PyHDX

Jochem Smit

Apr 23, 2022

CONTENTS:

1 Introduction **1**

2 Installation **3**
- 2.1 Stable release 3
- 2.2 Install from source 3
- 2.3 Running the web server 4
- 2.4 Install from source 4
- 2.5 Dependencies 5

3 PyHDX web application **7**
- 3.1 Peptide Input 7
- 3.2 Coverage 8
- 3.3 Initial Guesses 8
- 3.4 G Fit 9
- 3.5 Differential HDX 10
- 3.6 Color Transform 10
- 3.7 Protein Control 10
- 3.8 Graph Control 11
- 3.9 File Export 11
- 3.10 Figure Export 11
- 3.11 Session Manager 11

4 Examples **13**
- 4.1 PyHDX basics 13
- 4.2 Fitting of Gs 17

5 Citing and Resources **27**
- 5.1 Citing dependencies 27
- 5.2 Publications using PyHDX 28
- 5.3 Resources 28

6 Module Documentation **29**
- 6.1 Models 29
- 6.2 Fitting 38
- 6.3 Fitting PyTorch 42
- 6.4 FileIO 44
- 6.5 Output 46
- 6.6 Support 47

7 Web Application Reference **49**
- 7.1 Main Application 49

8 Indices and tables 59

Python Module Index 61

Index 63

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 (G) at the residue level.

An interactive web server is available where users can upload HDX-MS data and obtain G 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

Currently the recommended version to use is the latest beta release (v0.4.0bx)

2.1 Stable release

Installation with *conda*:

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

Installation with *pip*:

```
$ pip install pyhdx
```

To install with web application:

```
$ pip install pyhdx[web]
```

To install with pdf output:

```
$ pip install pyhdx[pdf]
```

2.2 Install from source

Create a new conda environment:

```
$ conda create --name py38_pyhdx python=3.8
# conda activate py38_pyhdx
```

Clone the github repository:

```
$ git clone https://github.com/Jhsmit/PyHDX
$ cd PyHDX
```

Generate conda requirements files from *setup.cfg*:

```
$ python _requirements.py
```

Install the base dependencies and optional extras. For example, to install PyHDX with web app:

```
$ conda install --file _req-base.txt --file _req-web.txt
```

To run the web application:

```
$ python pyhdx/web/serve.py
```

This runs the pyhx web application without a Dask cluster to submit jobs to, so submitting a fitting job will give an error.

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

```
python local_cluster.py
```

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 serve --scheduler_address <ip>:<port>
```

If no dask cluster is found at the specified address, a LocalCluster will be started (on localhost) using the specified port number.

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 Install from source

Clone the github repository:

```
$ git clone https://github.com/Jhsmit/PyHDX
$ cd PyHDX
```

You can use one of the files in ‘dev/deps/pinned’ to install a pretested set of pinned dependencies.

With *pip*:

```
$ pip install -r dev/deps/pinned/py38_windows_pip.txt
```

Or use ‘py38_linux_pip.txt’, which should be the same.

With *conda*:

```
$ conda env create -f dev/deps/pinned/py38_windows_conda.yml
```

Otherwise, you try your luck with the latest versions of the dependencies. If you would like a specific PyTorch version to use with PyHDX (ie CUDA/ROCm support), you should install this first. Installation instructions are on the Pytorch website.

Then, install the other base dependencies and optional extras.

Create a new conda environment:

```
$ conda create --name py38_pyhdx python=3.8
# conda activate py38_pyhdx
```

To install all dependencies:

```
$ conda install --file req-all.txt
```

Or choose which extras to install by using the 'req-<extra>.txt' files.

Install PyHDX in develop/editable mode

```
conda develop .
```

To run the web application:

```
$ python pyhdx/web/serve.py
```

This runs the pyhx web application without a Dask cluster to submit jobs to, so submitting a fitting job will give an error.

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

```
python local_cluster.py
```

2.5 Dependencies

The requirements for PyHDX and its extras 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

Peptide d-uptake data can be added into the web application either in 'Batch' or 'Manual' mode. Use *Input Mode* to switch between input modes.

When using 'Batch', multiple measurements can be added quickly through an *yaml* file specification. An example of an *yaml* file can be found in 'tests/test-data' on GitHub.

When using 'Single', each measurement and experimental metadata must be input manually. 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 in the field *Deuterium percentage*. This percentage should be as high as possible, typically >90%.

Use the fields *Temperature (K)* and *pH read* to specify the temperature and pH at which the D-labelling was done. The pH is the value 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

The 'Coverage' figure in the main application area rectangles 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.

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

- peptide_id: Index of the peptide per timepoint starting at the first peptide at 0
- start, end: Inclusive, exclusive interval of residue numbers in this peptide (Taking N-terminal resiudues into account)
- RFU: Relative fraction uptake of the peptide
- D(corrected): Absolute D-uptake, corrected by FD control
- 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 (*Fitting model*): 'Half-life' (fast but less accurate), or 'Association' (slower but more accurate).

Using the 'Association' procedure is recommended. This model fits two time constants the the weighted-averaged uptake kinetics of each residue. At *Lower bound* and *Upper bound* the bounds of these rate constants can be specified but in most cases the autosuggested bounds are sufficient. The bounds can be changed per dataset by using the *Dataset* field or for all datasets at the same time by ticking the *Global bounds* checkbox. Rarely issues might arise when the initial guess rates are close to the specified bounds at which point the bounds should be moved to contain a larger interval. This can be checked by comparing the fitted rates *k1* and *k2* (*File Export* → *Target dataset* → *rates*) Both rates and associated amplitudes are converted to a single rate value used for initial guesses. To calcualte 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 G Fit

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 **`_ACS`_**).

At *Initial guess*, select which dataset to use for initial guesses (typically 'Guess_1'). Both previous fits (G values) or estimated HX rates can be used as initial guesses. The initial guesses can be applied as 'One-to-one', where each protein state gets initial guesses derived from that state, or 'One-to-many', where one protein state is use as initial guesses for all states. Users can switch between both modes using *Guess mode*.

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 G 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 *Regualizer 2* control the magnitude of the regualizers. Please refer to our **`_ACS`_** publication 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 (Gs 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 G differences (G). 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 Gs). This can lead to artefactual differences in the final G result, as features might be resolved in out dataset and not in the other, which will show up as G. The penalty from *r2* can be calculated either with respect to a selected reference state (

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 when processing multiple protein states with *Fit mode* set to 'Single'. 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 G values are shown in the 'G' graph.

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

3.5 Differential HDX

This control panel can be used to generate differential HDX datasets. Select the fit to use with *Fit_ID*, then choose which state should be the reference state with *Reference state*. Assign a name to the new comparison and then click *Add comparison* to calculate G values. The values are calculated by taking each state and subtracting the reference from them (Test - Reference). Therefore if the test if more flexible (lower G) compared to the test, G value are negative and appear on the top of the G figure, by default colored green. Rigids parts are colored purple and are on the bottom of the graph. (note that the y axis is inverted as for the G figure) When adding a comparison, RFU values are automatically calculated, independent of the selected *Fit_ID*

3.6 Color Transform

The color transform panel can be used to update color transforms for each data quantity (rfu, drfu, dG, ddG). Select which quantity to update with *Target Quantity*. When selecting data quantities, the name of the current color map is shown below the selector.

Mode can be used to select between the available color modes; *Colormap*, *Continuous* and *Discrete*. *Discrete* splits the G 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 the PyHDX defaults, user defined color maps, or from `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 data values. A color for residues which are covered by peptides can be chosen at *No coverage*.

Assign an unique name using *Color transform name* and press *Update color transform* to create or update the color transform.

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 Control

Selected datasets can be directly visualized on a protein structure using the built in PDBeMolStar protein viewer. Use the selector *Input mode* to either directly download a PDB file from the RCSB PDB (specify *Pdb id*) or to upload a local .pdb file from your computer.

The *Table* selector can be used to choose which of the data tables to use to assign colors to the 3D structure (RFU, RFU, G or G values). *Visual Style* and *Lighting* can be used to tweak the appearance.

Use the buttons and menu on the protein viewer itself to export the current image to .png format.

3.8 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 and *exposure* to switch between exposure times. The selector *peptide_id* is used to choose which peptide uptake curve and corresponding fit to show in the Peptide graph. All corresponding graphs and selector options will update when changing these settings, including the protein view.

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 'Peptide MSE' shows the total mean squared error of all timepoints per peptide. The color scale adjust automatically so yellow colors do not necessarily reflect a poor fit, but highlight 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*.

3.9 File Export

All tables which underlie the graphs in the PyHDX web application can be downloaded directly. Choose the the desired dataset 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 dataset with an assigned color transform, 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, or as .csv/.txt files with hexadecimal color codes.

3.10 Figure Export

This panel can be used to export publication quality figures of G or G values. Figure options are scatterplot, linear bars or rainbowclouds and export filetypes can be .png, .pdf, .svg or .eps.

Use the selector *Reference* to set a reference state. This will then export the figures with G values. If set to *None*, figures are exported with G values.

Some parameters of the output figure format (number of columns, aspect ratio, figure width) can be tuned before generating the figure.

3.11 Session Manager

From here .zip files can be downloaded which contain all underlying data tables used in the current view. Click *Export session* to generate the .zip file. This file can then later be uploaded to recover the current session. At the moment, this only reproduces the data in the figures. It is not possible to calculate additional G fits after reloading a session. However, exporting figures is possible. Use *Browse*, select your PyHDX session .zip file and click *Load session* to reload your session.

The button *Reset session* can be used to clear all data. But its probably better to just use the refresh button in the browser (F5).

CHAPTER

FOUR

EXAMPLES

4.1 PyHDX basics

[10]:
```
from pyhdx import PeptideMasterTable, read_dynamx, HDXMeasurement
from pyhdx.plot import peptide_coverage
import matplotlib.pyplot as plt
import proplot as pplt
from pathlib import Path
```

We can use the `read_dynamx` function to read the input DynamX state file. Exposure times in the .csv files are in the field exposure, and we specify the unit with the `time_unit` keyword argument. The exposure time units are converted to seconds.

This function returns a `numpy` structured array where each entry corresponds to one peptide, in this example 567 peptides.

[3]:
```
fpath = Path() / '..' / '..' / 'tests' / 'test_data' / 'input' / 'ecSecB_apo.csv'
data = read_dynamx(fpath, time_unit='min')
data.size
```

[3]:
```
9072
```

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.

[4]:
```
master_table = PeptideMasterTable(data, drop_first=1, ignore_prolines=True)
master_table.data['ex_residues'][:50]
```

[4]:
```
0       8.0
2       8.0
1       8.0
3       8.0
4       8.0
5       8.0
6       8.0
7       8.0
8       8.0
```

(continues on next page)

(continued from previous page)

