined differently. A chain-break is set in between the two partners (A and B), and the jump points correspond to residues closest to the center of masses for each partner, (residues *i* and *j*, respectively). The fold tree is as follows, and the refolding process is the same as in loop modeling:

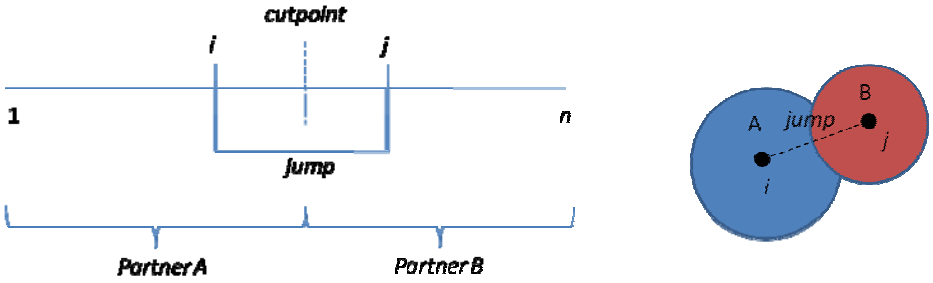


Figure 7.4 Docking fold tree

The docking `Foldtree` can be set up automatically using the `DockingProtocol()`:

```
>>DockingProtocol(jump_num).setup_foldtree(pose)
>>print pose.foldtree()
<<FOLD_TREE  EDGE 1 190 -1  EDGE 190 238 -1  EDGE 190 285 1  EDGE 285 239 -1
   EDGE 285 301 -1
```

By default it will create a jump between the first two chains in the Pose. For multi-chain partners, such as anti-bodies, the chains for each partner must be specified. In the example below, partner 1 is consists of chains ‘H’ and ‘L’, and partner 2 consists of chain ‘A’:

```
>>DockingProtocol(jump_num).setup_foldtree(pose, ‘HL_A’)
```

7.5 Rigid Body Movers

A jump defines the relative orientation of two residues (jump points) in rigid-body space. This requires three dimensions in Cartesian coordinates define the relative *position* of one residue to the other, and three dimensions in polar coordinates that define the relative *orientation* of one residue to the other. Finally, a rigid-body center, which defines the coordinate frame of the jump is also defined. These values can be accessed and set as vectors (R and T) for each jump:

```
>>print pose.jump(1)
>>pose.jump(1).get_translation()
>>pose.jump(1).get_rotation()
```

For rigid body perturbations, we generally use `RigidBodyMover`s that perturb the Jump rather than manually altering the R and T values themselves. For example, the `RigidBodyTransMover` moves the partners along the axis defined by the jump between the two partners. It can be used to move the partners towards or away from each other.

```
>>trans_mover = RigidBodyTransMover( pose, jump_num )
>>trans_mover.step_size(50)
>>trans_mover.apply( pose )
```

Additionally, a translation axis can be manually accessed and modified.

```
>>new_axis = xyzVector()
>>new_axis = trans_mover.trans_axis()
>>new_axis.negate()
>>trans_mover.trans_axis(new_axis)
```

In the following section there are additional examples of `RigidBodyMover`s, including `RigidBodyPerturbMover`, `RigidBodyRandomizeMover`, and `RigidBodySpinMover`.

7.6 Docking Protocol Movers

In the standard protein-protein docking algorithm we have three stages: 1) the initial perturbation, 2) low-resolution search and 3) high-resolution refinement. The first two stages are carried out in centroid-mode, the final stage is in full-atom mode.

