prospr

Release 1.2.19

Okke van Eck

Apr 06, 2026

CONTENTS:

Release v1.2.19.

Welcome to Prospr's documentation! Prospr is a Python toolbox for protein structure prediction, build on a C++ core. The Python package can be used when quick development is of key, and when Prospr's datasets and visualization module are of interest. The C++ core is available as a standalone for all high-performance computing applications. Ideally, the C++ core is used for time efficient data gathering, while the Python package is used for managing experiments. However, the Python package wraps all the C++ core's functionality, hence one can solely rely on using the Python package.

See the *Installation* and *Quickstart* pages for an easy introduction to the basics of Prospr. There is also the *API Reference* where all functionality of Prospr is explained in-depth per module. It is recommended to first take a look at the *Quickstart* before consulting the *API Reference*.

The source code of Prospr is publicly available on GitHub. All code is licenced under the *LGPL V3 license*.

CHAPTER

ONE

INSTALLATION

One can choose to use the Python package, or only the C++ core. The C++ core is handy for maximizing the execution speed of your experiments, while the Python package may lower development time and offers more functionality. Installation instructions are provided for both scenarios.

1.1 Python package

Prospr offers support for Python 3.9 and newer. However, using the latest Python version is always recommended. The instructions below assume that a new project will be set up.

1.1.1 Virtual environments

It is recommended to use a new virtual environment for each Python project. An easy way to setup a new virtual environment is via the **venv** module that comes with Python 3. After creation, one must activate the virtual environment before installing any packages. Examples for Linux/MacOS and Windows are given below.

For Linux/MacOS:

```
$ python3 -m venv venv
$ source venv/bin/activate
```

For Windows:

```
> py -3 -m venv venv
> venv/Scripts/activate
```

1.1.2 Installing Prospr (Python)

After activating the environment, use pip to install Prospr:

```
$ pip install prospr
```

Congratulations! Prospr is now installed. Check out the *Quickstart* to see how to use the basics, or read the *API Reference* for all functionality.

1.2 C++ core

Using the C++ core for your project is very easy. Follow the installation steps below and include the header files you want to use in your code.

1.2.1 Installing Prospr (C++)

Using the C++ core is very easy. Download a *prospr_core* archive from the archives folder on GitHub, then drag the source files to your code directory. Add the files to your Makefile in order to compile the Prospr files with your project. No additional libraries need to be linked during compile time.

Congratulations! Prospr is now installed. Check out the *Quickstart* to see how to use the basics, or read the *API Reference* for all functionality.

CHAPTER

TWO

QUICKSTART

One can choose to use the Python package, or only the C++ core. The C++ core is handy for maximizing the execution speed of your experiments, while the Python package may lower development time and offers more functionality.

Example usage is given using the Python package and only shows basic usage. Please refer to the *API Reference* to see the equivalent usage in C++ and all functionality.

Please take a look at the *Installation* page first to see how Prospr can be installed.

2.1 Creating Proteins

After installing Prospr, creating Protein objects is quite easy. A Protein object can be seen as a space manager for AminoAcid objects. The AminoAcid objects store the internal linkage, their index in the protein sequence, and their type. When generating a Protein object, the required AminoAcid objects are created and linked automatically as well.

One can simply create an HP-model Protein object as follows:

```
from prospr import Protein

p_2d = Protein("HPPHPPH")
p_3d = Protein("HPPHPPH", dim=3)
p_4d = Protein("HPPHPPH", dim=4)
p_5d = Protein("HPPHPPH", dim=5)
...
```

where *dim* is the dimension to fold in.

The *model* parameter allows for selecting different models in the future. So far, only the HP and HPXN models are supported. You can create a HPXN-model protein as follows:

```
from prospr import Protein

p_2d_hpxn = Protein("HPNPPNH", model="HPXN")
...
```

Custom models are also possible by providing a dictionary mapping the possible ways to bond. The dictionary should map strings to integers, where the strings are two characters identifying the amino acid types that can bond, and the integer the stability value of that bond. As an example, this is a redefinition of the HPXN-model:

```
from prospr import Protein

bond_values = {"HH": -4, "PP": -1, "PN": -1, "NN": 1}
p_2d_HP = Protein("HPNPPNH", dim=2, bond_values=bond_values)
```

Note that the inverse of the bonds (e.g. *NP* from the *PN* bond) are added automatically. If you **do not** want this, disable *bond_symmetry* through the parameter:

```
from prospr import Protein

bond_values = {"HH": -4, "PP": -1, "PN": -1, "NN": 1}
p_2d_HP = Protein("HPNPPNH", dim=2, bond_values=bond_values, bond_symmetry=False)
```