```
9      6.0
11     6.0
10     6.0
12     6.0
13     6.0
14     6.0
15     6.0
16     6.0
17     6.0
18    12.0
20    12.0
19    12.0
21    12.0
22    12.0
23    12.0
24    12.0
25    12.0
26    12.0
27    13.0
29    13.0
28    13.0
30    13.0
31    13.0
32    13.0
33    13.0
34    13.0
35    13.0
36    14.0
38    14.0
37    14.0
39    14.0
40    14.0
41    14.0
42    14.0
43    14.0
44    14.0
45    20.0
47    20.0
46    20.0
48    20.0
49    20.0
Name: ex_residues, dtype: float64
```

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.

[5]:
```
master_table.set_control(('Full deuteration control', 0.167*60))
master_table.data['uptake'][:50]
```

[5]:
```
0      0.000000
1      0.000000
```

(continues on next page)

(continued from previous page)

```
2      5.073400
3      2.486444
4      2.857141
5      3.145738
6      3.785886
7      4.082950
8      4.790625
9      0.000000
10     0.000000
11     3.642506
12     1.651437
13     1.860919
14     2.107151
15     2.698036
16     2.874801
17     3.449561
18     0.000000
19     0.000000
20     5.264543
21     1.839924
22     2.508343
23     2.969332
24     3.399092
25     3.485568
26     4.318144
27     0.000000
28     0.000000
29     6.317900
30     2.532099
31     3.306167
32     3.996718
33     4.389410
34     4.379495
35     5.283969
36     0.000000
37     0.000000
38     6.812215
39     3.119850
40     3.874881
41     4.342807
42     4.854057
43     4.835639
44     5.780219
45     0.000000
46     0.000000
47    10.815100
48     5.432395
49     6.131800
Name: uptake, dtype: float64
```

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.

[6]:
```
master_table.states
state_data = master_table.get_state('SecB WT apo')
state_data.size
```

[6]:
```
10584
```

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 G values. The pH values are uncorrected values are measured by the pH meter (ie p(H, D) values)

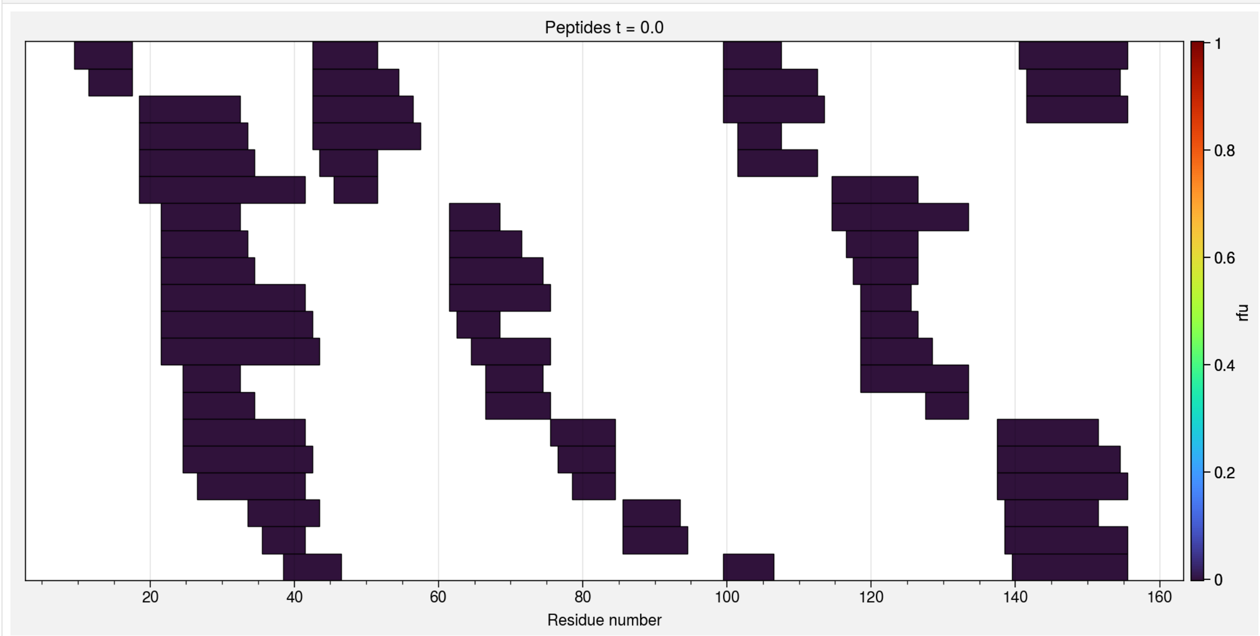
[7]:
```
hdxm = HDXMeasurement(state_data, temperature=303.15, pH=8., name='My HDX measurement')
type(hdxm), len(hdxm), hdxm.timepoints, hdxm.name, hdxm.state
```

[7]:
```
(pyhdx.models.HDXMeasurement,
 7,
 array([   0.        ,   10.02      ,   30.        ,   60.        ,  300.        ,
         600.        , 6000.00048]),
 'My HDX measurement',
 'SecB WT apo')
```

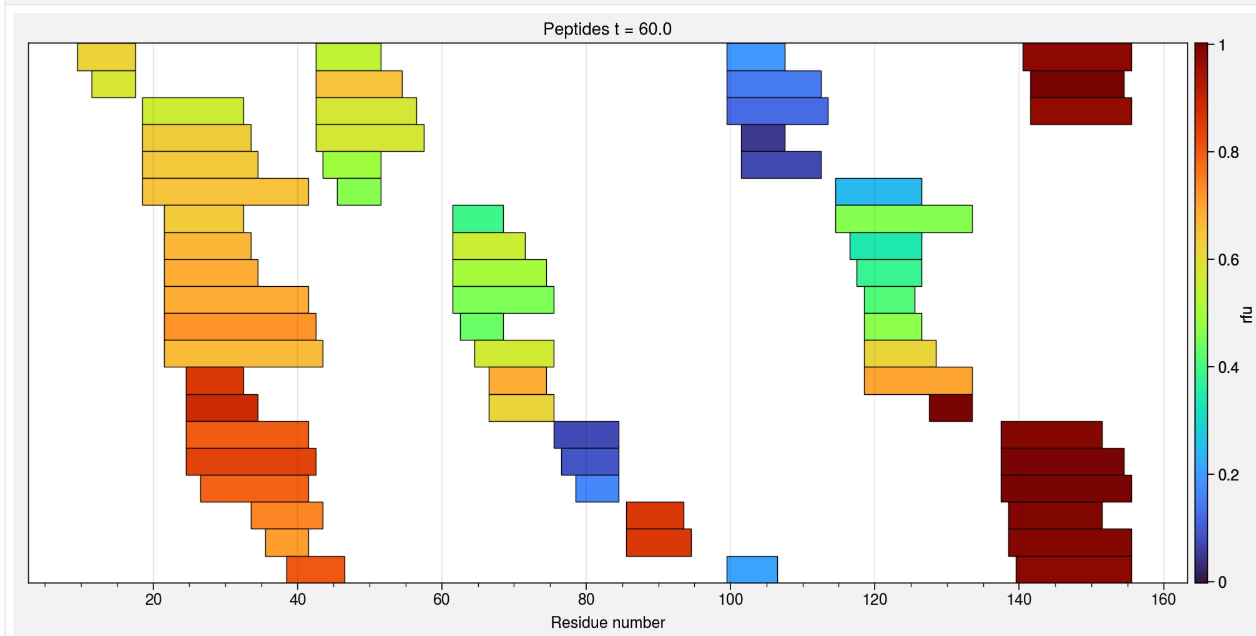
Iterating over a HDXMeasurement object returns a set of HDXTimepoint each with their own attributes describing the topology of the coverage. When creating the object, peptides which are not present in all timepoints are removed, such that all timepoints and HDXTimepoint have identical coverage.

Note that the internal time units in PyHDX are seconds.

[17]:
```
fig, ax = pplt.subplots(figsize=(10, 5))
i = 0
peptide_coverage(ax, hdxm[i].data, 20, cbar=True)
t = ax.set_title(f'Peptides t = {hdxm.timepoints[i]}')
l = ax.set_xlabel('Residue number')
```



[18]:
```
fig, ax = pplt.subplots(figsize=(10, 5))
i = 3
peptide_coverage(ax, hdxm[i].data, 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.