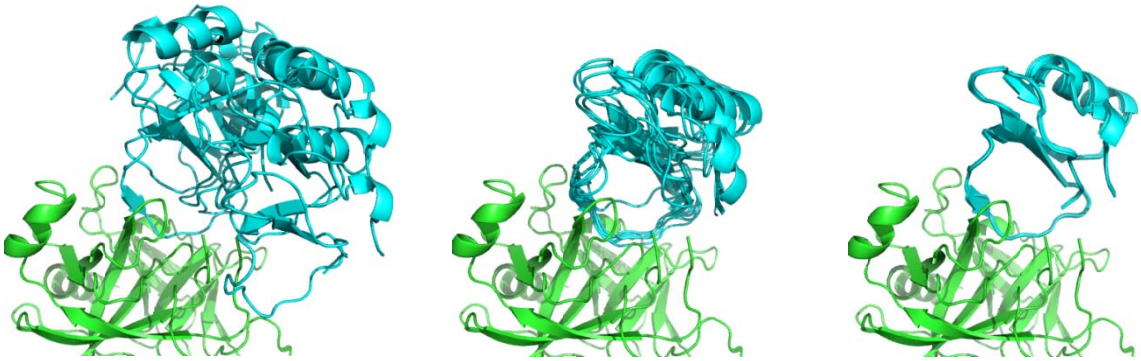


Figure 7.6. Relative size of perturbations during the initial perturbation phase of docking (left) for a dock_pert 3 8, the centroid-mode phase of docking (middle), and the high-resolution refinement phase of docking (right).

In the initial perturbation stage, the position of partner 2 relative to partner 1 is perturbed from its starting position. For a global docking run, one or both of the partners orientation is randomized. For a local docking perturbation, it is perturbed by 3A and 8 degrees on average. Additionally, one of the partners can be 'spun' about an axis defined by the jump by a random amount relative to the other. Note that '`partner_upstream`' and '`partner_downstream`' are hard-coded terms that refer to partner 1 and 2.

```
>>jump_num = 1
>>randomize1 = RigidBodyRandomizeMover(pose, jump_num, partner_upstream)
>>randomize2 = RigidBodyRandomizeMover(pose, jump_num, partner_downstream)
>>randomize1.apply( pose )
>>randomize2.apply( pose )

>>dock_pert = RigidBodyPerturbMover(jump_num, 3, 8)
>>spin = RigidBodySpinMover( jump_num )
>>dock_pert.apply( pose )
>>spin.apply( pose )
```

Finally, following a perturbation, the two partners must be brought back into contact with one another if any of the perturbations moved the partners away from each other:

```
>>slide_into_contact = DockingSlideIntoContact( jump_num )
>>slide_into_contact.apply( pose )
```

This function works only for centroid mode, but there is also a full-atom version of this mover:

```
>>slide_into_contact = FaDockingSlideTogether( jump_num )
```

Once the initial perturbation is complete, the low-resolution phase docking can be carried out using the `DockingLowRes` mover:

```
>>scorefxn_low = create_score_function('interchain_cen')
>>docking_low = DockingLowRes(scorefxn_low, jump_num)
>>docking_low.apply( pose )
```

Following the low-resolution phase of docking, the structure must be converted into a full-atom structure for the high-resolution refinement step. This can be done as described in Section 5.5, or with a `DockingProtocol()` function written specifically for this purpose (in the example below recovering the side-chains from a full-atom starting structure called `starting_pose`):

```
>>DockingProtocol().recover_sidechains(pose, starting_pose)
```

Finally, high-resolution refinement is carried out using the `DockingHighRes` mover:

```
>>scorefxn_high = create_score_function('docking')
>>docking_high = DockingHighRes(scorefxn_high, jump_num)
>>docking_high.apply( pose )
```

Additionally, there are a number of options that can be set that modify the `DockingHighResMover`. The `MoveMap` in this mover is primarily used during minimization – inputting a custom `MoveMap` can allow for energy minimization along additional degrees of freedom beyond rigid-body minimization, for example, backbone minimization.

```
>>docking_high.set_move_map( movemap )
>>docking_high.set_min_type( 'dfpmin' )
```

