PyHDX Documentation

Release v0.4.0b1+69.g4efb141.dirty

Jochem Smit

Jul 23, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

PyHDX is software to extract H/D exchange kinetics from HDX-MS data sets in terms of Gibbs free energy of exchange (ΔG) at the residue level.

An interactive web server is available where users can upload HDX-MS data and obtain ΔG values. Please refer to the GitHub page for up-to-date links to the web server. How to use the web app is described in *PyHDX web application*.

PyHDX analysis can also be ran headless from python scripts. The 'templates' directory on GitHub lists several examples. Further examples can be found in the section *Examples*.

For more information, please have a look at our manuscript on bioRxiv.

CHAPTER

TWO

INSTALLATION

2.1 Stable release (v0.3.2)

Installation with *conda*:

```
$ conda install -c conda-forge pyhdx
```

Installation with *pip*:

```
$ pip install pyhdx==0.3.2
```

2.2 Beta release (v0.4.0b1)

To install base PyHDX:

```
$ pip install pyhdx==0.4.0b1
```

To install with web application:

```
$ pip install pyhdx==0.4.0b1[web]
```

Currently custom bokeh extensions are not packaged, therefore to run the web application Node.js is required:

```
$ conda install nodejs
```

To install with pdf output:

```
$ pip install pyhdx==0.4.0b1[pdf]
```

2.3 Running the web server

PyHDX web application can be launched from the command line using `pyhdx` command with below options,

To run PyHDX server using default settings on your local server:

```
$ pyhdx serve
```

To run PyHDX server using the IP address and port number of your dask cluster:

```
$ pyhdx --cluster <ip> <port>
```

To start a dask cluster separately, open another terminal tab and run:

```
python local_cluster.py
```

This will start a Dask cluster on the scheduler address as specified in the PyHDX config. (user dir / .pyhdx folder)

2.4 Dependencies

The requirements for PyHDX are listed in setup.cfg

CHAPTER

THREE

PYHDX WEB APPLICATION

This section will describe a typical workflow of using the main web interface application. Detailed information on each parameter can be found in the web application reference docs *Web Application Reference*. The web application consists of a sidebar with controls and input, divided into sections, and a main view area with graphs and visualization. We will go through the functionality of the web interface per section.

3.1 Peptide Input

Use the *Browse* button to select peptide data files to upload. These should be 'peptide master tables' which is **long format** data where each entry should at least have the entries of:

- start (inclusive residue number at which the detected peptide starts, first residue = 1)
- stop (inclusive residue number at which the detected peptide stops)
- sequence (sequence of the peptide in one letter amino acid codes)
- exposure (time of exposure to deuterated solution)
- uptake (amount/mass (g/mol) of deuterium taken up by the peptide)
- state (identifier to which 'state' the peptide/protein is in (ie ligands, experimental conditions)

Currently the only data format accepted is exported 'state data' from Waters DynamX, which is .csv format. Exposure time units is assumed to be minutes. Other data format support can be added on request (eg HDExaminer).

Multiple files can be selected after which these files will be combined. Make sure there are no overlaps/clashes between 'state' entries when combining multiple files.

Choose which method of back-exchange correction to use. Options are either to use using a fully deuterated sample or to set a fixed back-exchange percentage for all peptides. The latter method should only be used if no FD sample is available. A percentage to set here can be obtained by running a back-exchange control once on your setup.

When selecting *FD Sample*, use the fields *FD State* and *FD Exposure* to choose which peptides from the input should be used as FD control. Note that these peptides will be matched to the ones in the experiment and peptides without control will not be included.

Use the fields *Experiment State* to choose the 'state' of your experiment. In 'Experiment Exposures' you can select which exposure times to add include the dataset.

In the *Drop first*` entry the number of N-terminal residues for each peptides can be chosen which should be ignored when calculating the maximum uptake for each peptide as they are considered to fully exchange back. Prolines are ignored by default as they do not have exchangeable amide hydrogens.

Next, specify the percentage of deuterium in the labelling solution as well as in the FD control labelling solution. There percentages should be as close as possible to eachother and as high as possible, typically >85%. Add the temperature (in Kelvin) and the pH at which the D-labelling was done. This is the pH as read from the pH meter without any correction.

The next fields *N term* and *C term* specify the residue number indices of the N-terminal and C-terminal residues, respectively. For the N-terminal this value is typically equal to 1, but if N-terminal affinity tags are used for purification this might be a negative number. The value specified should match with the residue indices used in in the input .csv file. The C-term value tells the software at which index the C-terminal of the protein is, as it is possible that the protein extends beyond the last residue included in any peptide and as the C-term exhibits different intrinsic rates of exchanges this needs to be taken into account. A sequence for the full protein (in the N-term to C-term range as specified) can be added to provide additional sequence information, but this is optional.

Finally, specify a name of the dataset, by default equal to the 'state' value and press 'Add dataset' to add the dataset. Datasets currently cannot be removed, if you want to remove datasets, press the browser 'refresh' button to start over.

3.2 Coverage

In the 'Coverage' figure in the main application area rectangles should show corresponding to the peptides of a single timepoint. Peptides are only included if they are in both all the timepoints as well as in the fully deuterated control sample. Under 'Coverage' in the sidebar, part of the coverage graph can be controlled, by specifying which color map to use. Controls for choosing which 'state' to display and which exposure time can be found under Graph Cont

By hovering the mouse over the peptides in the graph, more information is shown about each peptide:

- index: Index of the peptide per timepoint starting at the first peptide at 0
- rfu: Relative fraction uptake of the peptide (Absolute corrected D-uptake in brackets)
- sequence: FASTA sequence of the peptide. Non-exchanging N-terminal reisues marked as 'x' and prolines in lower case.

3.3 Initial Guesses

As a first step in the fitting procedure, initial guesses for the exchange kinetics need to be derived. This can be done through two options: 'Half-life' (fast but less accurate), or 'Association' (slower but more accurate). Using the 'Association' procedure is recommended. This model fit two time constants the the weighted-averaged uptake kinetics of each residue. A upper and lower bound of these rate constants can be specified but in most cases the autosuggested bounds are sufficient. Rarely issues might arise when the initial guess rates are close to the specified bounds at which point the bounds should be moved. This can be checked by comparing the fitted rates *k1* and *k2* (*File Export ▸ Target dataset ▸ rates*) Both rates are associated amplitudes are converted to a single rate value used for initial guesses. Select the model in the drop-down menu, assign a name to these initial guesses and the press 'Calculate Guesses'. The fitting is done in the background. When the fitting is done, the obtained rate is shown in the main area in the tab 'Rates'. Note that these rates are merely an guesstimate of HDX rates and these rates should not be used for any interpretation whatsoever but should only function to provide the global fit with initial guesses.

3.4 Fitting

After the initial guesses are calculated we can move on the the global fit of the data. Details of the fitting equation can be found the PyHDX publication (currently bioRxiv).

At 'Initial guess', select which dataset to use for initial guesses (typically 'Guess_1'). At 'Fit mode', users can choose either 'Batch' or 'Single' fitting. If only one datasets is loaded, only 'Single' is available. If 'Single' is selected, PyHDX will fit ΔG values for each datasets individually using the specified settings. In 'Batch' mode all data enters the fitting process at the same time. This allows for the use of a second regularizer between datasets. Note that when using 'Batch' mode, the relative magnitudes of the Mean Squared error losses and regularizer might be different, such that 'Batch' fitting with `r2` at zero is not identical to 'Single' fits.

The fields *Stop loss* and *Stop patience* control the fitting termination. If the loss improvement is less than *Stop loss* for *Stop patience* epochs (fit iterations), the fitting will terminate. *Learning rate* controls the step size per epoch. For typical a dataset with 62 peptides over 6 timepoints, the learning rate should be 50-100. Smaller datasets require larger learning rates and vice versa.

Momentum and *Nesterov* are advanced settings for the Pytorch `SGD` optimizer.

The maximum number of epochs or fit iterations is set in the field *Epochs*.

Finally, the fields *Regualizer 1* and *Regulizer 2* control the magnitude of the regualizers. Please refer to our bioRxiv manuscript for more details. In short, `r1` acts along consecutive residues and affects as a 'smoothing' along the primary structure. Higher values give a more smoothed result. This prevents overfitting or helps avoid problems in the 'non-identifiability' issue where in unresolved (no residue-level overlap) regions the correct kinetic components can be found (ΔGs of residues given correct choice of timepoints) but it cannot confidently be assigned to residues as resolution is lacking. The regualizer *r1* biases the fit result towards the residue assignment choice with the lowest variation along the primary structure. Typical values range from 0.01 to 0.5, depending on size of the input data.

r2 acts between samples, minimizing variability between them. This is used in differential HDX where users are interested in ΔG differences (ΔΔG). When measuring HD exchange with differing experimental conditions, such as differences in peptides detected, timepoints used or D-labelling temperature and pH, the datasets obtained will have different resolution, both 'spatially' (degree of resolved residues) and 'temporally' (range/accuracy of ΔGs). This can lead to artefactual differences in the final ΔΔG result, as features might be resolved in out dataset and not in the other, which will show up as ΔΔG.

Specify a unique name at *Fit name* and press *Do Fitting* do start the fit. The *Info log* in the bottom right corner displays information on when the fit started and finished. The fitting runs in the background and multiple jobs can be executed at the same time. However, please take into account that these fits are computationally intensive and currently if multiple users submit too many jobs it might overwhelm our/your server.

The output ΔG values are shown in the 'Gibbs' graph (bottom left).

See also the *Fitting example* section for more details on fitting and the effect of regualizers.

3.5 Graph Control

This section is used to control which dataset is currently show in the graphs. Use the selector *Fit id* to switch between fit results. The selector *State name* is used to switch between experimental states. In the current version (v0.4.0bx) the switching is a bit slow so please wait a few seconds before switching again.

Under *Coverage*, use the *Exposure* selector to choose which timepoint is shown in the the coverage graph. At *Peptide* and *Peptide index* the peptide to show at the Peptide (uptake curve) graph can be chosen.

We can use these control to inspect the quality of the fit obtained. First, at *Losses* (bottom right) the progress of the fit can be inspected. This should show a rapid decrease of the 'mse' loss, followed by a mostly flat plateau. If this is not the case, extend the number of epochs (*epochs* or *stop_loss* and *Stop patience*) or increase *Learning rate*.