[19]:
```
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 Gs

[1]:
```
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.

```
[2]: 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*60))
     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)

```
[3]: 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, :]

[3]: <pyhdx.models.Protein at 0x196c5b28640>
```

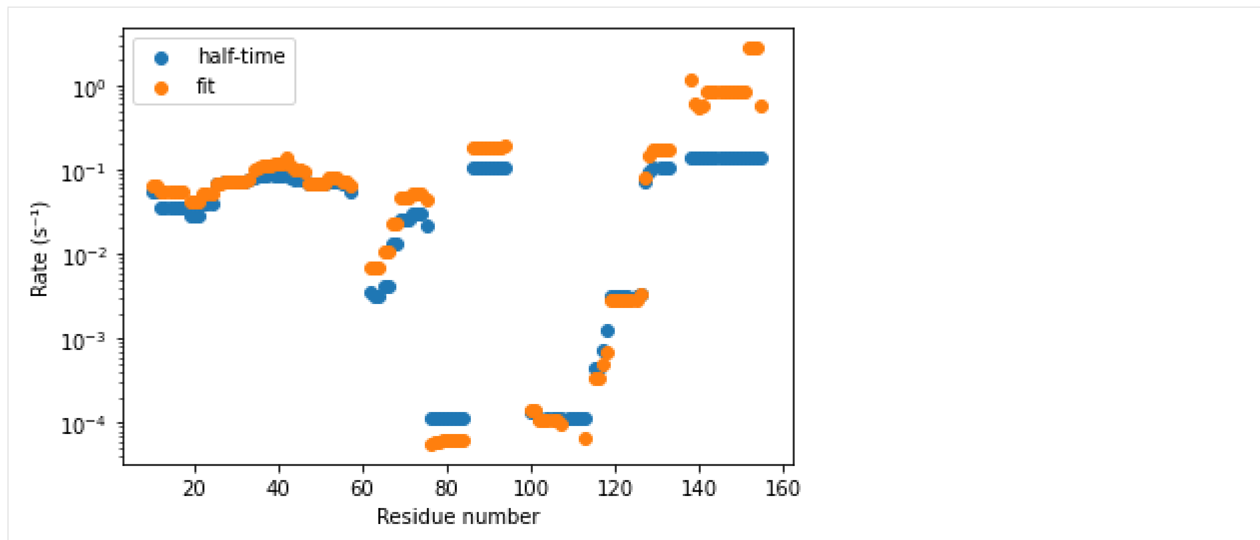
A more accurate result can be obtained by fitting the per-residue/timepoint RFU values to a biexponential association curve. This process is more time consuming and can optionally be processed in parallel by Dask.

```
[4]: 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 (units s^{-1}) can be extracted, as well as other fit parameters, from this object:

```
[5]: fr_wt_avg.output

C:\Users\jhsmi\pp\PyHDX\pyhdx\fit_models.py:224: RuntimeWarning: overflow encountered in␣
↪exp
  t = np.exp(t_log)

[5]: <pyhdx.models.Protein at 0x196cf9544c0>
```

```
[6]: 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')
     ax.set_ylabel('Rate (s¹)')
     ax.legend()

[6]: <matplotlib.legend.Legend at 0x196d1e45550>
```



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

[7]:
```
gibbs_guess = hdxm.guess_deltaG(fr_wt_avg.output['rate'])
gibbs_guess
```

[7]:
```
r_number
10     19765.736199
11     19301.237211
12     20635.398598
13     16860.539218
14     18951.649654
           ...
151    11573.754525
152     9064.889494
153    11502.744868
154    11783.079284
155     1390.996025
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.

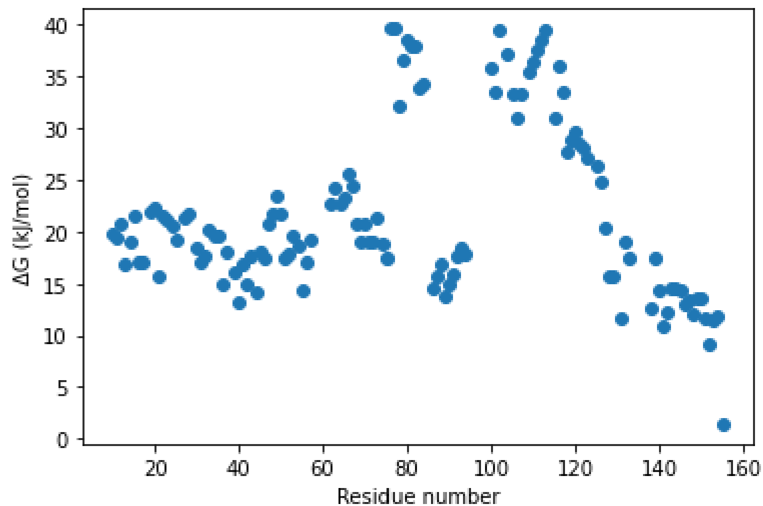
[8]:
```
gibbs_result = fit_gibbs_global(hdxm, gibbs_guess, epochs=1000)
gibbs_output = gibbs_result.output
gibbs_output
```