2.2 Protein attributes

A Protein object keeps track of multiple properties while it is being folded. These properties can be checked as attributes of the Protein object. All properties are listed below and speak for themselves, but please refer to the *API Reference* to see their exact descriptions.

```
from prospr import Protein

p_2d = Protein("HPPH")

p_2d.sequence
>>> "HPPH"

p_2d.cur_len
>>> 0

p_2d.dim
>>> 2

p_2d.last_move
>>> 0

p_2d.last_pos
>>> [0, 0]

p_2d.score
>>> 0

p_2d.solutions_checked
>>> 0

p_2d.aminos_placed
>>> 1

p_2d.bond_values
>>> {"HH": -1}

p_2d.max_weights
>>> [-1, 0, 0, -1]
```

2.3 Placing amino acids

A Protein object is generated with the first amino acid fixed at the origin. One can place the next amino acid via the *.place_amino(move)* function. This function takes a move as an argument, which is a number representing the axis to move over. As an example, 1 can be seen as the x-axis, 2 as the y-axis, etc. Negative numbers represent movement in negative direction.

```
...
p_2d.place_amino(1)
p_2d.place_amino(2)
p_2d.place_amino(-1)
```

2.4 Removing amino acids

Amino acids can be removed via the *.remove_amino()* function.

```
...
p_2d.remove_amino() # Leaving the previous moves [1, 2]
p_2d.remove_amino() # Leaving the previous move [1]
```

2.5 Validating moves

Before trying to place an amino acid, it it is recommended to check whether the move is valid. This can be done via the *.is_valid(move)* function, which takes the requested move as an argument.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.is_valid(1)
>>> True

p_2d.place_amino(1)
p_2d.is_valid(-1)
>>> False
```

2.6 Placement information

While writing algorithms, it might be necessary to check what amino acid is placed at a specific spot, or where the previous and next ones are placed. This can be checked via the *.get_amino(position)* function, which takes a list of integers representing the requested position as an argument. It returns an AminoAcid object, which has its *type*, *index*, *prev_move*, and *next_move* as attributes.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.place_amino(1)
p_2d.place_amino(2)
```

(continues on next page)

(continued from previous page)

```
amino = p_2d.get_amino([1, 0])

amino.type
>>> "P"

amino.index
>>> 1

amino.prev_move
>>> -1

amino.next_move
>>> 2
```

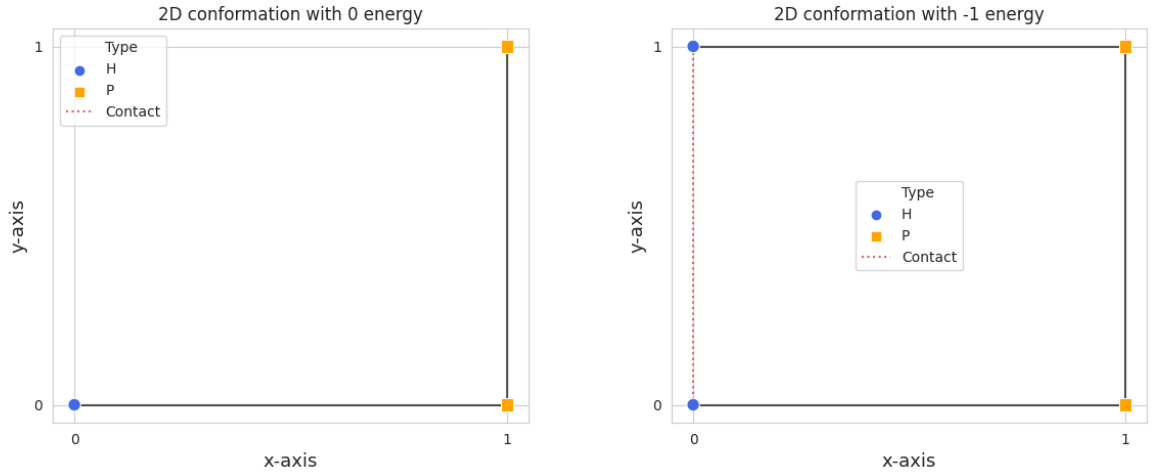
It might also occur that you want to check if an amino acid at a specific index can create bonds. This can be checked via the *.is_weighted(index)* function, which takes the index of the requested amino acid as an argument.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.is_weighted(0)
>>> True

p_2d.is_weighted(1)
>>> False
```

2.7 Checking stability

The stability of a (partially) folded Protein is tracked in the *.score* attribute. This attribute changes dynamically when placing and removing amino acids.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.place_amino(1)
p_2d.place_amino(2)
p_2d.score
>>> 0

p_2d.place_amino(-1)
p_2d.score
>>> -1
```