The graph 'coverage mse' shows the total mean squared error of all timepoints per peptide. The color scale adjust automatically so red colors do not necessarily reflect a poor fit, but highligh the worst fitted peptides in your dataset. Hover over the peptide with the mouse to find the index of the peptide and select the peptide with *Peptide index*.

3.6 Classification

The classification value can be used to create color datasets based on results from the global fit. At *fit_ID*, choose which of the fit runs to use. Use *state_name* to choose which experimental states to apply the color map to, use '*' to select all states. Finally use *quantity* to select which output column to use (typically deltaG)

A preview of the color map will be applied to the ΔG values in the 'Gibbs' graph.

Mode can be used to select between the available color modes; *Discrete*, *Continuous* and *Color map*. *Discrete* splits the ΔG values in *n* categories, which are all assigned the same color. When using *Continuous*, *n* color 'nodes' can be defined, where color values are interpolated between these nodes. *Color map* allows users to choose a colormap from either `matplotlib` or `colorcet`.

The number of categories can be set with *Number of colours*. When using *Discrete* coloring, the thresholds of the categories can be automatically determined by pressing the *Otsu* button (using Otsu's method). Use the button *Linear* to distribute threshold values automatically with equal distances between them, and the extrema at the largest/smallest ΔG values.

Toggle *Log space* to apply the color map in log space (typically used for colouring rates/protection factors). Assign an unique name using *Color set name* and press *Add colorset* to create the color dataset.

A color for residues which are covered by peptides can be chosen at *No coverage*. The colors for the color groups or nodes can be chosen at the bottom of the controllers, as well as the exact position of the thresholds. These values must be input such that they are always in decreasing order.

3.7 Protein Viewer

Colorsets can be directly visualized on a protein structure using the built in NGL protein viewer. Use *Input mode* to choose whether to download a structure from the RCSB database or to use a local file. When selecting *RCSB*, enter the entry ID under *Rcsb id* and press *Load structure* to download the pdb file and display it. When selecting *PDB File* use the *Choose File* button to upload a .pdb file from your computer, and press *Load structure*

Select wich color map to display on the structure with *color ID*, and select the experimental state with *State name*

3.8 File Export

All tables which underlie the graphs in the PyHDX web application can be downloaded directly. Choose the the desired dataset ('global_fit' for ΔG values) with *Target dataset*. The data can be exported in machine-readable .csv files or human-readable .txt (pprint) file by setting *Export format*. Make sure to download at least the .csv file for further.

When selecting a color dataset, the data can not only be download as a .csv file but also as (a zip file of) .pml files which contain pymol scripts to directly apply the color map to a structure in pymol.

CHAPTER

FOUR

EXAMPLES

4.1 PyHDX basics

[3]:
```
from pyhdx import PeptideMasterTable, read_dynamx, HDXMeasurement
from pyhdx.plot import plot_peptides
import matplotlib.pyplot as plt
from pathlib import Path
```

We can use the `read_dynamx` function to read the file. This function returns a `numpy` structured array where each entry corresponds to one peptide, in this example 567 peptides.

[2]:
```
fpath = Path() / '..' / '..' / 'tests' / 'test_data' / 'ecSecB_apo.csv'
data = read_dynamx(fpath)
data.size
```

[2]:
```
567
```

This array is loaded into the `PeptideMasterTable` class, which is the main data entry class. The parameter `drop_first` determines how many N-terminal residues are considered to be fully back-exchanged, and therefore is subtracted from the total amount of exchangable D per peptide. The parameter `ignore_prolines` is controls whether the number of Prolines residues in the peptide should be subtracted from the total amount of exchangable and should generally be set to `True`.

The final number of exchangable residues is found in the 'ex_residues' field.

[3]:
```
master_table = PeptideMasterTable(data, drop_first=1, ignore_prolines=True)
master_table.data['ex_residues'][:50]
```

[3]:
```
array([ 8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  8.,  6.,  6.,  6.,  6.,
        6.,  6.,  6.,  6.,  6., 12., 12., 12., 12., 12., 12., 12., 12.,
       12., 13., 13., 13., 13., 13., 13., 13., 13., 13., 14., 14., 14.,
       14., 14., 14., 14., 14., 14., 20., 20., 20., 20., 20.])
```

This master table allows us to control how the deuterium uptake content is determined. The method `set_control` can be used to choose which set of peptides is used as the fully deuterated (FD) control. This adds a new field called 'uptake' which is the normalized (to 100%) deuterium uptake of each peptide, with respect to the total amount of exchanging residues.

[4]:
```
master_table.set_control(('Full deuteration control', 0.167))
master_table.data['uptake'][:50]
```

[4]:
```
array([ 0.      ,  0.      ,  5.0734  ,  2.486444,  2.857141,  3.145738,
        3.785886,  4.08295 ,  4.790625,  0.      ,  0.      ,  3.642506,
        1.651437,  1.860919,  2.107151,  2.698036,  2.874801,  3.449561,
```

(continues on next page)

(continued from previous page)

```
       0.      ,  0.      ,  5.264543,  1.839924,  2.508343,  2.969332,
       3.399092,  3.485568,  4.318144,  0.      ,  0.      ,  6.3179  ,
       2.532099,  3.306167,  3.996718,  4.38941 ,  4.379495,  5.283969,
       0.      ,  0.      ,  6.812215,  3.11985 ,  3.874881,  4.342807,
       4.854057,  4.835639,  5.780219,  0.      ,  0.      , 10.8151  ,
       5.432395,  6.1318  ])
```

Next we'll select our state of interest from the master Table. The available states are listed in `master_table.states`. Using `get_state` allows us to select all entries which belong to this state.

[5]:
```
master_table.states
state_data = master_table.get_state('SecB WT apo')
state_data.size
```

[5]:
```
441
```

This `data` array can now be used to create an `HDXMeasurement` object, the main data object in PyHDX. Experimental metadata such as labelling pH and temperature can be specified. These quantities are required for calculating intrinsic exchange rates and ΔG values. The pH values are uncorrected values are measured by the pH meter (ie p(H, D) values)

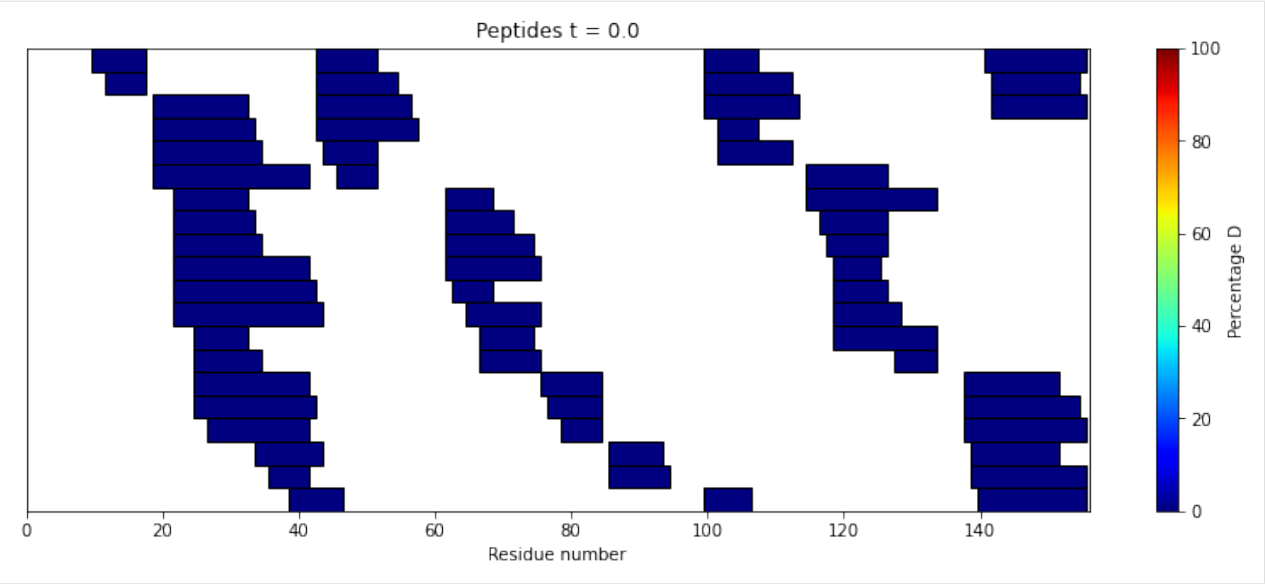
[6]:
```
hdxm = HDXMeasurement(state_data, temperature=303.15, pH=8., name='My HDX measurement
↪')
type(hdxm), len(hdxm), hdxm.timepoints, hdxm.name, hdxm.state
```

[6]:
```
(pyhdx.models.HDXMeasurement,
 7,
 array([  0.      ,   0.167   ,   0.5     ,   1.      ,   5.      ,
         10.      , 100.000008]),
 'My HDX measurement',
 'SecB WT apo')
```

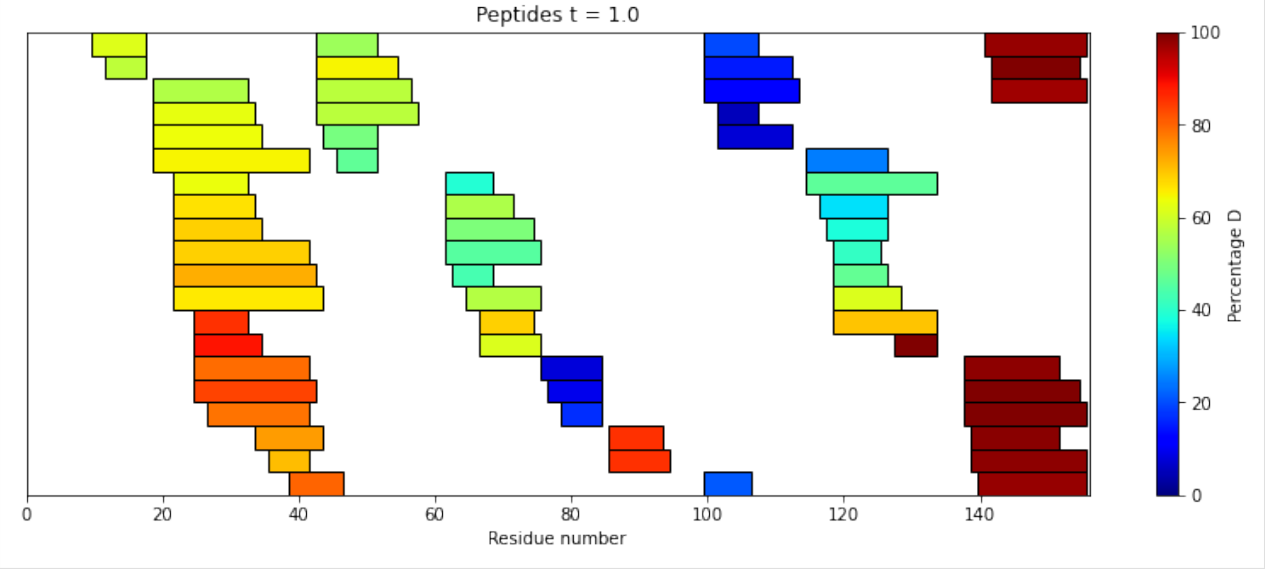
Iterating over a `HDXMeasurement` object returns a set of `PeptideMeasurements` each with their own attributes describing the topology of the coverage. When creating the object, peptides which are not present in all timepoints are removed, such that all timepoints and `PeptideMeasurements` have identical coverage.