```
100%|| 1000/1000 [00:00<00:00, 1501.50it/s]
```

[8]:
```
<pyhdx.models.Protein at 0x196d172b310>
```

Along with G the fit result returns covariances of G and protection factors. The column `k_obs` is the observed rate of exchange without taking into account the intrinsic exchange rate of each residue. If users want to obtain a result truly independent of the intrinsic rate of exchange, the regularization value r1 should be set to zero (as this works on G, which is obtained by taking intrinsic rate of exchange into account) and users should provide their own initial guesses for G (as determination of initial guesses also uses intrinsic rates of exchange).

[9]:
```
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('G (kJ/mol)')
None
```



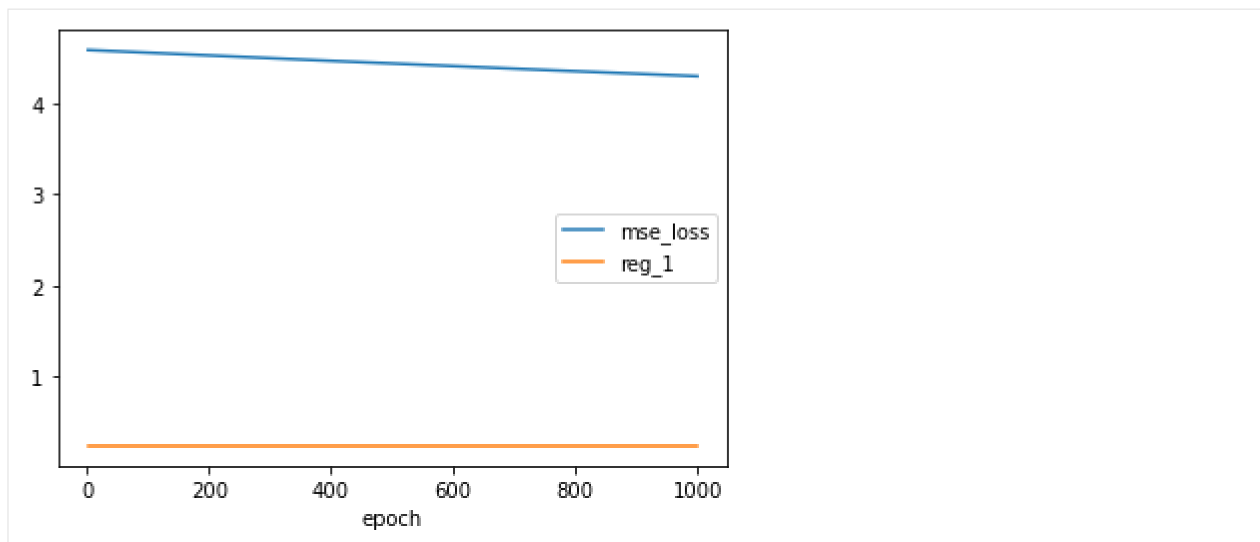
4.2.1 Number of epochs and overfitting

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 decrease sharply and then flatten out.

[10]:
```
plt.figure()
gibbs_result.losses.plot()
```

[10]:
```
<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_1 is the loss resulting from the regualizer.

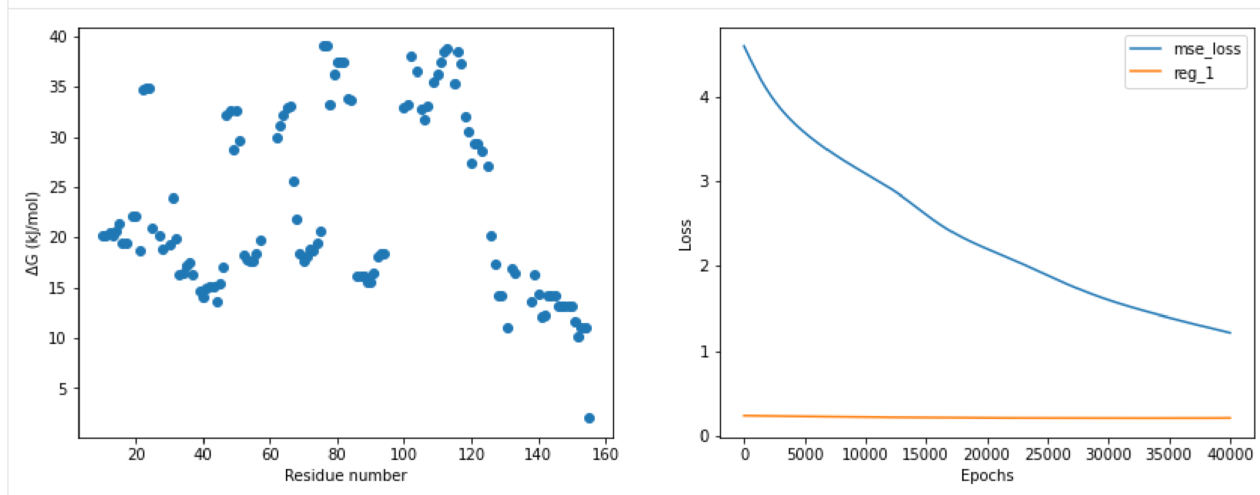
If the losses do not decrease, this is likely due to a too low number of epochs or a too low learning rate. Lets tune the fit parameters such that we obtain the desired result.

[11]:
```
gibbs_result_updated = fit_gibbs_global(hdxm, gibbs_guess, epochs=40000)
```

```
100%|| 40000/40000 [00:27<00:00, 1455.32it/s]
```

[12]:
```
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('G (kJ/mol)')
gibbs_result_updated.losses.plot(ax=axes[1])
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss')
```

[12]:
```
Text(0, 0.5, 'Loss')
```



By increasing the learning rate and the number of epochs, our result improves as the final MSE loss is lower.

With `stop_loss` at 1E-6 and `patience` at 50 (=default), the fitting will not terminate until for 50 epochs the progress (loss decrease) has been less than 1E-6.

The default value of `stop_loss` is 1E-5. This will ensure optimization will stop when not enough progress is being made and thereby prevent overfitting.

Users can keep track of the G values per epoch by using `Checkpoint` callbacks (See templates/04; advanced/experimental usage).

[13]:
```
gibbs_result_updated = fit_gibbs_global(hdxm, gibbs_guess, epochs=60000, lr=1e5, stop_
↪loss=1e-6, patience=50)
```

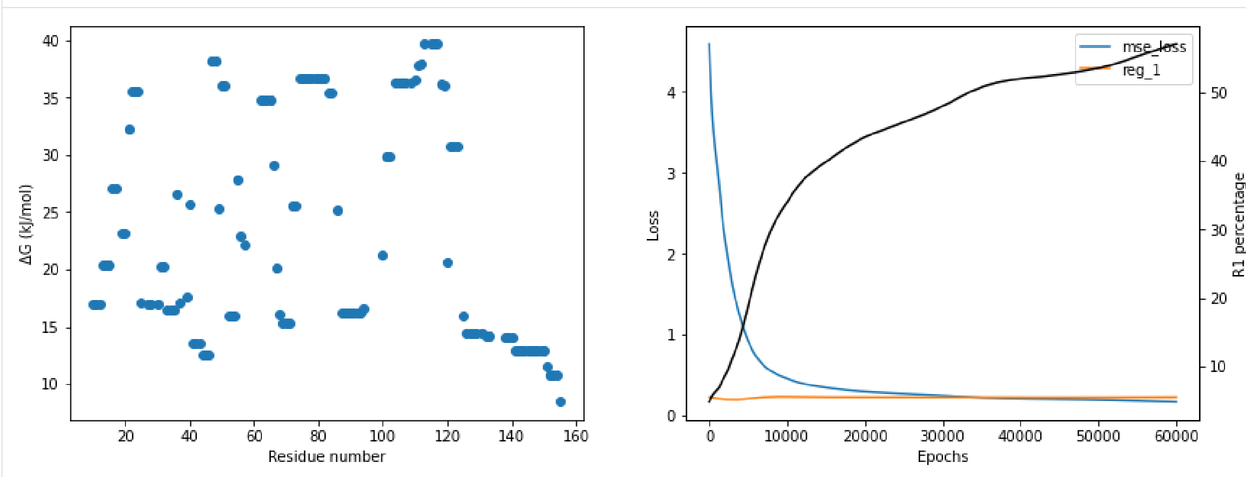
```
100%|| 60000/60000 [00:43<00:00, 1384.86it/s]
```