2.8 Checking number of changes

In order to compare the efficiency of algorithms, a Protein object also keeps track of the number of moves performed thus far. This does not include the removal of amino acids. The current number of performed moves is tracked in the *.changes* attribute.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.place_amino(1)
p_2d.changes
>>> 1

p_2d.place_amino(2)
p_2d.remove_amino()
p_2d.place_amino(-2)
p_2d.place_amino(-1)
p_2d.changes
>>> 4
```

2.9 Hashing folds

The current fold of a Protein can be generated via the *.hash_fold()* function. The function will return the sequence of moves for the current conformation.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.place_amino(1)
p_2d.hash_fold()
>>> [1]

p_2d.place_amino(2)
```

(continues on next page)

(continued from previous page)

```
p_2d.place_amino(-1)
p_2d.hash_fold()
>>> [1, 2, -1]
```

2.10 Setting folds

At any time, a Protein's conformation can be set to a given set of moves. This is done via the *.set_hash(fold_hash)* function, which takes a sequence of moves as an argument, just like the ones generated by the *.hash_fold()* function.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.place_amino(2)
p_2d.place_amino(-1)
p_2d.hash_fold()
>>> [2, -1]

p_2d.set_hash([1, 2, -1])
p_2d.hash_fold()
>>> [1, 2, -1]
```

2.11 Resetting Proteins

Sometimes you might want to reset a Protein object. This can be because you want to reuse the same Protein object, or because you want to clear the conformation. Each of these scenarios has their own function.

In order to reset the whole Protein object, use the *.reset()* function.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.place_amino(1)
p_2d.place_amino(2)
p_2d.place_amino(-1)
p_2d.changes
>>> 3

p_2d.hash_fold()
>>> [1, 2, -1]

p_2d.reset()
p_2d.changes
>>> 0

p_2d.hash_fold()
>>> []
```

Use the *.reset_conformation()* function to only reset the placement of the amino acids. This includes setting the *.score* to 0, as only the amino acid in the origin remains in place.

```
from prospr import Protein

p_2d = Protein("HPPH")
p_2d.place_amino(1)
p_2d.place_amino(2)
p_2d.place_amino(-1)
p_2d.changes
>>> 3

p_2d.hash_fold()
>>> [1, 2, -1]

p_2d.reset_conformation()
p_2d.changes
>>> 3

p_2d.hash_fold()
>>> []
```

2.12 Built-in algorithms

Prospr offers some algorithms for folding Proteins. These are included in the C++ core, making them time efficient relative to Python alternatives. The *API Reference* contain a list of all available built-in algorithms. They can all be easily used via a direct import, as is shown below.

```
from prospr import Protein, depth_first

p_2d = Protein("HPPH")
p_2d = depth_first(p_2d)
p_2d.score
>>> 1

p_2d.hash_fold()
>> [1, 2, -1]
```

2.13 Checkpoints

The algorithm *depth_first_bnb(protein)* supports checkpoints to resume an interrupted search by storing the state of the protein and the algorithm to a file, after a signal (*SIGTERM* or *SIGINT*) is received.

```
import os

from prospr import Protein, depth_first_bnb

os.environ["PROSPR_CACHE_DIR"] = "/tmp/prospr"

p_2d = Protein("HPPH")
# Will read/write /tmp/prospr/depth_first_bnb/HPPH.checkpoint
depth_first_bnb(p_2d)
```

(continues on next page)