Note that exposure time units is assumed to be minutes.

[7]:
```
fig, ax = plt.subplots(figsize=(14, 5))
i = 0
plot_peptides(hdxm[i], ax, 20, cbar=True)
t = ax.set_title(f'Peptides t = {hdxm.timepoints[i]}')
l = ax.set_xlabel('Residue number')
```



[8]:
```
fig, ax = plt.subplots(figsize=(14, 5))
i = 3
plot_peptides(hdxm[i], ax, 20, cbar=True)
t = ax.set_title(f'Peptides t = {hdxm.timepoints[i]}')
l = ax.set_xlabel('Residue number')
```



The data in an `HDXMeasurement` object can be saved to and reloaded from disk (with associated metadata) in .csv format.

[ ]:
```
from pyhdx.fileIO import csv_to_hdxm

hdxm.to_file('My_HDX_file.csv')
hdx_load = csv_to_hdxm('My_HDX_file.csv')
```

4.2 Fitting of ΔGs

[2]:
```
import matplotlib.pyplot as plt
from pyhdx import PeptideMasterTable, read_dynamx, HDXMeasurement
from pyhdx.fitting import fit_rates_half_time_interpolate, fit_rates_weighted_average,
↪ fit_gibbs_global
from pathlib import Path
import numpy as np
from dask.distributed import Client
```

We load the sample SecB dataset, apply the control, and create an `HDXMeasurement` object.

[3]:
```
fpath = Path() / '..' / '..' / 'tests' / 'test_data' / 'ecSecB_apo.csv'
data = read_dynamx(fpath)
master_table = PeptideMasterTable(data, drop_first=1, ignore_prolines=True)
master_table.set_control(('Full deuteration control', 0.167))
state_data = master_table.get_state('SecB WT apo')
hdxm = HDXMeasurement(state_data, temperature=303.15, pH=8.)
```

The first step is to obtain initial guesses for the observed rate of D-exchange. By determining the timepoint at which 0.5 RFU (relative fractional uptake) is reached, and subsequently converting to rate, a rough estimate of exchange rate per residue can be obtained. Here, RFU values are mapped from peptides to individual residues by weighted averaging (where the weight is the inverse of peptide length)

[4]:
```
fr_half_time = fit_rates_half_time_interpolate(hdxm)
fr_half_time.output
```

```
C:\Users\jhsmi\pp\PyHDX\pyhdx\models.py:615: RuntimeWarning: invalid value␣
↪encountered in true_divide
  return self.Z / np.sum(self.Z, axis=0)[np.newaxis, :]
```

[4]:
```
<pyhdx.models.Protein at 0x21d22d34970>
```

A more accurate result can be obtained by fitting the per-residue/timepoint RFU values to a biexponential association curve. This process is more time consuming and can optionally be processed in parallel by Dask.

[5]:
```
client = Client()
fr_wt_avg = fit_rates_weighted_average(hdxm, client=client)
```

```
C:\Users\jhsmi\pp\PyHDX\pyhdx\models.py:615: RuntimeWarning: invalid value␣
↪encountered in true_divide
  return self.Z / np.sum(self.Z, axis=0)[np.newaxis, :]
```

The return value is a `KineticsFitResult` object. This object has a list of models, intervals in withing the protein sequence to which these models apply, and their corresponding `symfit` fit result with parameter values. The effective exchange rate can be extracted, as well as other fit parameters, from this object:

[6]:
```
fr_wt_avg.output
```

[6]:
```
<pyhdx.models.Protein at 0x21d2d5eba90>
```

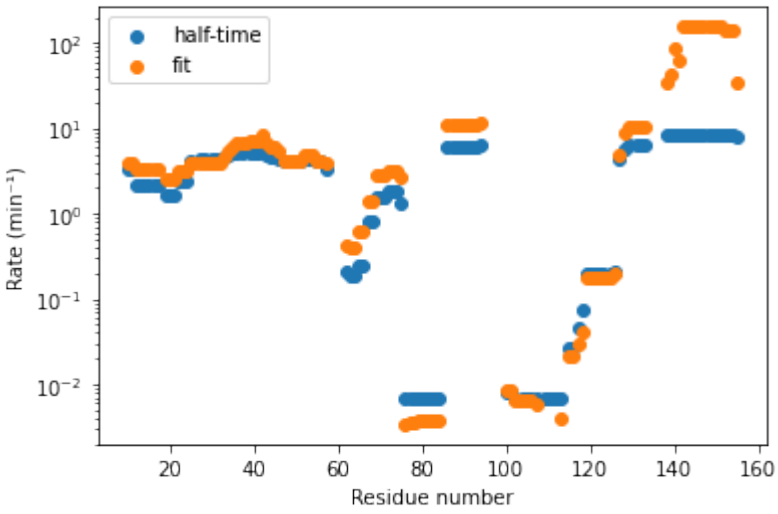
[7]:
```
fig, ax = plt.subplots()
ax.set_yscale('log')
ax.scatter(fr_half_time.output.index, fr_half_time.output['rate'], label='half-time')
ax.scatter(fr_wt_avg.output.index, fr_wt_avg.output['rate'], label='fit')

ax.set_xlabel('Residue number')
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('Rate (min$^{-1}$)')
ax.legend()
```

[7]:
```
<matplotlib.legend.Legend at 0x21d2f064d00>
```



[8]:
```
fr_half_time.output.index
fr_half_time.output['rate']
```

[8]:
```
r_number
10      3.267322
11      3.267322
12      2.118046
13      2.118046
14      2.118046
          ...
151     8.196291
152     8.187618
153     8.187618
154     8.187618
155     8.129137
Name: rate, Length: 146, dtype: float64
```

We can now use the guessed rates to obtain guesses for the Gibbs free energy. Units of Gibbs free energy are J/mol.

[9]:
```
gibbs_guess = hdxm.guess_deltaG(fr_wt_avg.output['rate'])
gibbs_guess
```

[9]:
```
r_number
10      19771.172125
11      19306.673568
12      20633.307571
13      16858.446171
14      18949.558074
             ...
151      8652.194876
152      9553.898792
153     11984.389764
154     12264.248043
```

(continues on next page)

(continued from previous page)

```
155     1388.887021
Length: 146, dtype: float64
```

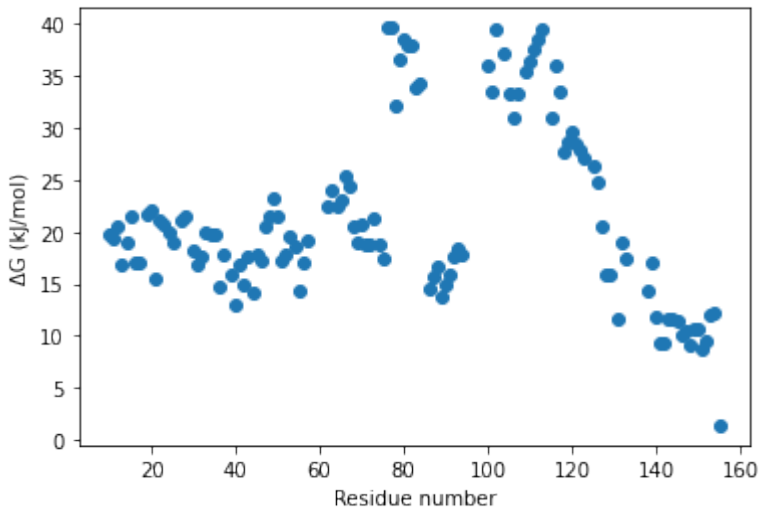
To perform the global fit (all peptides and timepoints) use `fit_gibbs_global`. The number of epochs is set to 1000 here for demonstration but for actually fitting the values should be ~100000.

[10]:
```
gibbs_result = fit_gibbs_global(hdxm, gibbs_guess, epochs=1000)
gibbs_output = gibbs_result.output
gibbs_output.output
```

[10]:
```
<pyhdx.models.Protein at 0x21d2f14bee0>
```

Along with ΔG the fit result returns covariances of ΔG and protection factors.