[14]:
```
gibbs_result_updated.regularization_percentage
```

[14]:
```
57.09043167835134
```

[15]:
```
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('G (kJ/mol)')
gibbs_result_updated.losses.plot(ax=axes[1])
ax = axes[1].twinx()
ax.set_ylabel('R1 percentage')
percentage = 100 * gibbs_result_updated.losses['reg_1'] / (gibbs_result_updated.losses.
↪sum(axis=1))
ax.plot(percentage, color='k')
axes[1].set_xlabel('Epochs')
axes[1].set_ylabel('Loss')
```

[15]:
```
Text(0, 0.5, 'Loss')
```



[16]:
```
gibbs_result_updated.losses.sum(axis=1)
```

[16]:
```
epoch
1        4.823331
2        4.821007
3        4.818298
```

(continues on next page)

(continued from previous page)

```
4        4.815398
5        4.812404
           ...
59996    0.406689
59997    0.406686
59998    0.406685
59999    0.406682
60000    0.406680
Length: 60000, dtype: float64
```

With these settings, the losses and the result become highly influenced by the regularizer `r1`, which dampens the result and removes scatter in G values.

4.2.2 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 G 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 exchange rates (G) 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 0.1, 2 and 5. In this dataset, despite the fact that for `r1=2` contribution `reg_1` is 50% of `total_loss`, most features in G are still resolved. In this case, it is recommended to try different values of `r1` (starting low and increasing) and find the optimal value based on the G result and fit metrics (Total mse loss at fit termination,

$$\chi^2$$

per peptide and fit curves per peptide (available in web interface)

[17]:
```
r1_vals = [0.1, 2, 5]
results_dict = {}
for r1 in r1_vals:
    print(r1)
    result = fit_gibbs_global(hdxm, gibbs_guess, epochs=60000, lr=1e5, stop_loss=1e-5,␣
↪patience=50, r1=r1)
    results_dict[r1] = result
```

```
 32%|      | 19147/60000 [00:13<00:29, 1406.90it/s]
 39%|      | 23147/60000 [00:16<00:26, 1385.27it/s]
100%|| 60000/60000 [00:44<00:00, 1335.27it/s]
```

```
0.1
2
5
```

[18]:
```
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)
```

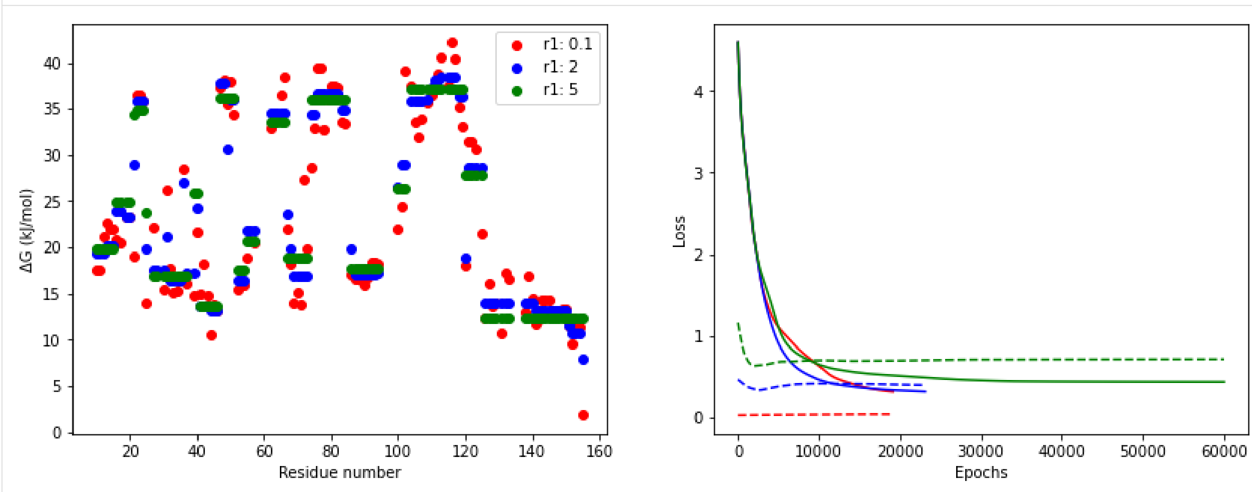
(continues on next page)

(continued from previous page)

```
    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('G (kJ/mol)')
    axes[1].plot(v.losses['mse_loss'], color=color)
    axes[1].plot(v.losses['reg_1'], color=color, linestyle='--')
    axes[1].set_xlabel('Epochs')
    axes[1].set_ylabel('Loss')

axes[0].legend()
```

```
10.09466832487272
55.56463704204826
62.118677400826414
```

[18]: <matplotlib.legend.Legend at 0x196d20b48b0>



4.2.3 Fit result covariances

Covariances on G are estimated from the Hessian matrix

$$\mathcal{H}$$

. This matrix describes the local curvature of

$$(sumofsquaredresiduals)ateachresidueforthesetofGvaluesobtainedfromthefit.FromtheHessianmatrix,covariancesared$$

$$\sqrt{|(-\mathcal{H})_{ii}^{-1}|}$$

).

When these covariances are high, the optimization landscape is flat and therefore it presents a higher difficulty to finding the exact minimum. High covariances frequently signal that the obtained G values are at the extreme ends of the time resolution of the experiment. In these regions, the G values found represent an upper or lower limit and in order to improve the G range of the experiment shorter or longer timepoints need to be added.

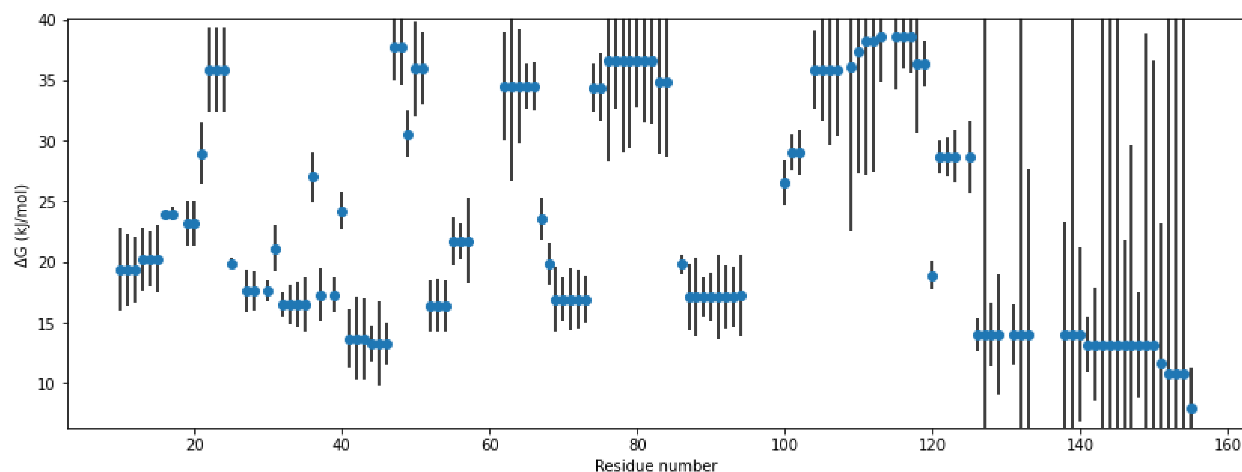
Covariances are stored in the fit result and we can plot these on top of the G values to see which regions are within the range of the experiment.