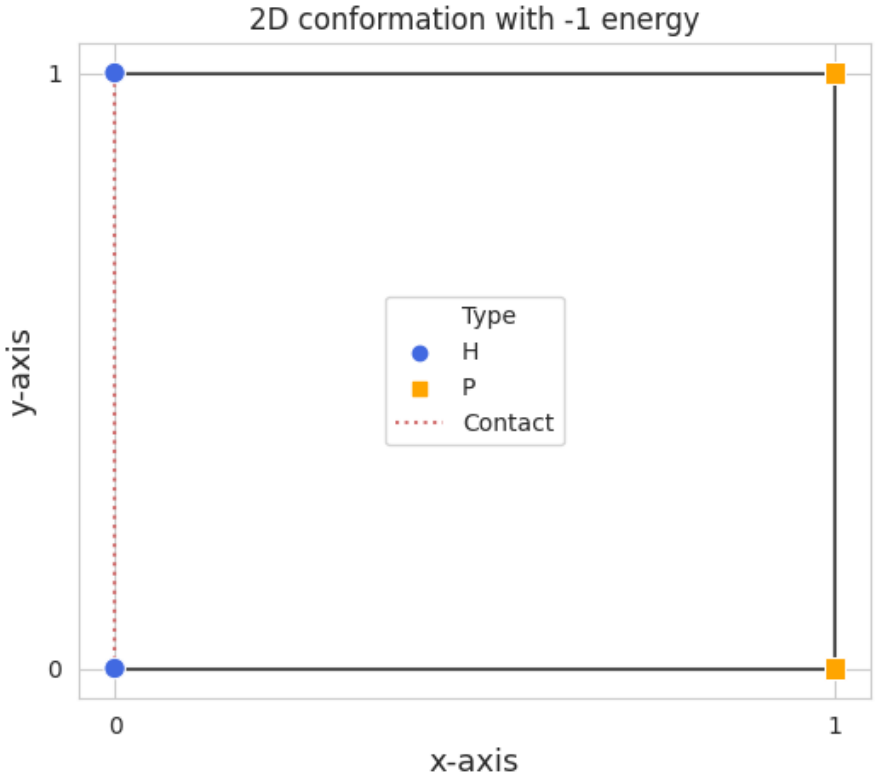
(continued from previous page)

```
print("Done.")
>>>
```

2.14 Visualizing conformations

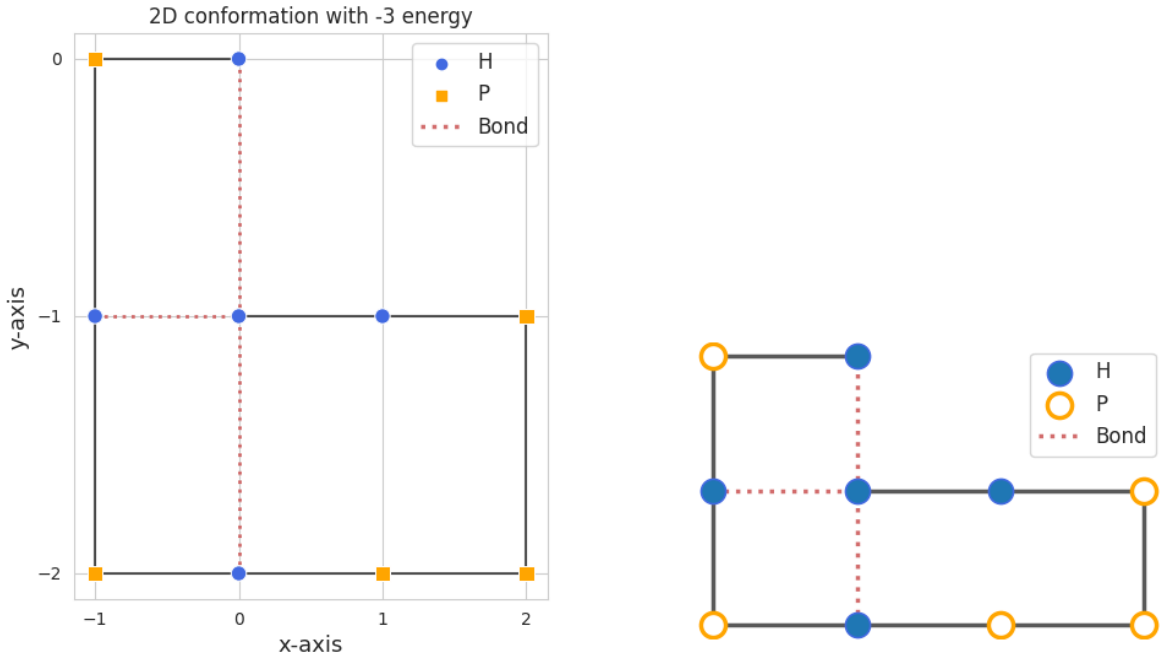
Visualizing conformations can be key to understanding how the resulting conformation was found. It also helps illustrating your research. Prospr's Python package has a built-in visualization module so you do not have to write your own. The module automatically detects the dimension and plots accordingly. Visualizing a conformation can easily be done via the *plot_protein()* function from the *prospr.visualize* module.

```
from prospr import Protein
from prospr.visualize import plot_protein

p_2d = Protein("HPPH")
p_2d.place_amino(1)
p_2d.place_amino(2)
p_2d.place_amino(-1)
plot_protein(p_2d)
>>>
```