[11]:
```
fig, ax = plt.subplots()
#ax.set_yscale('log')
ax.scatter(gibbs_output.index, gibbs_output['deltaG']*1e-3)
ax.set_xlabel('Residue number')
ax.set_ylabel('ΔG (kJ/mol)')
None
```

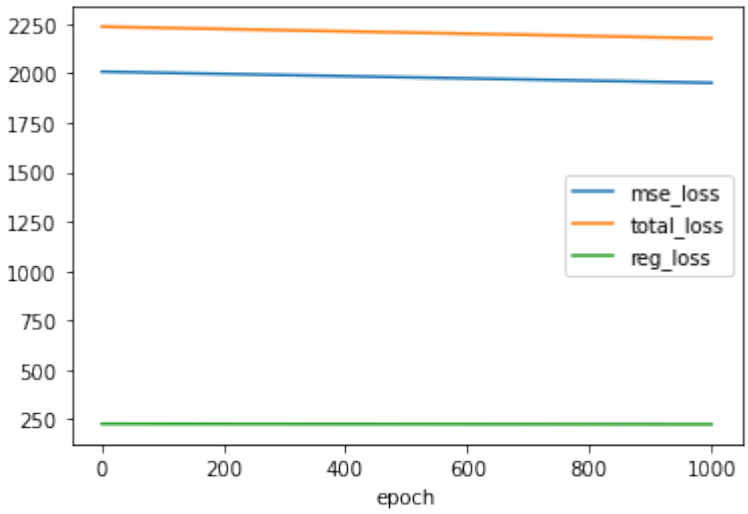


The returned fit result object also has information on the losses of each epoch of the fittng process. These are stored as a `pd.DataFrame` in the `losses` attribute. During a successful fitting run, the losses should decline and then plateau as the fit converged.

[13]:
```
plt.figure()
gibbs_result.losses[['mse_loss', 'total_loss', 'reg_loss']].plot()
```

[13]:
```
<AxesSubplot:xlabel='epoch'>
```

```
<Figure size 432x288 with 0 Axes>
```



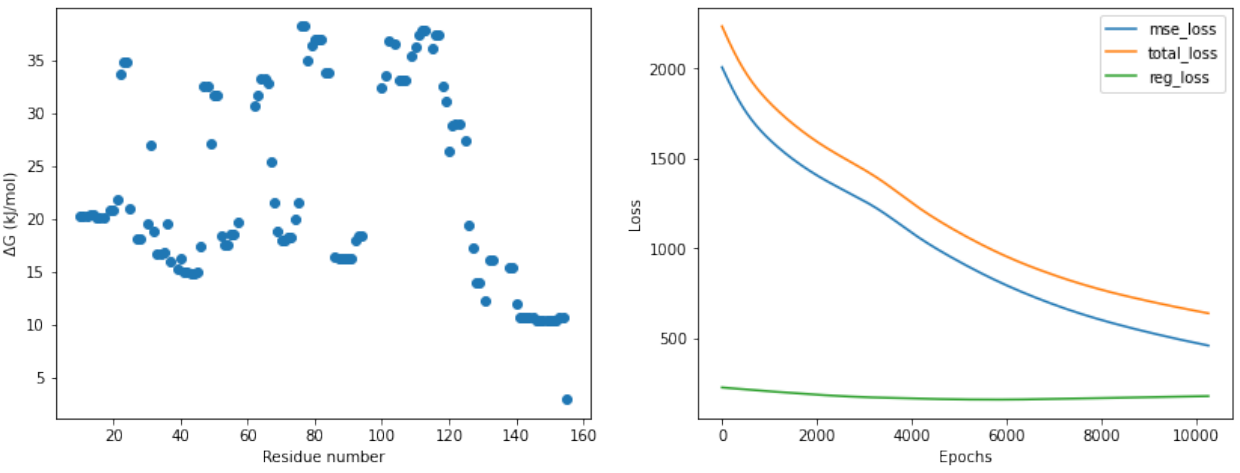
In the figure above, `mse_loss` is the loss resulting from differences in calculated D-uptake and measured D-uptake (mean squared error). The `reg_loss` are losses resulting from the regualizer.

If the losses do not decrease, this is likely due to a too low number of epochs or a too low learning rate. If the losses do not plateau, more epochs are required, or the fitting stop condition needs to be tweaked by Lets tune the fit parameters such that we obtain the desired result.

[30]:
```
gibbs_result_updated = fit_gibbs_global(hdxm, gibbs_guess, epochs=40000, lr=1e2)
```

[31]:
```
fig, axes = plt.subplots(ncols=2, figsize=(14, 5))
axes[0].scatter(gibbs_result_updated.output.index, gibbs_result_updated.output['deltaG
↪']*1e-3)
axes[0].set_xlabel('Residue number')
axes[0].set_ylabel('ΔG (kJ/mol)')
gibbs_result_updated.losses[['mse_loss', 'total_loss', 'reg_loss']].plot(ax=axes[1])
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss')
```

[31]:
```
Text(0, 0.5, 'Loss')
```



By increasing the learning rate and the number of epochs, our result improves as the final loss is now much lower, the fitting terminates prematurely as the stop threshold `stop_loss` is set too high. We rerun the analysis, now with a lower

threshold. With `stop_loss` at 0.001 and `patience` at 50 (=default), the fitting will not terminate until for 50 epochs the progress (loss decrease) has been less than 0.001.

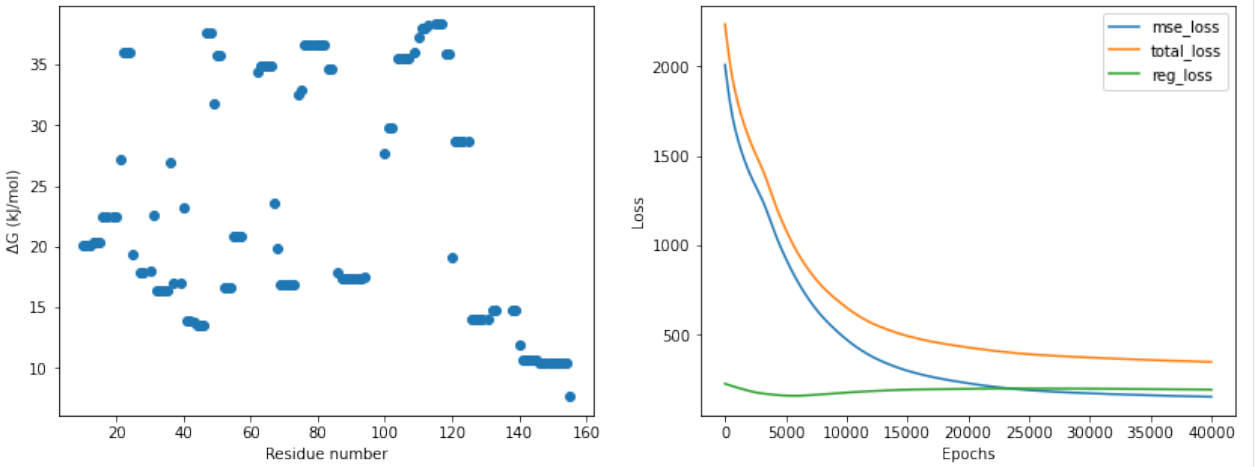
[32]:
```
gibbs_result_updated = fit_gibbs_global(hdxm, gibbs_guess, epochs=40000, lr=1e2, stop_
↪loss=0.001, patience=50)
```

[34]:
```
gibbs_result_updated.regularization_percentage
```

[34]:
```
55.70539624555
```

[33]:
```
fig, axes = plt.subplots(ncols=2, figsize=(14, 5))
axes[0].scatter(gibbs_result_updated.output.index, gibbs_result_updated.output['deltaG
↪']*1e-3)
axes[0].set_xlabel('Residue number')
axes[0].set_ylabel('ΔG (kJ/mol)')
gibbs_result_updated.losses[['mse_loss', 'total_loss', 'reg_loss']].plot(ax=axes[1])
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss')
```

[33]:
```
Text(0, 0.5, 'Loss')
```



4.2.1 The choice of regularizer value r1

The regularizer `r1` acts between subsequent residues minimizing differences between residues, unless data support these differences. Higher values flatten out the ΔG values along residues, while lower values allow for more variability.

The main function of `r1` is to mitigate the non-identifiability problem, where if multiple effective echanges rates (ΔG) values are found within a peptide, it is impossible to know which rate should be assigned to which residue. Among the possible choices, the regularizer `r1` will bias the result towards the choice of rate per residue assignment with the least variability along residues.

The 'best' value of r1 depends on the size of the protein and the coverage, (the number/size of peptides). Below is an example of the differences with regularizer values 0, 0.01 and 0.1 (=default). In this dataset, despite the fact that for `r1=0.1` contribution `reg_loss` is 50% of `total_loss`, the results are mostly the same. For `r1=0.1` we see some slight damping (lower extreme ΔG) values with respect to lower values of `r1`

[35]:
```
r1_vals = [0., 0.01, 0.1]
results_dict = {}
```

(continues on next page)

(continued from previous page)

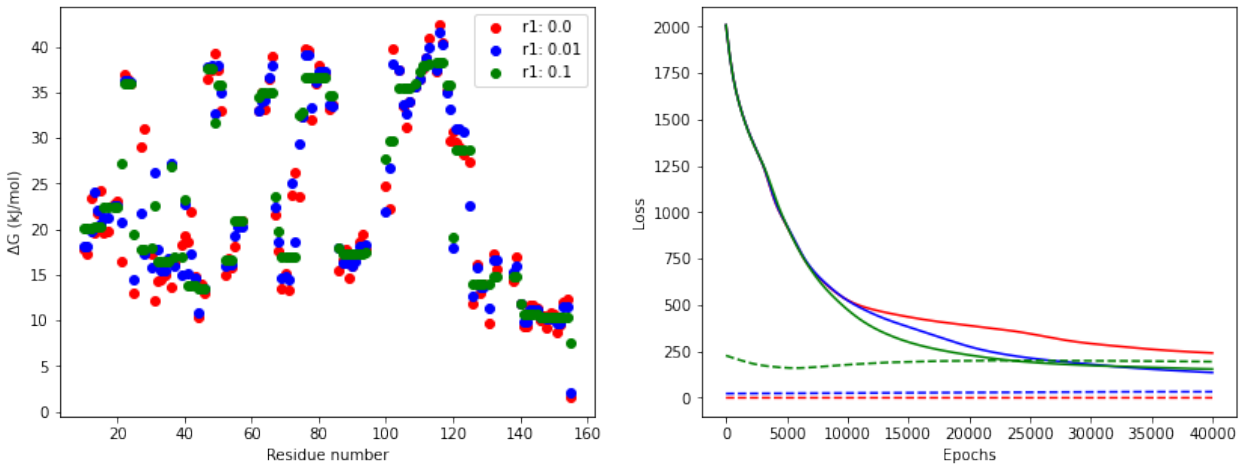
```
for r1 in r1_vals:
    print(r1)
    result = fit_gibbs_global(hdxm, gibbs_guess, epochs=40000, lr=1e2, stop_loss=0.
↪001, patience=50, r1=r1)
    results_dict[r1] = result
```

```
0.0
0.01
0.1
```

[45]:
```
colors = iter(['r', 'b', 'g'])
fig, axes = plt.subplots(ncols=2, figsize=(14, 5))
for k, v in results_dict.items():
    print(v.regularization_percentage)
    color = next(colors)
    axes[0].scatter(v.output.index, v.output['deltaG']*1e-3, color=color, label=f'r1:
↪{k}')
    axes[0].set_xlabel('Residue number')
    axes[0].set_ylabel('ΔG (kJ/mol)')
    axes[1].plot(v.losses['mse_loss'], color=color)
    axes[1].plot(v.losses['reg_loss'], color=color, linestyle='--')
    axes[1].set_xlabel('Epochs')
    axes[1].set_ylabel('Loss')

axes[0].legend()
```

```
0.0
19.27855063617599
55.70539624555
```

[45]:
```
<matplotlib.legend.Legend at 0x21d422bbb50>
```



CHAPTER

FIVE

MODULE DOCUMENTATION

This page contains the full API docs of PyHDX

5.1 Models

class pyhdx.models.**Coverage**(*data*, *c_term=0*, *n_term=1*, *sequence=''*)

Object describing layout and coverage of peptides and generating the corresponding matrices. Peptides should all belong to the same state and have the same exposure time.