The DockingHighRes mover uses a variation of the RigidBodyPerturbMover that uses the center of masses of the interface residues to define the reference frame for rigid body perturbations instead of the center of masses of the entire partners which is the default behavior. It also uses translation and rotation magnitudes of 0.1Å and 5.0˚ respectively.

```
>>use_interface = True
>>rbmover = RigidBodyPerturbMover( jump_num, 0.1, 5.0, partner_downstream,
   use_interface)
```

7.7 Modeling small molecules in PyRosetta

PyRosetta is generally set up to model proteins using the 20 standard amino acids. Small-molecules and other non-amino acids moieties, such as post-translational modifications and cofactors, are often critical to accurate modeling certain systems. In PyRosetta, these non-standard molecules are treated as additional 'residues' to the standard residue set.

Parameters describing the chemical and atomic properties of the standard residue set are stored in the `minirosetta_database` directory within PyRosetta. In order to use non-standard

molecules, a parameter file must first be created so that PyRosetta can properly model the structure and energies of the non-standard molecule. A parameter file (called a `.params` file) must be generated from an MDL Molfile format (`.mol` file) which contains the necessary structural and connectivity information and can be created from a PDB file containing the atomic coordinates for the molecule (`.pdb`):

1. Isolate the atomic coordinates of the non-standard molecule into a PDB file
2. Generate a .mol file from the .pdb file. This can be done with the free web tool MN.CONVERT at http://www.molecular-networks.com/demos
3. Use the molfile_to_params.py (found in the ligand docking download at http://www.pyrosetta.org/scripts.html) on the .mol file to generate 1) a .params file and 2) a PDB file containing the atomic coordinates of the non-standard molecule using Rosetta-standardized atom-types.

```
>>molfile_to_params.py ATP.mdl –n ATP
outputs: ATP_0001.pdb ATP.params
```

4. Replace the coordinates of the non-standard molecule in the PDB file of the starting structure with the coordinates output from the script in the previous step.

To load the nonstandard-molecule, first create a list of all nonstandard-molecule parameter files, then create a residue set that includes them, and finally, use that expanded residue set to read in the PDB file:

```
>>params_list = Vector1['ATP.params']
>>res_set = generate_nonstandard_residue_set(params_list)

>>pose_from_pdb(pose, res_set, "PKA.pdb")
```

It is important to note that non-standard molecules are currently only supported in a full-atom representation.

Finally, as the small molecule is modeled as a residue in the Pose object, a residue in the Pose object can be queried as to whether it is a ligand or is one of the standard 20 amino acids. In the example below, residue 105 is a small molecule:

```
>>pose.residue(105).is_protein()
<<False
>>pose.residue(105).is_ligand()
<<True
```

7.8 Modeling DNA and RNA

Parameters for all the standard nucleotides in DNA and RNA are already loaded into the minirosetta_database, so modeling DNA and RNA is straightforward in PyRosetta. Each nucleotide in the DNA or RNA is considered a 'residue' in the Pose object. Like with small molecules, DNA and RNA is only supported in the full-atom mode, parameters do not currently exist for modeling it in centroid mode.

As with the small molecule, a residue can be queried as to whether it is a standard amino acid, or a nucleotide (again, in the case of residue 105):

```
>>pose.residue(105).is_protein()
<<False
>>pose.residue(105).is_ligand()
<<True
```

Note that the names of the atoms in the DNA or RNA have to follow a standardized form. Please see the DNA modeling example in the scripts section of the PyRosetta website (http://www.pyrosetta.org/scripts.html) for more information.

Appendix: PyRosetta Reference Sheet

Python Commands and Syntax	
`i = 1` `j = "Bob"`	variable assignments
`print j, " thinks ", i, " = 0."`	prints Bob thinks 1 = 0.
`for i in range(1,10):` `  print i`	The newly defined variable "i" ranges from 1 up to,(but not including) 10 and the command **`print i`** is executed for each value.