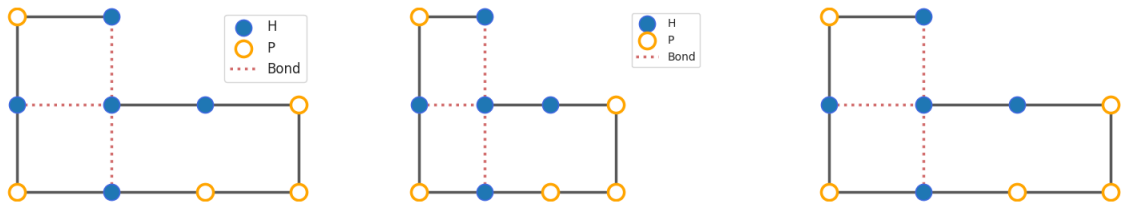
The *plot_protein()* function has a couple parameters to style the figure to your likings. Most importantly, there are two styles to select: *basic* and *paper*. The first will show the protein in a clear and zoomed-in way, while the latter is more compact and fancy. Here you can see the difference between the two for the same conformation.

```
p_2d = Protein("HPHPHPPPHH")
depth_first_bnb(p_2d)
plot_protein(p_2d)
>>>

plot_protein(p_2d, style="paper")
>>>
```



Besides the style, it is also possible to change the positioning of the legend. You can turn the legend off through the *legend* parameter, or change its position to be *inner* or *outer* via the *legend_style* parameter.

```
plot_protein(p_2d, style="paper", legend_style="inner")
>>>

plot_protein(p_2d, style="paper", legend_style="outer")
>>>

plot_protein(p_2d, style="paper", legend=False)
>>>
```



There are also some parameters that alter the style of the figure. Please refer to the *API Reference* for a full overview.

2.15 Using datasets

Datasets are valuable for a fair comparison between algorithms. That is why Prospr's Python package comes with a built-in collection of datasets. Loading a dataset can easily be done via the available load functions in the *prospr.datasets* module. Currently, there are three datasets available: *vanEck250*, *vanEck1000*, and *vanEck_hratio*.

The *vanEck1000* dataset contains 1000 unique proteins for lengths [10, 15, 20, .., 100], where the chances of sampling a H or P are equal. *vanEck250* offers a subset of *vanEck1000*, by simply offering the first 250 proteins for each length. *vanEck_hratio* has around 1000 proteins sampled for each of the H-ratio intervals {(0.0, 0.1), [0.1, 0.2), [0.2, 0.3), .., [0.9, 1.0)}. You can find their usage below, as well as in the *API Reference*.

```
from prospr.datasets import load_vanEck250, load_vanEck1000, load_vanEck_hratio

length_10 = load_vanEck250()
length_15 = load_vanEck250(15)
length_20 = load_vanEck250(20)
len(length_20)
>>> 250

length_20 = load_vanEck1000(20)
len(length_20)
>>> 1000

length_25_hratio_01 = load_vanEck_hratio()
length_10_hratio_04 = load_vanEck_hratio(10, 0.4)
length_15_hratio_06 = load_vanEck_hratio(length=15, hratio=0.6)
```

CHAPTER

THREE

API REFERENCE

The sections beneath contain all functions that can be used, ordered by module. The modules that are part of the C++ core are indicated by their section title. All functions have the same signature in the Python package and the C++ core. Only the properties of the Protein class are called differently. All API references are given in alphabetical order.

3.1 Algorithms - core

Prospr contains a couple of built-in search algorithms that will find the most optimal conformation. These can all be imported directly from *prospr* without specifying a submodule, e.g.

```
from prospr import depth_first
```

depth_first(*protein*)

Finds the most optimal conformation using a depth-first algorithm.

Does not reset the Protein properties beforehand!

Parameters:

* **protein** - *Protein*: the Protein object to fold.

Returns:

* **Protein** - the Protein object set at the found conformation and with updated properties according to the performed moves.

depth_first_bnb(*protein*)

Finds the most optimal conformation using a depth-first branch-and-bound algorithm.

Does not reset the Protein properties beforehand!

Parameters:

* **protein** - *Protein*: the Protein object to fold.