Parameters

data [ndarray] Numpy structured array with input peptides

c_term [int] Residue index number of the C-terminal residue (where first residue in index number 1)

n_term [int] Residue index of the N-terminal residue. Default value is 1, can be negative to accomodate for N-terminal purification tags

sequence [str] Amino acid sequence of the protein in one-letter FASTA encoding. Optional, if not specified the amino acid sequence from the peptide data is used to (partially) reconstruct the sequence. Supplied amino acid sequence must be compatible with sequence information in the peptides.

Attributes

X [ndarray] N x M matrix where N is the number of peptides and M equal to *prot_len*. Values are 1/(ex_residues) where there is coverage.

Z [ndarray] N x M matrix where N is the number of peptides and M equal to *prot_len*. Values are 1/(ex_residues) where there is coverage, #todo account for prolines: so that rows sum to 1 is currently not true

Methods

apply_interval(array_or_series)	Given a Numpy array or Pandas series with a length equal to the full protein, returns the section of the array equal to the covered region.
get_sections([gap_size])	get the intervals of sections of coverage intervals are inclusive, exclusive

property Np

int: Number of peptides.

property Nr
int: Total number of residues spanned by the peptides.

property X_norm
ndarray: *X* coefficient matrix normalized column wise.

property Z_norm
ndarray: *Z* coefficient matrix normalized column wise.

apply_interval(*array_or_series*)
Given a Numpy array or Pandas series with a length equal to the full protein, returns the section of the array equal to the covered region. Returned series length is equal to number of columns in the X matrix

property block_length
ndarary: Lengths of unique blocks of residues in the peptides map, along the *r_number* axis

get_sections(*gap_size=- 1*)
get the intervals of sections of coverage intervals are inclusive, exclusive

gap_size : int

Gaps of this size between adjacent peptides is not considered to overlap. A value of -1 means that peptides with exactly zero overlap are separated. With gap_size=0 peptides with exactly zero overlap are not separated, and larger values tolerate larger gap sizes.

property index
RangeIndex

property percent_coverage
float: Percentage of residues covered by peptides

property r_number
ndarray: Array of residue numbers corresponding to the part of the protein covered by peptides

property redundancy
float: Average redundancy of peptides in regions with at least 1 peptide

class pyhdx.models.**HDXMeasurement**(*data*, ***metadata*)
Main HDX data object. This object has peptide data of a single state but with multiple timepoints.

Timepoint data is split into *PeptideMeasurements* objects for each timepoint Supplied data is made 'uniform' such that all timepoints have the same peptides

Parameters

data [ndarray or list] Numpy structured array with peptide entries corresponding to a single state, or list of *PeptideMeasurements*

****metadata** Dictionary of optional metadata. By default, holds the *temperature* and *pH* parameters.

Attributes

state [str] State of the HDX measurement

timepoints [ndarray] Array with exposure times (sorted)

peptides [list] List of *PeptideMeasurements*, one list element per timepoint.

cov [*Coverage*] Coverage object describing peptide layout. When this *uniform* is *False*, this attribute is *None*

Methods

get_tensors([exchanges])	Returns a dictionary of tensor variables for fitting to Linderstrøm-Lang kinetics.
guess_deltaG(rates[, crop])	**Parameters**
to_file(file_path[, include_version, …])	Write Protein data to file.

property Np
int: Number of peptides.

property Nr
int: Total number of residues spanned by the peptides.

property Nt
int: Number of timepoints.

property d_exp
np.ndarray (shape Np x Nt) Experimentally measured D-uptake values, corrected for back-exchange

property full_data
returns the full dataset of all timepoints

get_tensors(*exchanges=False*)
Returns a dictionary of tensor variables for fitting to Linderstrøm-Lang kinetics.

Tensor variables are (shape): Temperature (1 x 1) X (Np x Nr) k_int (Nr x 1) timepoints (1 x Nt) uptake (D) (Np x Nt)

Parameters

exchanges [bool] if True only returns tensor data describing residues which exchange (ie have peptides and are not prolines)

Returns

tensors [dict]

guess_deltaG(*rates*, *crop=True*)

Parameters

rates [Series] pandas series of estimated hdx exchangs rates. Index is protein residue number

return_type

property rfu_residues
Relative fractional uptake per residue. Shape Nr x Nt

to_file(*file_path*, *include_version=True*, *include_metadata=True*, *fmt='csv'*, ***kwargs*)
Write Protein data to file.

Parameters

file_path [str] File path to create and write to.

include_version [bool] Set True to include PyHDX version and current time/date

fmt: :obj: `str` Formatting to use, options are 'csv' or 'pprint'

include_metadata [bool] If *True*, the objects' metadata is included

****kwargs** [dict, optional] Optional additional keyword arguments passed to *df.to_csv*

Returns

——-

None

property uptake_corrected
matrix shape N_t, N_p

class pyhdx.models.**HDXMeasurementSet**(*hdxm_list*)
Set of multiple *HDXMeasurement*

Parameters

hdxm_list [list] or list of *HDXMeasurement*

Attributes

timepoints [ndarray] Ns x Nt array of zero-padded timepoints

d_exp [ndarray] Ns x Np x Nt array with zero-padded measured D-uptake values

Methods

add_alignment(alignment[, first_r_numbers])	**param alignment** list
guess_deltaG(rates_list)	create deltaG guesses from rates

get_tensors	

add_alignment(*alignment*, *first_r_numbers=None*)

Parameters

- **alignment** – list
- **first_r_numbers** – default is [1, 1, …] but specifiy here if alignments do not all start at residue 1

Returns

guess_deltaG(*rates_list*)
create deltaG guesses from rates

Parameters

rates_list [iterable] list of pandas series with k_obs esimates

Returns

deltaG_array: ndarray deltaG guesses Ns x Nr shape

class pyhdx.models.**PeptideMasterTable**(*data*, *drop_first=1*, *ignore_prolines=True*, *d_percentage=100.0*, *sort=True*, *remove_nan=True*)
Main peptide input object. The input numpy structured array *data* must have the following entires for each peptide:

start: Residue number of the first amino acid in the peptide end: Residue number of the last amino acid in the peptide (inclusive) sequence: Amino acid sequence of the peptide (one letter code) exposure: Typically the time the sample was exposed to a deuterated solution. This can correspond to other times if

the kinetics of the experiment are set up differently

state: String describing to which state (experimental conditions) the peptide belongs uptake: Number of deuteriums the peptide has taken up

The following fields are added to the *data* array upon initialization:

_start: Unmodified copy of initial start field _end: Unmodified copy of initial end field _sequence: Unmodified copy of initial sequence ex_residues: Number of residues that undergo deuterium exchange. This number is calculated using the *drop_first* and

ignore_prolines parameters

N-terminal residues which are removed because they are either within *drop_first* or they are N-terminal prolines are marked with 'x' in the *sequence* field. Prolines which are removed because they are in the middle of a peptide are marked with a lower case 'p' in the sequence field.

The field *scores* is used in calculating exchange rates and can be set by either the *set_backexchange* or *set_control* methods.

Parameters

data [ndarray] Numpy recarray with peptide entries.

drop_first [int] Number of N-terminal amino acids to ignore. Default is 1.

d_percentage [float] Percentage of deuterium in the labelling solution.

ignore_prolines [bool] Boolean to toggle ignoring of proline residues. When True these residues are treated as if they're not present in the protein.

sort [bool] Set to `True` to sort the input. Sort order is 'start', 'end', 'sequence', 'exposure', 'state'.

remove_nan [bool] Set to `True` to remove NaN entries in uptake

Attributes

exposures ndarray Array with unique exposures

states ndarray Array with unique states

Methods

get_data(state, exposure)	Get all peptides matching *state* and *exposure*.
get_state(state)	Returns entries in the table with state 'state'
groupby_state(**kwargs)	Groups measurements in the dataset by state and returns them in a dictionary as a *HDXMeasurement*.
isin_by_idx(array, test_array)	Checks if entries in *array* are in *test_array*, by *start* and *end* field values.
set_backexchange(back_exchange)	Sets the normalized percentage of uptake through a fixed backexchange value for all peptides.
set_control(control_1[, control_0])	Apply a control dataset to this object.

property exposures
ndarray Array with unique exposures

get_data(*state*, *exposure*)
Get all peptides matching *state* and *exposure*.

Parameters

state [str] Measurement state

exposure [float] Measurement exposure time

Returns

output_data [ndarray] Numpy structured array with selected peptides

get_state(*state*)
Returns entries in the table with state 'state'

Parameters

state [str]

groupby_state(***kwargs*)
Groups measurements in the dataset by state and returns them in a dictionary as a *HDXMeasurement*.

Returns

out [dict] Dictionary where keys are state names and values are *HDXMeasurement*.