`if x < 0:`
`  x = 0`
`  print`
`elif x==0:`
`  print "zero"`
`else:`
`  print "positive"`
	Conditional statement that executes lines only if Boolean statements are true.
	
	Use indenting to indicate blocks of code executed together under the conditional

`def myfunc(a, b)`
`  # code here`
`return c,d,e`
	Defines a function. Also acceptable, **`return(c, d, e)`**, but not **`return[c, d, e]`**

`returned_values = myfunc(a, b)`
`value_of_c = returned_values[0]`
`value_of_d = returned_values[1]`
`value_of_e = returned_values[2]`
	syntax for using multiple values returned by a function called with variables a and b.

`outfile = open('out.txt','w')`
`print >>outfile "hello"`
`outfile.close()`
	prints hello to a new file named **`out.txt`**

`outfile.write(` `  str(i)+";"+str(score) +"\n")`	alternate way to write to a (previously opened) file

Python Math	
`import random`	imports random number functions from Python
`random.random()`	returns a random float between 0 and 1
`random.randint(5,10)`	returns a random integer between 5 and 10 (inclusive)
`random.gauss(5,10)`	returns a random number from a Gaussian distribution with a median of 5 and a standard deviation of 10
`import math`	imports math functions from Python
`math.exp(5)`	returns the value of e^5

Rosetta: Vector	
`rosetta.utility.vector1_string`	creates a C++ string vector in Python
`rosetta.utility.vector1_float`	creates a C++ float vector in Python
`rosetta.utility.vector1_int`	creates a C++ int vector in Python
`rosetta.utility.vector1_bool`	creates a C++ bool vector in Python
`v1 = rosetta.utility.vector1_int` `  v1.append(5)`	Appends an element to a C++ vector
`v1 = numeric.xyzVector(x,y,z)`	Creates a C++ xyz vector used for Cartesian coordinates

Rosetta: Pose Object	
`pose = Pose()`	Creates a an empty pose object.
`pose_from_pdb(pose,"/path/to/ input_file.pdb")`	Creates new object called pose from the pdb file.
`make_pose_from_sequence(pose, "AAAAAA", "fa_standard")`	Creates a pose from the given sequence string using standard residue type templates
`print pose`	Displays PDB filename, sequence, and fold tree
`pose.assign(otherpose)`	Copies 'otherpose' onto 'pose'. You cannot simply write "pose = otherpose", as that will only point 'pose' to 'otherpose' and not actually copy it.
`dump_pdb(pose, "/path/to/output_file.pdb")`	Creates pdb file named output_file.pdb using information from pose object.
`pose.total_residue()`	Returns number of residues in pose
`pose.phi(5)` `pose.psi(5)` `pose.chi(2,5)`	Returns the ϕ or ψ angle of the 5th residue in the pose; returns 2nd χ of the 5th residue
`pose.set_phi(5,60.0)` `pose.set_psi(5,60.0)` `pose.set_chi(2,5, 60.0)`	Sets the ϕ or ψ angle of the 5th residue in pose to 60°; sets the 2nd χ of the 5th residue to 60°
`print pose.residue(5)`	Prints the amino acid details of residue 5
`print pose.residue(5).xyz(2)`	Prints the numeric.xyzVector for the second atom (CA) of residue 5
`pose.conformation().set_bond_length(atom1 ,atom2,length)`	Sets the bond length between objects "atom1" and "atom2" to a value of "length".
`pose.conformation().set_bond_angle(atom1, atom2,atom3,bond_angle)`	Sets the bond angle of objects "atom1," "atom2" and "atom3" to a value of "bond _angle".
`atomN = pose.residue(5).atom('N')`	Creates a pointer to the N atom object of residue 5
`coord = atomN.xyz()` `print coord.x, coord.y, coord.z`	Prints the Cartesian coordinates of atomN
`NCbond = atomN.xyz() – atomC.xyz()` `print NCbond.norm()`	Calculates and prints the distance between atomN and atomC
`for i in range (1,pose.total_residue()+1): <command> # on pose.residue(i)`	Loops through all residues in pose and runs <command> on each one
`pose.pdb_info().name()`	Gives the name of the PDB file input to pose
`pose.pdb_info().number(i)`	Gives the PDB number of residue i
`pose.pdb_info().chain(i)`	Gives the PDB chain of residue i
`pose.pdb_info().icode()`	Gives the PDB insert code of residue i
`pose.pdb_info().pdb2pose("A",100)`	Gives the pose's internal residue
`pose.pdb_info().pose2pdb(25)`	Gives the PDB chain/number from pose number