Returns:

* **Protein** - the Protein object set at the found conformation and with updated properties according to the performed moves.

beam_search(*protein, beam_width=-1*)

Finds a best-effort conformation using a beam search algorithm.

Does not reset the Protein properties beforehand!

Parameters:

* **protein** - *Protein*: the Protein object to fold.

* **beam_width** - *int (optional)*: the beam width to use, where -1 indicates traversal of the entire search space.

Returns:

* **Protein** - the Protein object set at the found conformation and with updated properties according to the performed moves.

3.2 AminoAcid - core

Protein objects use AminoAcid objects internally to keep track of the chain on the grid. AminoAcid objects have read-only properties, which are consulted internally by the Protein object. The AminoAcid class can be imported directly from *prospr* without specifying a submodule, e.g.

```
from prospr import AminoAcid
```

3.2.1 AminoAcid Properties

When using the Python package, each property can be directly called as an attribute. If the C++ core is used, the property can be accessed using a method. Each property is described below with the Python and C++ syntax for accessing them.

type

Type of the AminoAcid.

Python	*.type*
C++	*.get_type()*
Return type	*str*

index

Index of the AminoAcid within the Protein's sequence.

Python	*.index*
C++	*.get_index()*
Return type	*int*

prev_move

Move to perform in order to get to the previous AminoAcid in the chain.

Python	*.prev_move*
C++	*.get_prev_move()*
Return type	*int*

next_move

Move to perform in order to get to the next AminoAcid in the chain.

Python	*.next_move*
C++	*.get_next_move()*
Return type	*int*

3.3 Datasets

Prospr provides many datasets. Each can be imported from the *prospr.datasets* submodule, e.g.

```
from prospr.datasets import load_vanEck250
```

load_vanEck250(*length=10*)

Loads the vanEck250 dataset containing 250 proteins per length, with lengths from 10 till 100.

Parameters:

* **length** - *int (optional)*: the length of the protein sequences to load.

Returns:

* **DataFrame** - a Pandas DataFrame containing the protein sequences.

load_vanEck1000(*length=10*)

Loads the vanEck1000 dataset containing 1000 proteins per length, with lengths from 10 till 100.

Parameters:

* **length** - *int (optional)*: the length of the protein sequences to load.

Returns:

* **DataFrame** - a Pandas DataFrame containing the protein sequences.

load_vanEck_hratio(*length=25, hratio=0.1*)

Loads the vanEck_hratio dataset containing proteins per length and hratio combination.

Parameters:

* **length** - *int (optional)*: the length of the protein sequences to load.
* **hratio** - *float (optional)*: the hratio upperbound of the hratio interval to use.

Returns:

* **DataFrame** - a Pandas DataFrame containing the protein sequences.

3.4 Helpers

The helpers submodule contains functions used internally. These functions are very specific for their internal use, but maybe of use to you as well. Each can be imported from the *prospr.helpers* submodule, e.g.

```
from prospr.helpers import get_ordered_positions
```

get_ordered_positions(*protein*)

Returns a Numpy ndarray with the positions of all placed amino acids in order of placement.

Parameters:

* **protein** - *Protein*: the Protein object to fetch the positions from.

Returns:

* **ndarray** - a Numpy ndarray with the positions and types of all placed amino acids and in order of placement.
E.g. *[[[0, 0], "H"], [[0, 1], "P"], [[[1, 1], "P"], [[[1, 0], "H"]]*

get_scoring_aminos(*protein*)

Returns a dictionary with the positions of all placed amino acids from the given protein that may score points.

Parameters:

* **protein** - *Protein*: the Protein object to compute the scoring amino acids from.

Returns:

* **dict** - a dictionary mapping the locations of all amino acids that may score points to their previous move and next move.
E.g. *{[1, 0]: [-1, 2]}*

get_scoring_pairs(*protein*)

Returns a Numpy ndarray with arrays containing tuples of the positions of amino acid pairs from the given protein that score points.

Parameters:

* **protein** - *Protein*: the Protein object to compute the scoring pairs from.

Returns:

* **ndarray** - a Numpy ndarray with arrays containing tuples of the positions of amino acid pairs from the given protein that score points.
E.g. *[[(0, 0), (0, 1)], [(0, 0), (-1, 0)]]*

export_protein(*protein, path*)

Save conformation of a protein in Protein Data Bank (PDB) file format for processing or visualization with external software such as Mol*.

Parameters:

* **protein** - *Protein*: Protein object to save the hash of.
* **path** - *os.PathLike or str*: The path of the output file.