****kwargs** Additional keyword arguments to be passed to the *HDXMeasurement*.

static isin_by_idx(*array*, *test_array*)
Checks if entries in *array* are in *test_array*, by *start* and *end* field values.

Parameters

array [ndarray] Numpy input structured array

test_array [ndarray] Numpy structured array to test againts

Returns

isin [ndarray, bool] Boolean array of the same shape as *array* where entries are *True* if they are in *test_array*

set_backexchange(*back_exchange*)
Sets the normalized percentage of uptake through a fixed backexchange value for all peptides.

Parameters

back_exchange [float] Percentage of back exchange

set_control(*control_1*, *control_0=None*)
Apply a control dataset to this object. A *scores* attribute is added to the object by normalizing its uptake value with respect to the control uptake value to 100%. Entries which are in the measurement and not in the control or vice versa are deleted. Optionally, control_zero can be specified which is a dataset whose uptake value will be used to zero the uptake.

#todo insert math

Parameters

control_1 [tuple] tuple with (*state*, *exposure*) for peptides to use for normalization

control_0 [tuple, optional] tuple with (*state*, *exposure*) for peptides to use for zeroing uptake values

property states
ndarray Array with unique states

class pyhdx.models.**PeptideMeasurements**(*data*, ***kwargs*)
Class with subset of peptides corresponding to only one state and exposure

Parameters

data [ndarray] Numpy structured array with input data

scores [ndarray] Array with D/H uptake scores, typically in percentages or absolute uptake numbers.

Attributes

start [int] First peptide starts at this residue number (starting from 1)

stop [int] Last peptide ends at this residue number (incusive)

prot_len [int] Total number of residues in this set of peptides, not taking regions of no coverage into account.

exposure [float] Exposure time of this set of peptides (minutes)

state [string] State describing the experiment

bigX

X

properties:

big_x_norm

x_norm

scores nnls

scores lsq

Methods

calc_rfu(residue_rfu)	Calculates RFU per peptide given an array of individual residue scores
weighted_average(field)	Calculate per-residue weighted average of values in data column given by 'field'

calc_rfu(*residue_rfu*)
Calculates RFU per peptide given an array of individual residue scores

Parameters

residue_rfu [ndarray] Array of rfu per residue of length *prot_len*

Returns

scores [ndarray] Array of rfu per peptide

property rfu_residues
Weighted averaged relative fractional uptake

weighted_average(*field*)
Calculate per-residue weighted average of values in data column given by 'field'

class pyhdx.models.**Protein**(*data*, *index=None*, ***metadata*)
Object describing a protein

Protein objects are based on panda's DataFrame's with added functionality

Parameters

data [ndarray or dict or DataFrame] data object to initiate the protein object from

index [str, optional] Name of the column with the residue number (index column)

****metadata** Dictionary of optional metadata.

Attributes

c_term

n_term

Methods

set_k_int(temperature, pH)	Calculates the intrinsic rate of the sequence.
to_file(file_path[, include_version, …])	Write Protein data to file.

set_k_int (*temperature*, *pH*)

Calculates the intrinsic rate of the sequence. Values of no coverage or prolines are assigned a value of -1 The rates run are for the first residue (1) up to the last residue that is covered by peptides

When the previous residue is unknown the current residue is also assigned a value of -1.g

Parameters

temperature [float] Temperature of the labelling reaction (Kelvin)

pH [float] pH of the labelling reaction

Returns

k_int [ndarray] Array of intrisic exchange rates

to_file (*file_path*, *include_version=True*, *include_metadata=True*, *fmt='csv'*, ***kwargs*)

Write Protein data to file.

Parameters

file_path [str] File path to create and write to.

include_version [bool] Set True to include PyHDX version and current time/date

fmt [str] Formatting to use, options are 'csv' or 'pprint'

include_metadata [bool] If *True*, the objects' metadata is included

****kwargs** [dict, optional] Optional additional keyword arguments passed to *df.to_csv*

Returns

——-

None

pyhdx.models.**array_intersection** (*arrays_list*, *fields*)

Find and return the intersecting entries in multiple arrays.

Parameters

arrays_list [iterable] Iterable of input structured arrays

fields [iterable] Iterable of fields to use to decide if entires are intersecting

Returns

selected [iterable] Output iterable of arrays with only intersecting entries.

pyhdx.models.**contiguous_regions**(*condition*)
Finds contiguous True regions of the boolean array "condition". Returns a 2D array where the first column is the start index of the region and the second column is the end index.

pyhdx.models.**hdx_intersection**(*hdx_list*, *fields=None*)
Finds the intersection between peptides in HDXMeasurement and returns new objects such that all peptides (coverage, exposure) between the measurements are identical.

Optionally intersections by custom fields can be made.

Parameters

hdx_list [list] Input list of *HDXMeasurement*

fields [list] By which fields to take the intersections. Default is ['_start', '_end', 'exposure']

Returns

hdx_out [list] Output list of *HDXMeasurement*

5.2 Fitting

class pyhdx.fitting.**EmptyResult**(*chi_squared*, *params*)

Attributes

chi_squared Alias for field number 0

params Alias for field number 1

chi_squared
Alias for field number 0

params
Alias for field number 1

class pyhdx.fitting.**KineticsFitResult**(*hdxm*, *intervals*, *results*, *models*)
Fit result object. Generally used for initial guess results.

Parameters

hdxm [*HDXMeasurement*]

intervals:

results:

models:

Attributes

model_type

output

rate Returns an array with the exchange rates

tau Returns an array with the exchange rates

Methods

`__call__`(timepoints)	call the result with timepoints to get fitted uptake per peptide back
get_d(t)	calculate d at timepoint t only for lsqkinetics (refactor glocal) type fitting results (scores per peptide)
get_p(t)	Calculate P at timepoint t.
get_param(name)	Get an array of parameter with name *name* from the fit result.

get_output

get_d(*t*)
calculate d at timepoint t only for lsqkinetics (refactor glocal) type fitting results (scores per peptide)

get_p(*t*)
Calculate P at timepoint t. Only for wt average type fitting results

get_param(*name*)
Get an array of parameter with name *name* from the fit result. The length of the array is equal to the number of amino acids.

Parameters

name [str] Name of the parameter to extract

Returns

par_arr [ndarray] Array with parameter values

property rate
Returns an array with the exchange rates

property tau
Returns an array with the exchange rates

pyhdx.fitting.**check_bounds**(*fit_result*)
Check if the obtained fit result is within bounds

pyhdx.fitting.**fit_gibbs_global**(*hdxm*, *initial_guess*, *r1=0.1*, *epochs=100000*, *patience=50*, *stop_loss=0.05*, *optimizer='SGD'*, ***optimizer_kwargs*)
Fit Gibbs free energies globally to all D-uptake data in the supplied hdxm

Parameters

hdxm [*HDXMeasurement*]

initial_guess [Series or ndarray] Gibbs free energy initial guesses (shape Nr)

r1 [float]

epochs

patience

stop_loss

optimizer [str]

optimizer_kwargs

pyhdx.fitting.**fit_gibbs_global_batch**(*hdx_set*, *initial_guess*, *r1=2*, *r2=5*, *r2_reference=False*, *epochs=100000*, *patience=50*, *stop_loss=0.05*, *optimizer='SGD'*, ***optimizer_kwargs*)

Batch fit gibbs free energies to multiple HDX measurements

Parameters

hdx_set [*HDXMeasurementSet*]

initial_guess

r1

r2

r2_reference, if true the first dataset is used as a reference to calculate r2 differences, otherwise the mean is used

epochs

patience

stop_loss

optimizer

optimizer_kwargs

pyhdx.fitting.**fit_gibbs_global_batch_aligned**(*hdx_set*, *initial_guess*, *r1=2*, *r2=5*, *epochs=100000*, *patience=50*, *stop_loss=0.05*, *optimizer='SGD'*, ***optimizer_kwargs*)

Batch fit gibbs free energies to two HDX measurements. The supplied HDXMeasurementSet must have alignment information (supplied by HDXMeasurementSet.add_alignment)

Parameters

hdx_set [*HDXMeasurement*]

initial_guess

r1

r2

epochs

patience

stop_loss

optimizer

optimizer_kwargs

pyhdx.fitting.**fit_kinetics**(*t*, *d*, *model*, *chisq_thd=100*)

Fit time kinetics with two time components and corresponding relative amplitude.

Parameters

t [ndarray] Array of time points

d [ndarray] Array of uptake values

model [KineticsModel]

chisq_thd [float] Threshold chi squared above which the fitting is repeated with the Differential Evolution algorithm.

Returns

res [FitResults] Symfit fitresults object.

pyhdx.fitting.**fit_rates**(*hdxm*, *method='wt_avg'*, ***kwargs*)

Fit observed rates of exchange to HDX-MS data in *hdxm*

Parameters

hdxm [*HDXMeasurement*]

method [str] Method to use to determine rates of exchange

kwargs Additional kwargs passed to fitting

Returns

fit_result [*KineticsFitResult*]

pyhdx.fitting.**fit_rates_half_time_interpolate**(*hdxm*)

Calculates exchange rates based on weighted averaging followed by interpolation to determine half-time, which is then calculated to rates.

Parameters

hdxm [*HDXMeasurement*]

Returns

output: **ndarray** array with fields r_number, rate

pyhdx.fitting.**fit_rates_weighted_average**(*hdxm*, *bounds=None*, *chisq_thd=0.2*, *model_type='association'*, *client=None*, *pbar=None*)

Fit a model specified by 'model_type' to D-uptake kinetics. D-uptake is weighted averaged across peptides per timepoint to obtain residue-level D-uptake.

Parameters

hdxm [*HDXMeasurement*]

bounds [tuple, optional] Tuple of lower and upper bounds of rate constants in the model used.

chisq_thd [float] Threshold of chi squared result, values above will trigger a second round of fitting using DifferentialEvolution

model_type [str] Missing docstring

client [: ??] Controls delegation of fitting tasks to Dask clusters. Options are: *None*: Do not use task, fitting is done in the local thread in a for loop. :class: Dask Client : Uses the supplied Dask client to schedule fitting task. *worker_client*: The function was ran by a Dask worker and the additional fitting tasks created are scheduled on the same Cluster.

pbar: Not implemented

Returns

fit_result [*KineticsFitResult*]

pyhdx.fitting.**get_bounds**(*times*)

estimate default bound for rate fitting from a series of timepoints

Parameters

times [array_like]

Returns

bounds [tuple] lower and upper bounds

pyhdx.fitting.**run_optimizer**(*inputs*, *output_data*, *optimizer_klass*, *optimizer_kwargs*, *model*, *criterion*, *regularizer*, *epochs=100000*, *patience=50*, *stop_loss=0.05*)

Runs optimization/fitting of PyTorch model.