Rosetta: Scoring	
`scorefxn = create_score_function('standard')`	Defines a score function with standard full-atom energy terms and weights
`scorefxn2=core.scoring.ScoreFunction()` `scorefxn2.set_weight(core.scoring.fa_atr, 1.0)`	Defines a function called "scorefxn," in which the energies accounted for are: The numbers are the relative weights assigned to each energy and can be set to any real value. This is not an inclusive list of energies.
`print scorefxn`	Shows score function weights and details
`scorefxn(pose)`	Returns the score of pose with the defined function "scorefxn".
`scorefxn.show(pose)`	Returns the score of pose with the defined function "scorefxn".
`pose.energies().show()` `pose.energies().show(resnum)`	Shows the breakdown of the energies by residue
`emap = rosetta.core.scoring.TwoBodyEMapVector()`	Creates an energy map object to store a vector of scores
`scorefxn.eval_ci_2b(rsd1,rsd2,pose,emap)`	Evaluates context-independent two-body energies between residues rsd1 and rsd2 and stores the energies in the energy map
`print emap[rosetta.core.scoring.fa_atr]`	Print fa_atr term from the energy map
`hbond_set = rosetta.core.scoring.hbonds.HBondSet()`	Creates an HBond set object for storing hydrogen bonding information
`pose.update_residue_neighbors();` `rosetta.core.scoring.hbonds.fill_hbond_set(pose,False,hbond_set)`	Stores H-bond info from pose in the Hbond_set object.
`hbond_set.show(pose)`	Prints H-bond info from the hbond_set
`calc_total_sasa(pose, 1.5)`	Calculates the total solvent-accessible surface area using a 1.5A probe

Rosetta Full-atom Scoring Functions		
Van der Waals net attractive energy	FA	fa_atr
Van der Waals net repulsive energy	FA	fa_rep
Hydrogen bonds, short and long-range, (backbone)	FA/CEN	hbond_sr_bb, hbond_lr_bb
Hydrogen bonds, short and long-range, (side-chain)	FA	hbond_sc, hbond_bb_sc
Solvation (Lazaridis-Karplus)	FA	fa_sol
Dunbrack rotamer probability	FA	fa_dun
Statistical residue-residue pair potential	FA	fa_pair
Intra-residue repulsive Van der Waals	FA	fa_intra_rep
Electrostatic potential	FA	hack_elec
Disulfide statistical energies (S-S distance, etc.)	FA	dslf_ss_dst, dslf_cs_ang, dslf_ss_dih, dslf_ca_dih
Amino acid reference energy (chemical potential)	FA/CEN	ref
Statistical backbone torsion potential	FA/CEN	rama
Van der Waals "bumps"	CEN	vdw
Statistical environment potential	CEN	env
Statistical residue-residue pair potential (centroid)	CEN	pair
Cb	CEN	cbeta

Residue Type Set Mover	
`switch = SwitchResidueTypeSetMover('centroid')`	creates a mover which will change poses to the centroid residue type set ('fa_standard' also avail.)
`switch.apply(pose)`	changes pose to the centroid residue types