Returns:

* **None**

3.5 Protein - core

The Protein class is the cornerstone of Prospr. An instance tracks many properties and has methods to alter the Protein's state. First, an overview will be made of all the properties and how to reach them. Second, all the non-property bounded methods will be given. The Protein class can be imported directly from *prospr* without specifying a submodule, e.g.

```
from prospr import Protein
```

3.5.1 Protein Properties

When using the Python package, each property can be directly called as an attribute. If the C++ core is used, the property can be accessed using a method. Each property is described below with the Python and C++ syntax for accessing them.

bond_values

The ways to form bonds and their stability.

Python	*.bond_values*
C++	*.get_bond_values()*
Return type	*Dict[str, int]*

changes

The number of amino acids placed so far.

Python	*.changes*
C++	*.get_changes()*
Return type	*int*

cur_len

The length of the current conformation.

Python	*.cur_len*
C++	*.get_cur_len()*
Return type	*int*

dim

The maximum dimension in which the Protein can fold.

Python	*.dim*
C++	*.get_dim()*
Return type	*int*

h_idxs

The indexes of the "H" amino acids in the sequence.

Python	*.h_idxs*
C++	*.get_h_idxs()*
Return type	*List[int]*

last_move

The last performed move.

Python	*.last_move*
C++	*.get_last_move()*
Return type	*List[int]*

last_pos

The position of the amimo acid at the end of the current conformation.

Python	*.last_pos*
C++	*.get_last_pos()*
Return type	*List[int]*

max_weigths

For each amino acid, the maximum value a bond can make.

Python	*.max_weigths*
C++	*.get_max_weigths()*
Return type	*List[int]*

score

The score of the current conformation.

Python	*.score*
C++	*.get_score()*
Return type	*int*

sequence

The amino acid sequence of the Protein.

Python	*.sequence*
C++	*.get_sequence()*
Return type	*str*

3.5.2 Methods

The Protein class knows many methods to interact with a protein. The Python package and C++ core use the same signatures, so no distinction is made in the references below.

.get_amino(*position*)

Returns a list with the amino acid index and next move of the amino acid placed at the given position.

Parameters:

* **position** - *List[int]*: position of the amino acid.

Returns:

* **List[int]** - a list with the amino acid's index and next move.

E.g. *[0, 1]*

.get_bonds()

Returns a list of amino acid index pairs that are bonding.

Parameters:

* **None**

Returns:

* **List[Tuple[int,int]]** - a list of tuples with two amino acid indexes that bond.

E.g. *[(0, 9), (2, 9), (9, 2), (9, 0)]*

.hash_fold()

Returns a list of moves representing the current conformation.

Parameters:

* **None**

Returns:

* **List[int]** - a list of moves.

E.g. *[1, 2, -1]*

.is_hydro(*index*)

Returns if the amino acid at the given index is an H.

Parameters:

* **index** - *int*: index of the amino acid.

Returns:

* **bool** - A boolean indicating if the amino acid is an H.

.is_valid(*move*)

Returns if the given move is a valid next move.

Parameters:

* **move** - *int*: possible next move to perform.

Returns:

* **bool** - A boolean indicating if the given move is valid.

.place_amino(*move, track=True*)

Places the next amino acid in the given direction.

Parameters:

* **move** - *int*: direction to place the next amino acid.
* **track** - *bool (optional)*: set to True if the move should be tracked as a change.

Returns:

* **None**

.remove_amino()

Removes the previously placed amino acid.

Parameters:

* **None**

Returns:

* **None**

.reset()

Reset the whole Protein as if it was just created.

Parameters:

* **None**

Returns:

* **None**

.reset_conformation()

Reset the placement of amino acids for the given Protein. Also sets the *.score* property to 0.

Parameters:

* **None**

Returns:

* **None**

.set_hash(*fold_hash*)

Set the conformation to the given sequence of moves.

Parameters:

* **fold_hash** - *List[int]*: a list of moves as provided by **.hash_fold()**.

Returns:

* **None**

3.6 Visualize

Functions from the visualize module of Prospr can be used to illustrate your research. Each function can be imported from the *prospr.visualize* submodule, e.g.

```
from prospr.visualize import plot_protein
```

plot_protein(*protein, style="basic", ax=None, legend=True, legend_style="inner", show=True, linewidth=2.5, markersize=210, annotate_first=False*)

Plots the current set conformation of the given Protein object.