Parameters

inputs [list] List of input Tensors

output_data [Tensor] comparison data to model output

optimizer_klass [optim]

optimizer_kwargs [dict] kwargs to pass to pytorch optimizer

model [Module] pytorch model

criterion: callable loss function

regularizer callable regularizer function

epochs [int] Max number of epochs

patience [int] Number of epochs with less progress than *stop_loss* before terminating optimization

stop_loss [float] Threshold of optimization value below which no progress is made

5.3 Fitting PyTorch

class pyhdx.fitting_torch.**DeltaGFit**(*deltaG*)

Methods

forward(temperature, X, k_int, timepoints)	# inputs, list of:

forward(*temperature*, *X*, *k_int*, *timepoints*)

 # inputs, list of: temperatures: scalar (1,) X (N_peptides, N_residues) k_int: (N_peptides, 1)

class pyhdx.fitting_torch.**TorchBatchFitResult**(**args*, ***kwargs*)

Methods

__call__(timepoints)	timepoints: must be Ns x Nt, will be reshaped to Ns x 1 x Nt output: Ns x Np x Nt array

class pyhdx.fitting_torch.**TorchFitResult**(*data_obj*, *model*, *losses=None*, ***metadata*)

PyTorch Fit result object.

Parameters

data_obj [*HDXMeasurement* or *HDXMeasurementSet*]

model

****metdata**

Attributes

deltaG output deltaG as `Series` or as `DataFrame`

mse_loss obj:*float*: Losses from mean squared error part of Lagrangian

reg_loss `float`: Losses from regularization part of Lagrangian

regularization_percentage `float`: Percentage part of the total loss that is regularization loss

total_loss obj:*float*: Total loss value of the Lagrangian

Methods

generate_output(hdxm, deltaG)	**Parameters**
get_mse()	np.ndarray: Returns the mean squared error per peptide per timepoint.

to_file	

property deltaG
output deltaG as `Series` or as `DataFrame`

index is residue numbers

static generate_output(*hdxm*, *deltaG*)

Parameters

hdxm [*HDXMeasurement*]

deltaG [`Series` with r_number as index]

get_mse()
np.ndarray: Returns the mean squared error per peptide per timepoint. Output shape is Np x Nt

property mse_loss
obj:*float*: Losses from mean squared error part of Lagrangian

property reg_loss
`float`: Losses from regularization part of Lagrangian

property regularization_percentage
`float`: Percentage part of the total loss that is regularization loss

property total_loss
obj:*float*: Total loss value of the Lagrangian

class pyhdx.fitting_torch.**TorchSingleFitResult**(**args*, ***kwargs*)

Methods

`__call__`(timepoints)	timepoints: Nt array (will be unsqueezed to 1 x Nt) output: Np x Nt array

pyhdx.fitting_torch.**estimate_errors**(*hdxm*, *deltaG*)

Calculate covariances

Parameters

hdxm [*HDXMeasurement*]

deltaG [ndarray] Array with deltaG values.

5.4 FileIO

pyhdx.fileIO.**csv_to_dataframe**(*filepath_or_buffer*, *comment='#'*, ***kwargs*)

Reads a .csv file or buffer into a **:pandas:`DataFrame`** object. Comment lines are parsed where json dictionaries marked by tags are read. The <pandas_kwargs> marked json dict is used as kwargs for *pd.read_csv* The <metadata> marked json dict is stored in the returned dataframe object as `df.attrs['metadata'].

Parameters

filepath_or_buffer [str, pathlib.Path or io.StringIO] Filepath or StringIO buffer to read.

comment [str] Indicates which lines are comments.

kwargs Optional additional keyword arguments passed to *pd.read_csv*

Returns

——-

df: pd.DataFrame

pyhdx.fileIO.**csv_to_hdxm**(*filepath_or_buffer*, *comment='#'*, ***kwargs*)

Reads a .csv file or buffer into a pyhdx.models.HDXMeasurement object. Comment lines are parsed where json dictionaries marked by tags are read. The <pandas_kwargs> marked json dict is used as kwargs for *pd.read_csv* The <metadata> marked json dict is stored in the returned dataframe object as `df.attrs['metadata'].

Parameters

filepath_or_buffer [str or pathlib.Path or io.StringIO] Filepath or StringIO buffer to read.

comment [str] Indicates which lines are comments.

****kwargs** [dict, optional] Optional additional keyword arguments passed to *pd.read_csv*

Returns

——-

protein [pyhdx.models.HDXMeasurement] Resulting HDXMeasurement object with *r_number* as index

pyhdx.fileIO.**csv_to_protein**(*filepath_or_buffer*, *comment='#'*, ***kwargs*)

Reads a .csv file or buffer into a pyhdx.models.Protein object. Comment lines are parsed where json dictionaries marked by tags are read. The <pandas_kwargs> marked json dict is used as kwargs for *pd.read_csv* The <metadata> marked json dict is stored in the returned dataframe object as `df.attrs['metadata'].

Parameters

filepath_or_buffer [str or pathlib.Path or io.StringIO] Filepath or StringIO buffer to read.

comment [str] Indicates which lines are comments.

****kwargs** [dict, optional] Optional additional keyword arguments passed to *pd.read_csv*

Returns

——-

protein [pyhdx.models.Protein] Resulting Protein object with *r_number* as index

pyhdx.fileIO.**dataframe_to_file**(*file_path*, *df*, *fmt='csv'*, *include_metadata=True*, *include_version=False*, ***kwargs*)

Save a pd.DataFrame to an io.StringIO object. Kwargs to read the resulting .csv object with pd.read_csv to get the original pd.DataFrame back are included in the comments. Optionally additional metadata or the version of PyHDX used can be included in the comments.

Parameters

file_path: :obj:`str` or `pathlib.Path` File path of the target file to write.

df: pd.DataFrame The pandas dataframe to write to the file.

fmt: :obj:`str` Specify the formatting of the output. Options are '.csv' (machine readable) or 'pprint' (human readable)

include_metadata: :obj:`bool` or :obj:`dict` If *True*, the metadata in df.attrs['metadata'] is included. If dict, this dictionary is used as the metadata. If *False*, no metadata is included.

include_version [bool] *True* to include PyHDX version information.

****kwargs** [dict, optional] Optional additional keyword arguments passed to *df.to_csv*

Returns

sio: io.StringIO Resulting io.StringIO object.

pyhdx.fileIO.**dataframe_to_stringio**(*df*, *sio=None*, *fmt='csv'*, *include_metadata=True*, *include_version=True*, ***kwargs*)

Save a pd.DataFrame to an io.StringIO object. Kwargs to read the resulting .csv object with pd.read_csv to get the original pd.DataFrame back are included in the comments. Optionally additional metadata or the version of PyHDX used can be included in the comments.

Parameters

df: pd.DataFrame The pandas dataframe to write to the io.StringIO object.

sio: `io.StringIO`, optional The *io.StringIO* object to write to. If *None*, a new *io.StringIO* object is created.

fmt: :obj:`str` Specify the formatting of the output. Options are 'csv' (machine readable) or 'pprint' (human readable)

include_metadata: :obj:`bool` or :obj:`dict` If *True*, the metadata in df.attrs['metadata'] is included. If dict, this dictionary is used as the metadata. If *False*, no metadata is included.

include_version [bool] *True* to include PyHDX version information.

****kwargs** [dict, optional] Optional additional keyword arguments passed to *df.to_csv*

Returns

sio: io.StringIO Resulting io.StringIO object.

pyhdx.fileIO.**fmt_export**(*arr*, *delimiter='\t'*, *header=True*, *sig_fig=8*, *width='auto'*, *justify='left'*, *sign=False*, *pad=''*)
Create a format string for array *arr* such that columns are aligned in the output file when saving with np.savetxt

pyhdx.fileIO.**load_fitresult**(*fit_dir_or_file*, *hdxm=None*, *losses=None*)
Load a fitresult into a fitting_torch.TorchSingleFitResult object

Loading of TorchBatchFitResult is not implemented

pyhdx.fileIO.**read_dynamx**(**file_paths*, *intervals=('inclusive', 'inclusive')*, *time_unit='min'*)
Reads a dynamX .csv file and returns the data as a numpy structured array

Parameters

file_paths [iterable] File path of the .csv file or StringIO object

intervals [tuple] Format of how start and end intervals are specified.

time_unit [str] Not implemented

Returns

data [ndarray] Peptides as a numpy structured array

pyhdx.fileIO.**save_fitresult**(*output_dir*, *fit_result*, *log_lines=None*)
Save a fit result object to the specified directory with associated metadata

Output directory contents: deltaG.csv/.txt: Fit output result (deltaG, covariance, k_obs, pfact) losses.csv/.txt: Losses per epoch log.txt: Log file with additional metadata (number of epochs, final losses, pyhdx version, time/date)

Parameters

output_dir: pathlib.Path or :obj:`str` Output directory to save fitresult to

fit_result: pydhx.fittin_torch.TorchFitResult fit result object to save

log_lines: :obj:`list` Optional additional lines to write to log file.

5.5 Output

This module allows users to generate a .pdf output report from their HDX measurement

(Currently outdated/not working)

class pyhdx.output.**Report**(*output*, *name=None*, *doc=None*, *add_date=True*)
.pdf output document

Methods

rm_temp_dir()	Remove the temporary directory specified in _tmp_path.

add_coverage_figures	
add_peptide_figures	
generate_pdf	
make_subfigure	
make_temp_dir	
test_mpl	
test_subfigure	

rm_temp_dir()
Remove the temporary directory specified in `_tmp_path`.