MoveMap	
`movemap = MoveMap()`	creates a MoveMap
`movemap.show(Nres)`	prints the MoveMap contents for residues 1-Nres
`movemap.set_bb(True)`	Allows all backbone torsion angles to vary when movemap is applied
`movemap.set_chi(True)`	Allows all side chain torsion angles (χ) to vary when movemap is applied
`movemap.set_bb(10,False)` `movemap.set_chi(10,False)`	Forbid residue 10's backbone and side chain torsion angles from varying
`movemap.set_bb_true_range(10,20)`	Allows backbone torsion angles to vary in residues 10 to 20, inclusive; sets all other residues to False.
`movemap.set_jump(1, True)`	Allows jump #1 to be flexible

Fragment Movers	
`fragset = ConstantLengthFragSet(3, "aatestA03_05.200_v1_3")`	creates a fragment set and loads the fragments from the data file
`mover_3mer = ClassicFragmentMover(fragset,movemap)`	Creates a fragment mover using the fragset and the movemap
`mover_3mer.apply(pose)`	inserts a random 3-mer fragment from the fragset into the pose, only in positions allowed by the movemap
`smoothmover = SmoothFragmentMover(fragset, movemap)`	Fragment insertions are followed by a second, downstream fragment insertion chosen to minimize global disruption

Small and Shear Movers	

`kT = 1.0`
`n_moves = 1`
`smallmover = SmallMover(movemap,kT,n_moves)`
`shearmover = ShearMover(movemap,kT,n_moves)`
	creates a small or shear mover with a movemap, a temperature, and the number of moves

`smallmover = SmallMover()` `shearmover = ShearMover()`	Default settings are all backbone moves allowed, *kT* = 0.5, and n_moves = 1
`smallmover.apply(pose)` `shearmover.apply(pose)`	applies the movers

Minimize Mover	
`minmover = MinMover()`	creates a minimize mover with default arguments
`minmover = MinMover(movemap, scorefxn, min_type, tolerance, True)`	Creates a minimize mover with a particular MoveMap,ScoreFunction, minimization type, or score tolerance
`minmover.movemap(movemap)`	Set a movemap
`minmover.score_function(scorefxn)`	Set a scorefunction
`minmover.min_type('linmin')`	Set a the minimization type to a line minimization (one direction in the space)
`minmover.min_type('dfpmin')`	Set a the minimization type to a David-Fletcher-Powell minimization (multiple iterations of linmin in conjugate directions)

`minmover.tolerance(0.5)`	Set the mover to iterate until within 0.5 score points of the minimum
`minmover.apply(pose)`	Apply the minmover to a pose

MonteCarlo	
`mc = MonteCarlo(pose,scorefxn,kT)`	creates a MonteCarlo object
`mc.set_temperature(1.0)`	Sets the temperature in the MonteCarlo object
`mc.boltzmann(pose)`	Accepts or rejects the pose object, compared to the pose last time the mc object was called, according to the standard Metropolis criterion.

`mc.show_scores()`
`mc.show_counters()`
`mc.show_state()`
	Shows stored scores, counts of moves accepted/rejected, or both.

`mc.recover_low(pose)`	Sets the pose to the lowest-energy configuration ever seen during the search
`mc.reset(pose)`	Resets all counters and sets the low- and last-pose to the current pose state.

TrialMover	
`smalltrial = TrialMover(smallmover,mc)`	Creates a mover which will apply the small mover, then call the MonteCarlo object mc. This mover will also give more explicit tags for the `mc.show_state()` output.
`smalltrial.num_accepts()`	Number of times the move was accepted
`smalltrial.acceptance_rate()`	Acceptance rate of the moves

SequenceMover and RepeatMover	

`seqmover = SequenceMover()`
`seqmover.addmover(smallmover)`
`seqmover.addmover(shearmover)`
`seqmover.addmover(minmover)`
	Creates a mover which will call a series of other movers in sequence.