[19]:

```
colors = iter(['r', 'b', 'g'])
fig, ax = plt.subplots(ncols=1, figsize=(14, 5))
v = results_dict[2]

ax.scatter(v.output.index, v.output['deltaG']*1e-3)
ylim = ax.get_ylim()
ax.errorbar(v.output.index, v.output['deltaG']*1e-3, yerr=v.output['covariance']*1e-3,␣
↪linestyle='none', color='k', zorder=-1)
ax.set_ylim(*ylim)
ax.set_xlabel('Residue number')
ax.set_ylabel('G (kJ/mol)')
```

[19]:

```
Text(0, 0.5, 'G (kJ/mol)')
```



In the graph above, covariances are high at SecB's disordered c-tail, highlighting that G values obtained here represent an upper limit where the actual G values are likely to be lower. Likewise, high G values also show high covariances which can be improved by adding longer D-exposure datapoints. Shorter D-exposure datapoints can be added in principle, but regions with low G are more likely to exchagne via EX1 kinetics, where PyHDX kinetics approximations break down.

[19]:

CHAPTER

FIVE

CITING AND RESOURCES

When using PyHDX in your research, please cite our publication:

- Jochem H. Smit, Srinath Krishnamurthy, Bindu Y. Srinivasu, Rinky Parakra, Spyridoula Karamanou, and Anastassios Economou. Probing Universal Protein Dynamics Using Hydrogen–Deuterium Exchange Mass Spectrometry-Derived Residue-Level Gibbs Free Energy. *Analytical Chemistry*, 93(38):12840–12847, September 2021. doi:10.1021/acs.analchem.1c02155.

5.1 Citing dependencies

When using the different modules offered by PyHDX, please consider citing the following papers/DOIs:

Intrinsic H/D exchange rates:

- Yawen Bai, John S. Milne, Leland Mayne, and S. Walter Englander. Primary structure effects on peptide group hydrogen exchange. *Proteins: Structure, Function, and Bioinformatics*, 17(1):75–86, 1993. doi:10.1002/prot.340170110.
- Gregory P. Connelly, Yawen Bai, Mei-Fen Jeng, and S. Walter Englander. Isotope effects in peptide group hydrogen exchange. *Proteins: Structure, Function, and Genetics*, 17(1):87–92, September 1993. doi:10.1002/prot.340170111.
- Susumu Mori, Peter C. M. van Zijl, and David Shortle. Measurement of water–amide proton exchange rates in the denatured state of staphylococcal nuclease by a magnetization transfer technique. *Proteins: Structure, Function, and Bioinformatics*, 28(3):325–332, 1997. doi:10.1002/(SICI)1097-0134(199707)28:3<325::AID-PROT3>3.0.CO;2-B.
- David Nguyen, Leland Mayne, Michael C. Phillips, and S. Walter Englander. Reference Parameters for Protein Hydrogen Exchange Rates. *Journal of the American Society for Mass Spectrometry*, 29(9):1936–1939, September 2018. doi:10.1021/jasms.8b05911.

PDBe MolStar protein viewer:

- David Sehnal, Sebastian Bittrich, Mandar Deshpande, Radka Svobodová, Karel Berka, Václav Bazgier, Sameer Velankar, Stephen K Burley, Jaroslav Koča, and Alexander S Rose. Mol* Viewer: modern web app for 3D visualization and analysis of large biomolecular structures. *Nucleic Acids Research*, 49(W1):W431–W437, July 2021. doi:10.1093/nar/gkab314.

5.2 Publications using PyHDX

The following publications use PyHDX:

- Srinath Krishnamurthy, Nikolaos Eleftheriadis, Konstantina Karathanou, Jochem H. Smit, Athina G. Portaliou, Katerina E. Chatzi, Spyridoula Karamanou, Ana-Nicoleta Bondar, Giorgos Gouridis, and Anastassios Economou. A nexus of intrinsic dynamics underlies translocase priming. *Structure*, 29(8):846–858.e7, August 2021. doi:10.1016/j.str.2021.03.015.
- Srinath Krishnamurthy, Marios-Frantzeskos Sardis, Nikolaos Eleftheriadis, Katerina E. Chatzi, Jochem H. Smit, Konstantina Karathanou, Giorgos Gouridis, Athina G. Portaliou, Ana-Nicoleta Bondar, Spyridoula Karamanou, and Anastassios Economou. Preproteins couple the intrinsic dynamics of SecA to its ATPase cycle to translocate via a catch and release mechanism. *Cell Reports*, February 2022. doi:10.1016/j.celrep.2022.110346.
- Biao Yuan, Athina G. Portaliou, Rinky Parakra, Jochem H. Smit, Jiri Wald, Yichen Li, Bindu Srinivasu, Maria S. Loos, Harveer Singh Dhupar, Dirk Fahrenkamp, Charalampos G. Kalodimos, Franck Duong van Hoa, Thorben Cordes, Spyridoula Karamanou, Thomas C. Marlovits, and Anastassios Economou. Structural Dynamics of the Functional Nonameric Type III Translocase Export Gate. *Journal of Molecular Biology*, 433(21):167188, October 2021. doi:10.1016/j.jmb.2021.167188.

5.3 Resources

Other PyHDX resources:

Paper code: https://github.com/Jhsmit/PyHDX-paper

BioRxiv v1: https://www.biorxiv.org/content/10.1101/2020.09.30.320887v1

BioRxiv v2: https://www.biorxiv.org/content/10.1101/2020.09.30.320887v2

General HDX-MS resources:

HDX-MS-resources: https://github.com/hadexversum/HDX-MS-resources

HDX-MS datasets (alpha):

HDX-MS-datasets: https://github.com/Jhsmit/HDX-MS-datasets

CHAPTER SIX

MODULE DOCUMENTATION

This page contains the full API docs of PyHDX

6.1 Models

class pyhdx.models.**Coverage**(*data*, *n_term=1*, *c_term=None*, *sequence=None*)

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** – DataFrame with input peptides
- **n_term** – Residue index of the N-terminal residue. Default value is 1, can be negative to accomodate for N-terminal purification tags
- **c_term** – Residue index number of the C-terminal residue (where first residue has index number 1)
- **sequence** – 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.

property Np: int

Number of peptides.

Return type int

property Nr: int

Total number of residues spanned by the peptides.

Return type int

X: numpy.ndarray

Np x Nr matrix (peptides x residues). Values are 1 where residue j is in peptide i.

property X_norm

X coefficient matrix normalized column wise.

Type ndarray

Z: numpy.ndarray

Np x Nr matrix (peptides x residues). Values are 1/(ex_residues) where residue j is in peptide i.

property Z_norm

Z coefficient matrix normalized column wise.

Type ndarray

apply_interval(*array_or_series*)

Returns the section of *array_or_series* in the interval

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

Parameters **array_or_series** – Input data object to crop to interval

Returns Input object cropped to interval of the interval spanned by the peptides

Return type Series

property block_length: numpy.ndarray

Lengths of unique blocks of residues in the peptides map, along the *r_number* axis

Return type ndarray

Type ndarary

get_sections(*gap_size=- 1*)

Get the intervals of independent sections of coverage.

Intervals are inclusive, exclusive. Gaps are defined with *gap_size*, adjacent peptides with distances bigger than this value are considered not to overlap. Set to -1 to treat touching peptides as belonging to the same section.

Parameters **gap_size** – The size which defines a gap

property index: pandas.core.indexes.range.RangeIndex

Pandas index numbers corresponding to the part of the protein covered by peptides

Return type RangeIndex

property percent_coverage: float

Percentage of residues covered by peptides

Return type float

property r_number: pandas.core.indexes.range.RangeIndex

Pandas index numbers corresponding to the part of the protein covered by peptides

Return type RangeIndex

property redundancy: float

Average redundancy of peptides in regions with at least 1 peptide

Return type float

class pyhdx.models.CoverageSet(*hdxm_list*)

Coverage object for multiple *HDXMeasurement* objects.

This objects finds the minimal interval of residue numbers which fit all :class:`.HDXMeasurement`s

Parameters **hdxm_list** – List of input :class:`.HDXMeasurment objects.