5.6 Support

pyhdx.support.**autowrap**(*start*, *end*, *margin=4*, *step=5*)
Automatically finds wrap value for coverage to not have overlapping peptides within margin

Parameters

start

end

margin

pyhdx.support.**colors_to_pymol**(*r_number*, *color_arr*, *c_term=None*, *no_coverage='#8c8c8c'*)
coverts colors (hexadecimal format) and corresponding residue numbers to pml script to color structures in pymol residue ranges in output are inclusive, incluive

c_term: optional residue number of the c terminal of the last peptide doedsnt cover the c terminal

pyhdx.support.**gen_subclasses**(*cls*)
Recursively find all subclasses of cls

pyhdx.support.**grouper**(*3*, *'abcdefg'*, *'x')* --> *('a'*, *'b'*, *'c')*, *('d'*, *'e'*, *'f')*, *('g'*, *'x'*, *'x'*)

pyhdx.support.**hex_to_rgb**(*h*)
returns rgb as int 0-255

pyhdx.support.**make_color_array**(*rates*, *colors*, *thds*, *no_coverage='#8c8c8c'*)

Parameters

- **rates** – array of rates
- **colors** – list of colors (slow to fast)
- **thds** – list of thresholds

no_coverage: color value for no coverage :return:

pyhdx.support.**make_monomer**(*input_file*, *output_file*)
reads input_file pdb file and removes all chains except chain A and all water

pyhdx.support.**multi_otsu**(**rates*, *classes=3*)
global otsu thesholding of multiple rate arrays in log space

Parameters

rates [iterable] iterable of numpy structured arrays with a 'rate' field

classes [int] Number of classes to divide the data into

Returns

thds [tuple] tuple with thresholds

pyhdx.support.**pprint_df_to_file**(*df*, *file_path_or_obj*)
Pretty print (human-readable) a dataframe to a file

Parameters

df [DataFrame]

file_path_or_obj [str, Path or StringIO]

pyhdx.support.**reduce_inter**(*args*, *gap_size=- 1*)

gap_size [int] Gaps of this size between adjacent peptides is not considered to overlap. A value of -1 means that peptides with exactly zero overlap are separated. With gap_size=0 peptides with exactly zero overlap are not separated, and larger values tolerate larger gap sizes.

https://github.com/brentp/interlap/blob/3c4a5923c97a5d9a11571e0c9ea5bb7ea4e784ee/interlap.py#L224 # MIT Liscence >>> reduce_inter([(2, 4), (4, 9)]) [(2, 4), (4, 9)] >>> reduce_inter([(2, 6), (4, 10)]) [(2, 10)]

pyhdx.support.**rgb_to_hex**(*rgb_a*)
Converts rgba input values are [0, 255]

alpha is set to zero

returns as '#000000'

pyhdx.support.**scale**(*x*, *out_range=(- 1, 1)*)
rescale input array x to range *out_range*

pyhdx.support.**series_to_pymol**(*pd_series*)
Coverts a pandas series to pymol script to color proteins structures in pymol Series must have hexadecimal color values and residue number as index

Parameters

pd_series [Series]

Returns

s_out [str]

pyhdx.support.**try_wrap**(*start*, *end*, *wrap*, *margin=4*)
Check for a given coverage if the value of wrap is high enough to not have peptides overlapping within margin

start, end interval is inclusive, exclusive

CHAPTER

SIX

WEB APPLICATION REFERENCE

This page contains auto-generated docs for PyHDX' web application.

The functionality in can be controlled by *Controllers* which can be found in the left sidebar. The control parameters of every controller per app is listed in the sections below.

6.1 Main Application

class pyhdx.web.controllers.**PeptideFileInputControl**(*parent*, ***params*)

Peptide Input

This controller allows users to input .csv file (Currently only DynamX format) of 'state' peptide uptake data. Users can then choose how to correct for back-exchange and which 'state' and exposure times should be used for analysis.

Input files (*List*, bounds=(0, None), default=[])

Back exchange correction method (*Selector*, default='FD Sample', options=['FD Sample', 'Flat percentage'])
Select method of back exchange correction

FD State (*Selector*, options=[])
State used to normalize uptake

FD Exposure (*Selector*, options=[])
Exposure used to normalize uptake

Experiment State (*Selector*, options=[])
State for selected experiment

Experiment Exposures (*ListSelector*, default=[], options=[''])
Selected exposure time to use

Back exchange percentage (*Number*, bounds=(0, 100), default=28.0)

Global percentage of back-exchange

Drop first (*Integer*, bounds=(0, None), default=1)
Select the number of N-terminal residues to ignore.

Ignore prolines (*Boolean*, bounds=(0, 1), default=True)
Prolines are ignored as they do not exchange D.

Deuterium percentage (*Number*, bounds=(0, 100), default=95.0)
Percentage of deuterium in the labelling buffer

FD Deuterium percentage (*Number*, bounds=(0, 100), default=95.0)
Percentage of deuterium in the FD control sample buffer

Temperature (K) (*Number*, bounds=(0, 373.15), default=293.15)
Temperature of the D-labelling reaction

pH read (*Number*, default=7.5)
pH of the D-labelling reaction, as read from pH meter

N term (*Integer*, default=1)
Index of the n terminal residue in the protein. Can be set to negative values to accommodate for purification tags. Used in the determination of intrinsic rate of exchange

C term (*Integer*, bounds=(0, None), default=0)
Index of the c terminal residue in the protein. Used for generating pymol export scriptand determination of intrinsic rate of exchange for the C-terminal residue

Sequence (*String*, default=”)
Optional FASTA protein sequence

Dataset name (*String*, default=”)

Add dataset (*Action*)
Parse selected peptides for further analysis and apply back-exchange correction

Datasets (*ListSelector*, options=[])
Lists available datasets

class pyhdx.web.controllers.**CoverageControl**(*parent*, ***params*)

Coverage

class pyhdx.web.controllers.**InitialGuessControl**(*parent*, ***params*)

Initial Guesses

This controller allows users to derive initial guesses for D-exchange rate from peptide uptake data.

Fitting model (*Selector*, default='Half-life (λ)', options=['Half-life (λ)', 'Association'])
Choose method for determining initial guesses.

Dataset (*Selector*, default='', options=[])
Dataset to apply bounds to

Global bounds (*Boolean*, bounds=(0, 1), default=False)
Set bounds globally across all datasets

Lower bound (*Number*, default=0.0)
Lower bound for association model fitting

Upper bound (*Number*, default=0.0)
Upper bound for association model fitting

Guess name (*String*, default='Guess_1')
Name for the initial guesses

Calculate Guesses (*Action*)
Start initial guess fitting

class pyhdx.web.controllers.**FitControl**(*parent*, ***params*)

Fitting

This controller allows users to execute PyTorch fitting of the global data set.

Currently, repeated fitting overrides the old result.

Initial guess (*Selector*, options=[])
Name of dataset to use for initial guesses.

Fit mode (*Selector*, default='Batch', options=['Batch', 'Single'])

Stop loss (*Number*, bounds=(0, None), default=0.01)
Threshold loss difference below which to stop fitting.

Stop patience (*Integer*, bounds=(1, None), default=100)
Number of epochs where stop loss should be satisfied before stopping.

Learning rate (*Number*, bounds=(0, None), default=10)
Learning rate parameter for optimization.

Momentum (*Number*, bounds=(0, None), default=0.5)
Stochastic Gradient Descent momentum

Nesterov (*Boolean*, bounds=(0, 1), default=True)
Use Nesterov type of momentum for SGD

Epochs (*Integer*, bounds=(1, None), default=100000)
Maximum number of epochs (iterations.

Regularizer 1 (peptide axis) (*Number*, bounds=(0, None), default=0.05)
Value of the regularizer along residue axis.

Regularizer 2 (sample axis) (*Number*, bounds=(0, None), default=0.5)
Value of the regularizer along sample axis.

Fit name (*String*, default='Gibbs_fit_1')
Name for for the fit result

Do Fitting (*Action*)
Start global fitting

class pyhdx.web.controllers.**GraphControl**(*parent*, ***params*)

Graph Control

Spin (*Boolean*, bounds=(0, 1), default=False)
Spin the protein object

State name (*Selector*, options=[])
Name of the currently selected state

Fit id (*Selector*, options=[])
Name of the currently selected fit ID

Peptide index (*Selector*, options=[])
Index of the currently selected peptide

class pyhdx.web.controllers.**ProteinControl**(*parent*, ***params*)

Protein Control

Input mode (*Selector*, default='PDB File', options=['PDB File', 'RCSB Download'])
Method of protein structure input

File binary (*Parameter*)

Rcsb id (*String*, default='')
RCSB ID of protein to download

Load structure (*Action*)

class pyhdx.web.controllers.**ClassificationControl**(*parent*, ***param*)

Classification

This controller allows users classify 'mapping' datasets and assign them colors.

Coloring can be either in discrete categories or as a continuous custom color map.

Target table (*Selector*, options=[])

Mode (*Selector*, default='Discrete', options=['Discrete', 'Continuous', 'Color map'])
Choose color mode (interpolation between selected colors).

Number of colours (*Integer*, bounds=(1, 10), default=3)
Number of classification colors.

Library (*Selector*, default='matplotlib', options=['matplotlib', 'colorcet'])

Color map (*Selector*, options=[])

Otsu (*Action*)
Automatically perform thresholding based on Otsu's method.

Linear (*Action*)
Automatically perform thresholding by creating equally spaced sections.

Log space (*Boolean*, bounds=(0, 1), default=False)
Boolean to set whether to apply colors in log space or not.

No coverage (*Color*, default='#8c8c8c')
Color to use for regions of no coverage

Color set name (*String*, default='')
Name for the color dataset to add

Add colorset (*Action*)

class pyhdx.web.controllers.**FileExportControl**(*parent*, ***param*)

File Export

This controller allows users to export and download datasets.

All datasets can be exported as .txt tables. 'Mappable' datasets (with r_number column) can be exported as .pml pymol script, which colors protein structures based on their 'color' column.

Target dataset (*Selector*, options=[])
Name of the dataset to export

Export format (*Selector*, default='csv', options=['csv', 'pprint'])
Format of the exported tables.'csv' is machine-readable, 'pprint' is human-readable format

CHAPTER

SEVEN

HISTORY

7.1 0.1.0 (2019-09-06)

- First release on PyPI.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

T

U

W

X

Z