`repeatmover = RepeatMover(fragmover,10)`	Creates a mover that will call the fragmover 10 times

`randommover = RandomMover()`
`randmover.addmover(smallmover)`
`randmover.addmover(shearmover)`
`randmover.addmover(minmover)`
	Creates a mover which will randomly apply one of a set of movers each time it is applied.

Rigid Body movers	

`pert_mover =`
`RigidBodyPerturbMover(jump_num,3,8)`
`pert_mover.apply(pose)`
	Makes a random rigid body move of the downstream partner. Random rotation chosen from a Gaussian of standard deviation of 8°, and translation chosen from a Gaussian of standard deviation 3 Å

`transmover = RigidBodyTransMover(pose,`
`jump_num)`
`transmover.trans_axis(a)`
`transmover.step_size(5)`
`transmover.apply(pose)`
	Creates a mover that will translate two partners, defined by jump_num, along an axis defined by numeric.xyzVector a, by 5 Angstroms.

`spinmover =`	Creates a mover that will spin partner 2 relative

RigidBodySpinMover(jump_num) **spinmover.spin_axis(a)** **spinmover.rot_center(b)** **spinmover.angle_size(45)**	to partner1, defined by jump_num, according to a spin axis and rotation center defined by numeric.xyzVectors a and b respectively, by 45 degrees. No specified angle_size randomizes the spin.

Sidechain Packing Movers	
pack_mover = **PackRotamersMover(scorefxn, task)** **pack_mover.apply(pose)**	Creates a mover that will use instructions from the 'task' to do packing to optimize side chain conformations in the pose
rot_trial = RotamerTrials(scorefxn, **task)** **rot_trial.apply(pose)**	Creates a mover that will use instructions from the 'task' to do Rotamer Trials to optimize side chain conformations in the pose
task = standard_packer_task(pose)	Creates a packer task based on a pose
task.or_include_current(True)	Includes current rotamers in pose to packer
task.restrict_to_repacking()	Restricts all residues to repacking
task.temporarily_fix_everything()	Sets all residues to no repacking
task.temporarily_set_pack_residue(i)	Sets residue i to allow repacking
task.read_resfile("resfile")	Sets task based on instructions in resfile
generate_resfile_from_pdb(test.pdb, **"resfile")** **generate_resfile_from_pose(pose,** **"resfile")**	Generates a resfile from a pdb file or a pose, respectively

TaskFactory TaskOperations	
tf = standard_task_factory()	Creates a default TaskFactory
Tf.create_task_and_apply_taskoperations(pose)	Creates a task based on the list of TaskOperations
tf.push_back(IncludeCurrent())	includes the current rotamers in the Pose to the rotamer sets used for packing. Defaulted on
tf.push_back(ReadResFile("test.resfile"))	applies instructions from the resfile when creating a PackerTask
tf.push_back(NoRepackDisulfides())	holds disulfide bond cysteine side-chains fixed. Defaulted on.
tf.push_back(RestrictToInterface(1))	allows repacking only at the interface defined by the jump number (in example jump# 1)
pr = PreventRepacking() **pr.include_residue(5)** **tf.push_back(pr)**	turns off repacking for specified residues (residue #5 in example)
rr = RestrictResidueToRepacking() **rr.include_residue(5)** tf.push_back(rr)	turns on repacking for specified residues (residue #5 in example)

Docking Movers	
DockingProtocol()	Protocol for a full, multiscale docking run
DockingProtocol().setup_foldtree(pose) **DockingProtocol().setup_foldtree(pose,'HL_A')**	Sets up a fold tree for docking, based on chain labels in the pose
movemap = MoveMap() **movemap.set_jump(jump_num,True)**	Sets up a mover to minimize over the rigid-body coordinates