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

`get_masks()`

mask of shape NsxNr with True entries covered by hdx measurements (exluding gaps)

property index: pandas.core.indexes.range.RangeIndex

Index of residue numbers

Return type `RangeIndex`

property s_r_mask: numpy.ndarray

Sample-residue mask

Boolean array where entries *ij* are `True` if residue *j* is covered by peptides of sample *i* (Coverage aps not taken into account)

Return type `ndarray`

class pyhdx.models.HDXMeasurement(*data*, ***metadata*)

Main HDX data object.

This object has peptide data of a single state and 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** – Dataframe with all peptides belonging to a single state.
- ****metadata** – Dictionary of optional metadata. By default, holds the *temperature* and *pH* parameters.

property Np: int

Number of peptides.

Return type `int`

property Nr: int

Total number of residues spanned by the peptides.

Return type `int`

property Nt: int

Number of timepoints.

Return type `int`

coverage: ***Coverage***

Coverage object describing peptide layout

property d_exp: pandas.core.frame.DataFrame

D-uptake values (corrected for back-exchange).

Shape of the returned DataFrame is Np (rows) x Nt (columns)

Return type `DataFrame`

data: pd.DataFrame

Dataframe with all peptides

`get_tensors`(*exchanges=False*, *dtype=None*)

Returns a dictionary of tensor variables for fitting HD kinetics.

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

Parameters

- **exchanges** – If `True` only returns tensor data describing residues which exchange (ie have peptides and are not prolines)
- **dtype** – Optional Torch data type. Use torch.float32 for faster fitting of large data sets, possibly at the expense of accuracy

Returns Dictionary with tensors

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

Obtain G initial guesses from apparent H/D exchange rates.

Units of rates are per second. As the intrinsic rate of exchange of the c-terminal residue is ~100 fold lower, guess values for PF and G are also much lower. Use the option *correct_c_term* to set the c-terminal guess value equal to the value of the residue preceding it.

Parameters

- **rates** – Apparent exchange rates (units s^-1). Series index is protein residue number.
- **correct_c_term** – If `True`, sets the guess value of the c-terminal residue to the value of the residue preceding it.

Returns G guess values (units kJ/mol)

Return type `Series`

property name: str

HDX Measurement name

Return type `str`

property pH: Optional[float]

pH of the H/D exchange reaction.

Return type `Optional[float]`

peptides: list[*HDXTimepoint*]

List of *HDXTimepoint*, one per exposure timepoint

property rfu_peptides: pandas.core.frame.DataFrame

Relative fractional uptake per peptide.

Shape of the returned DataFrame is Np (rows) x Nt (columns)

Return type `DataFrame`

property rfu_residues: pandas.core.frame.DataFrame

Relative fractional uptake per residue.

Shape of the returned DataFrame is Nr (rows) x Nt (columns)

Return type `DataFrame`

state: str

Protein state label for this HDX measurement

property temperature: Optional[float]

Temperature of the H/D exchange reaction (K).

Return type `Optional[float]`

timepoints: np.ndarray

Deuterium exposure times

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

Write the data in this `HDXMeasurement` to file.

Parameters

- **file_path** – File path to create and write to.
- **include_version** – Set `True` to include PyHDX version and current time/date
- **fmt** – Formatting to use, options are 'csv' or 'pprint'
- **include_metadata** – If `True`, the objects' metadata is included
- ****kwargs** – Optional additional keyword arguments passed to *df.to_csv*

Return type `None`

class pyhdx.models.HDXMeasurementSet(*hdxm_list*)

Set of multiple `HDXMeasurement` s

Parameters hdxm_list – Input list of `HDXMeasurement`

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

d_exp: numpy.ndarray

Array with measured D-uptake values, shape is Ns x Np x Nt, padded with zeros.

property exchanges: numpy.ndarray

Boolean mask `True` where there are residues which exchange

Shape of the returned array is Ns x Np

Return type `ndarray`

get(*name*)

find a HDXMeasurement by name

Return type `HDXMeasurement`

get_tensors(*dtype=None*)

Returns a dictionary of tensor variables for fitting HD kinetics.

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

Returns Dictionary with tensors

guess_deltaG(*rates_df*, ***kwargs*)

Obtain G initial guesses from apparent H/D exchange rates.

Parameters

- **rates_df** – Pandas dataframe apparent exchange rates (units s^-1). Column names must correspond to HDX measurement names.
- ****kwargs** – Additional keyword arguments passed to *HDXMeasurement.guess_deltaG()*

Returns G guess values (units kJ/mol)

Return type DataFrame

property rfu_residues: pandas.core.frame.DataFrame

Relative fractional uptake per residue.

Shape of the returned DataFrame is Nr (rows) x Ns*Nt (columns) and is multiindexed by columns (state, exposure, quantity)

Return type DataFrame

timepoints: numpy.ndarray

Array with timepoints, shape is Ns x Nt, padded with zeros in case of samples with unequal number of timepoints

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

Write the data in this *HDXMeasurementSet* to file.

Parameters

- **file_path** – File path to create and write to.
- **include_version** – Set True to include PyHDX version and current time/date
- **fmt** – Formatting to use, options are 'csv' or 'pprint'
- **include_metadata** – If True, the objects' metadata is included
- ****kwargs** – Optional additional keyword arguments passed to *df.to_csv*

Return type None

class pyhdx.models.**HDXTimepoint**(*data*, ***kwargs*)

Class with subset of peptides corresponding to only one state and exposure

Parameters

- **data** – Dataframe with input data
- ****kwargs** –

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 **rfu** – Array of rfu per peptide

Return type ndarray

Return type ndarray

property d_exp: pandas.core.series.Series

Experimentally measured D-values (corrected)

Return type Series

exposure: float

Deuterium exposure time for this HDX timepoint (units seconds)

property name: str

Name of this HDX timepoint

Format is <state>_<exposure>

Return type str

property rfu_peptides: pandas.core.series.Series

Relative fractional uptake per peptide

Return type Series

property rfu_residues: pandas.core.series.Series

Relative fractional uptake (RFU) per residue.

RFU values are obtained by weighted averaging, weight value is the length of each peptide

Return type Series

state: str

Protein state label for this HDX timepoint

weighted_average(*field*)

Calculate per-residue weighted average of values in data column

Parameters field – Data field (column) to calculated weighted average of

Returns THe weighted averaging result

Return type Series

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 DataFrame *data* must have the following entries 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: The time the sample was exposed to a deuterated solution. Units are seconds. 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*,** *ignore_prolines*, and *d_percentage* 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** – Pandas dataframe with peptide entries
- **drop_first** – Number of N-terminal amino acids to ignore
- **d_percentage** – Percentage of deuterium in the labelling solution
- **ignore_prolines** – Toggle ignoring of proline residues. Should always be set to `True`
- **sort** – Set to `True` to sort the input. Sort order is 'start', 'end', 'sequence', 'exposure', 'state'.
- **remove_nan** – Set to `True` to remove `NaN` entries in the 'uptake' column

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** – DataFrame with selected peptides

Return type `DataFrame`

get_state(*state*)

Returns entries in the table with state 'state' Rows with *NaN* entries for 'uptake_corrected' are removed

Parameters **state** (`str`) – Name of the 'state' entries to select

Return type `DataFrame`

Returns Dataframe of peptides from specified 'state'

select(***kwargs*)

Select data based on column values.

Parameters **kwargs** (`dict`) – Column name, value pairs to select

Returns **output_data** – DataFrame with selected peptides

Return type `DataFrame`

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

Return type `None`

set_control(*control_1*, *control_0=None*)

Apply a control dataset to this object. The column 'RFU' is added to the object by normalizing its uptake value with respect to the control uptake value to one. Optionally, `control_zero` can be specified which is a dataset whose uptake value will be used to zero the uptake.

Nonmatching peptides are set to NaN

#todo insert math

Parameters

- **param** (control_0: tuple with (*state*, *exposure*) for peptides to use for zeroing uptake values (ND control)) –

- **param** –

property states

ndarray Array with unique states

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.

get_k_int(*temperature*, *pH*, ***kwargs*)

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 – Array of intrisic exchange rates

Return type ndarray

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*

Return type None

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

Find and return the intersecting entries in multiple arrays.