Parameters:

* **protein** - *Protein*: a Protein object to plot the conformation of.
* **style** - *str (optional)*: The figure style to use, either 'basic' or 'paper'.
* **ax** - *Axes (optional)*: If given, plot the conformation on the given Matplotlib Axes.
* **legend** - *bool (optional)*: Set to False to disable the legend.
* **legend_style** - *str (optional)*: The legend style to use, either 'inner' or 'outer'.
* **show** - *bool (optional)*: Set to False to disable plt.show() call.
* **linewidth** - *float (optional)*: Line width of the chain.
* **markersize** - *float (optional)*: Size of the amino acids.
* **annotate_first** - *float (optional)*: Set to True to highlight first amino acid with a color.

Returns:

* **None**

CHAPTER

FOUR

LICENSE

```
                    GNU LESSER GENERAL PUBLIC LICENSE
                        Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.


  This version of the GNU Lesser General Public License incorporates
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

  0. Additional Definitions.

  As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

  "The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

  An "Application" is any work that makes use of an interface provided
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

  A "Combined Work" is a work produced by combining or linking an
Application with the Library.  The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

  The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.

  The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.
```

(continues on next page)

(continued from previous page)

```
  1. Exception to Section 3 of the GNU GPL.

  You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

  2. Conveying Modified Versions.

  If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:

   a) under this License, provided that you make a good faith effort to
   ensure that, in the event an Application does not supply the
   function or data, the facility still operates, and performs
   whatever part of its purpose remains meaningful, or

   b) under the GNU GPL, with none of the additional permissions of
   this License applicable to that copy.

  3. Object Code Incorporating Material from Library Header Files.

  The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the
   Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the object code with a copy of the GNU GPL and this license
   document.

  4. Combined Works.

  You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

   a) Give prominent notice with each copy of the Combined Work that
   the Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the Combined Work with a copy of the GNU GPL and this license
   document.
```

(continues on next page)

(continued from previous page)

```
   c) For a Combined Work that displays copyright notices during
   execution, include the copyright notice for the Library among
   these notices, as well as a reference directing the user to the
   copies of the GNU GPL and this license document.

   d) Do one of the following:

       0) Convey the Minimal Corresponding Source under the terms of this
       License, and the Corresponding Application Code in a form
       suitable for, and under terms that permit, the user to
       recombine or relink the Application with a modified version of
       the Linked Version to produce a modified Combined Work, in the
       manner specified by section 6 of the GNU GPL for conveying
       Corresponding Source.

       1) Use a suitable shared library mechanism for linking with the
       Library.  A suitable mechanism is one that (a) uses at run time
       a copy of the Library already present on the user's computer
       system, and (b) will operate properly with a modified version
       of the Library that is interface-compatible with the Linked
       Version.

   e) Provide Installation Information, but only if you would otherwise
   be required to provide such information under section 6 of the
   GNU GPL, and only to the extent that such information is
   necessary to install and execute a modified version of the
   Combined Work produced by recombining or relinking the
   Application with a modified version of the Linked Version. (If
   you use option 4d0, the Installation Information must accompany
   the Minimal Corresponding Source and Corresponding Application
   Code. If you use option 4d1, you must provide the Installation
   Information in the manner specified by section 6 of the GNU GPL
   for conveying Corresponding Source.)

  5. Combined Libraries.

  You may place library facilities that are a work based on the
Library side by side in a single library together with other library
facilities that are not Applications and are not covered by this
License, and convey such a combined library under terms of your
choice, if you do both of the following:

   a) Accompany the combined library with a copy of the same work based
   on the Library, uncombined with any other library facilities,
   conveyed under the terms of this License.

   b) Give prominent notice with the combined library that part of it
   is a work based on the Library, and explaining where to find the
   accompanying uncombined form of the same work.

  6. Revised Versions of the GNU Lesser General Public License.
```

(continues on next page)

(continued from previous page)

```
  The Free Software Foundation may publish revised and/or new versions
of the GNU Lesser General Public License from time to time. Such new
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

  Each version is given a distinguishing version number. If the
Library as you received it specifies that a certain numbered version
of the GNU Lesser General Public License "or any later version"
applies to it, you have the option of following the terms and
conditions either of that published version or of any later version
published by the Free Software Foundation. If the Library as you
received it does not specify a version number of the GNU Lesser
General Public License, you may choose any version of the GNU Lesser
General Public License ever published by the Free Software Foundation.

  If the Library as you received it specifies that a proxy can decide
whether future versions of the GNU Lesser General Public License shall
apply, that proxy's public statement of acceptance of any version is
permanent authorization for you to choose that version for the
Library.
```