`minmover = MinMover()` `minmover.movemap(movemap)`	
`dock_lowres = DockingLowRes(scorefxn_low, jump_num)` `dock_lowres.apply(pose)`	low-resolution, centroid based MC search (50 RigidBodyPerturbMoves with adaptable step sizes)
`dock_hires = DockingHighRes(scorefxn_high, jump_num)` `dock_hires.apply(pose)`	high-resolution, all-atom based MCM search with rigid-body moves, side-chain packing, and minimization
`cs = ConformerSwitchMover(start,end, jump_num,scorefxn,"1aaa.pdb")` `cs.apply(pose)`	Picks a new backbone conformation from the ensemble (conformer selection docking). **`start`** and **`end`** indicate residue number range for backbone swapping.
`randomize1 = RigidBodyRandomizeMover(pose, jump_num, partner_upstream)`	When applied, globally randomizes the rotation of the upstream partner.
`randomize2 = RigidBodyRandomizeMover(pose, jump_num, partner_downstream)`	When applied, globally randomizes the rotation of the downstream partner.
`DockingProtocol().calc_Lrmsd(pose1, pose2)`	Calculates RMSD of smaller partner after superposition of larger partner

Job Distributor	
`jd = PyJobDistributor("output", 1000, scorefxn_high)`	Creates a job distributor which will create 1000 model structures named **`output_1.pdb`** to **`output1000.pdb`**. Files include scorefxn_high energies.
`Pose native_pose("1aaa.pdb")` `jd.native_pose = native_pose`	Sets the native pose (loaded from **`1aaa.pdb`**) for rmsd comparisons
`jd.job_complete`	Boolean indicating whether all decoys have been output.
`jd.output_decoy(pose)`	Outputs the pose to a file and increments the decoy number.

`while (jd.job_complete == False):`
`  #[create the decoy called pose]`
`  jd.output_decoy(pose)`
	Loop to create decoys until all have been output

`jd.additional_info = "Created by Andy"`	Sets a string to be output to the pdb file

RMSD	
`print CA_rmsd(pose1, pose2)`	calculates and prints the root-mean-squared deviation of the location of Cα atoms between the two poses

Fold Tree	
`ft = FoldTree()` `ft = pose.fold_tree()`	Extracts the current fold tree from the pose
`pose.fold_tree(ft)`	Attaches the fold tree `ft` into the pose.
`ft.add_edge(1,30,-1)`	Creates a peptide edge (code -1) from residues 1 to 30. This edge will build N-to-C.
`ft.add_edge(100,31,-1)`	Creates a peptide edge from residues 100 to 31. This edge will build C-to-N.
`ft.add_edge(30,100,1)`	Creates a jump (rigid-body connection) between

	residues 30 and 100.
`ft.add_edge(100,101,2)`	Creates a second jump between residues 100 and 101 (note the jump number is 2. Each jump needs a unique, sequential jump number).
`ft.check_fold_tree()`	Returns **`True`** only for valid trees.
`print ft`	Prints the fold tree
`ft.simple_tree(100)`	Creates a single-peptide-edge tree for a 100-residue protein
`ft.new_jump(40,60,50)`	Creates a jump from residues 40 to 60, a cutpoint between 50 and 51, and splits up the original edges as needed to finish the tree.
`ft.clear()`	Deletes all edges in the fold tree.

Loops	
`loop1 = Loop(15,24,20)`	Defines a loop with stems at residues 15 and 24, and a cut point at residue 20
`loops = Loops()` `loops.add_loop(loop1)`	Creates an object to contain a set of loops
`set_single_loop_fold_tree(pose, loop)`	Sets the pose's fold tree for single-loop optimization
`ccd =` `CcdLoopClosureMover(loop1,movemap)`	Creates a mover which performs Canutescu & Dunbrack's cyclic coordinate descent loop closure algorithm
`loop_refine =` `LoopMover_Refine_CCD(loops)`	Creates a high-resolution refinement protocol consisting of cycles of small and shear moves, side-chain packing, CCD loop closure, and minimization.
`Lrms = loop_rmsd(pose,reference_pose,` `loops, True)`	Calculates the rmsd of all loops in the reference frame of the fixed protein structure

References and Further Reading