Parameters

- **arrays** – Iterable of input structured arrays
- **fields** – Iterable of fields to use to decide if entires are intersecting

Returns Output iterable of arrays with only intersecting entries.

Return type selected

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.

Peptides are supplied as `HDXMeasurement` objects. After the intersection of peptides is found, new objects are returned where all peptides (coverage, exposure) between the measurements are identical.

Optionally intersections by custom fields can be made.

Parameters

- **hdx_list** – Input list of `HDXMeasurement`
- **fields** – By which fields to take the intersections. Default is [‘_start’, ‘_end’, ‘exposure’]

Returns Output list of `HDXMeasurement`

Return type hdx_out

6.2 Fitting

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

chi_squared

Alias for field number 0

params

Alias for field number 1

class pyhdx.fitting.**GenericFitResult**(*output*, *fit_function*, *name*)

class pyhdx.fitting.**KineticsFitResult**(*hdxm*, *intervals*, *results*, *models*)

Fit result object. Generally used for initial guess results.

Parameters

- **hdxm** (`HDXMeasurement`) – HDX measurement object to fit
- **intervals** (`list`) – List of tuples with intervals (inclusive, exclusive) describing which residues *results* and *models* refer to
- **results** (`list`) – List of `FitResults`
- **models** (`list`) – Lis of `KineticsModel`

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** – Array with parameter values

Return type `ndarray`

property output

Dataframe with fitted rates per residue

Type Dataframe

property rate

Returns an array with the exchange rates

property tau

Returns an array with the exchange rates

class pyhdx.fitting.**RatesFitResult**(*results*)

Accumulates multiple Generic/KineticsFit Results

pyhdx.fitting.**check_bounds**(*fit_result*)

Check if the obtained fit result is within bounds

pyhdx.fitting.**fit_gibbs_global**(*hdxm, initial_guess, r1=1, epochs=200000, patience=50, stop_loss=5e-06, optimizer='SGD', callbacks=None, **optimizer_kwargs*)

Fit Gibbs free energies globally to all D-uptake data in the supplied hdxm

Parameters

- **hdxm** (*HDXMeasurement*) – Input HDX measurement
- **initial_guess** (Series or ndarray) – Gibbs free energy initial guesses (shape Nr, units J/mol)
- **r1** (float) – Regularizer value r1 (along residues)
- **epochs** (int) – Maximum number of fitting iterations
- **patience** (int) – Number of epochs to wait until termination when progress between epochs is below *stop_loss*
- **stop_loss** (float) – Threshold for difference in loss between epochs when an epoch is considered to make no more progress.
- **optimizer** (str) – Which optimizer to use. Default is Stochastic Gradient Descent. See PyTorch documentation for information.
- **callbacks** (list or None) – List of callback objects. Call signature is callback(epoch, model, optimizer)
- ****optimizer_kwargs** – Additional keyword arguments passed to the optimizer.

Returns result

Return type TorchSingleFitResult

pyhdx.fitting.**fit_gibbs_global_batch**(*hdx_set, initial_guess, r1=1, r2=1, r2_reference=False, epochs=200000, patience=50, stop_loss=5e-06, optimizer='SGD', callbacks=None, **optimizer_kwargs*)

Fit Gibbs free energies globally to all D-uptake data in multiple HDX measurements

Parameters

- **hdx_set** (*HDXMeasurementSet*) – Input HDX measurements
- **initial_guess** (Series or DataFrame or ndarray) – Gibbs free energy initial guesses (shape Ns x Nr or Nr, units J/mol)
- **r1** (float) – Regularizer value r1 (along residues)

- **r2** (float) – Regularizer value r2 (along protein states/samples)
- **r2_reference** (bool:) – If *True* the first dataset is used as a reference to calculate r2 differences, otherwise the mean is used
- **epochs** (int) – Maximum number of fitting iterations
- **patience** (int) – Number of epochs to wait until termination when progress between epochs is below *stop_loss*
- **stop_loss** (float) – Threshold for difference in loss between epochs when an epoch is considered to make no more progress.
- **optimizer** (str) – Which optimizer to use. Default is Stochastic Gradient Descent. See PyTorch documentation for information.
- **callbacks** (list or None) – List of callback objects. Call signature is callback(epoch, model, optimizer)
- ****optimizer_kwargs** – Additional keyword arguments passed to the optimizer.

Returns result

Return type `TorchBatchFitResult`

`pyhdx.fitting.`**`fit_gibbs_global_batch_aligned`**(*hdx_set*, *initial_guess*, *r1=1*, *r2=1*, *epochs=200000*, *patience=50*, *stop_loss=5e-06*, *optimizer='SGD'*, *callbacks=None*, ***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** (*HDXMeasurementSet*) – Input HDX measurements
- **initial_guess** (Series or DataFrame or ndarray) – Gibbs free energy initial guesses (shape Ns x Nr or Nr, units J/mol)
- **r1** (float) – Regularizer value r1 (along residues)
- **r2** (float) – Regularizer value r2 (along protein states/samples)
- **epochs** (int) – Maximum number of fitting iterations
- **patience** (int) – Number of epochs to wait until termination when progress between epochs is below *stop_loss*
- **stop_loss** (float) – Threshold for difference in loss between epochs when an epoch is considered to make no more progress.
- **optimizer** (str) – Which optimizer to use. Default is Stochastic Gradient Descent. See PyTorch documentation for information.
- **callbacks** (list or None) – List of callback objects. Call signature is callback(epoch, model, optimizer)
- ****optimizer_kwargs** – Additional keyword arguments passed to the optimizer.

Returns result

Return type `TorchBatchFitResult`

`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** – Symfit fitresults object.

Return type FitResults

`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**

Return type *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** – dataclass with fit result

Return type dataclass

`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**

Return type *KineticsFitResult*

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

estimate default bound for rate fitting from a series of timepoints

Parameters **times** (*array_like*) –

Returns **bounds** – lower and upper bounds

Return type tuple

pyhdx.fitting.**run_optimizer**(*inputs*, *output_data*, *optimizer_klass*, *optimizer_kwargs*, *model*, *criterion*, *regularizer*, *epochs=200000*, *patience=50*, *stop_loss=5e-06*, *callbacks=None*, *tqdm=True*)

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
- **callable** (*regularizer*) – 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
- **callbacks** (list or *None*) – List of callback functions
- **tqdm** (bool) – Toggle tqdm progress bar

6.3 Fitting PyTorch

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

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.**TorchFitResult**(*hdxm_set*, *model*, *losses=None*, ***metadata*)

PyTorch Fit result object.

Parameters

- **hdxm_set** (*HDXMeasurementSet*) –
- **model** –
- ****metdata** –

property dG
: output dG as `Series` or as `DataFrame`

  index is residue numbers

eval(*timepoints*)
: evaluate the model at timepoints and return dataframe

static generate_output(*hdxm*, *dG*)

  Parameters

  - **hdxm** (*HDXMeasurement*) –
  - **dG** (`Series` with r_number as index) –

get_dcalc(*timepoints=None*)
: returns calculated d uptake for optional timepoints if no timepoints are given, a default set of logarithmically space timepoints is generated

get_peptide_mse()
: Get a dataframe with mean squared error per peptide (ie per peptide squared error averaged over time)

get_residue_mse()
: Get a dataframe with residue mean squared errors

  Errors are from peptide MSE which is subsequently reduced to residue level by weighted averaging

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

  Return type `ndarray`

property mse_loss
: Losses from mean squared error part of Lagrangian

  Type obj

  Type *float*

property reg_loss
: Losses from regularization part of Lagrangian

  Type `float`

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

  Type `float`

to_file(*file_path*, *include_version=True*, *include_metadata=True*, *fmt='csv'*, ***kwargs*)
: save only output to file

property total_loss
: Total loss value of the Lagrangian

  Type obj

  Type *float*

class pyhdx.fitting_torch.**TorchFitResultSet**(*results*)
: Set of multiple TorchFitResults

to_file(*file_path*, *include_version=True*, *include_metadata=True*, *fmt='csv'*, ***kwargs*)
save only output to file

pyhdx.fitting_torch.**estimate_errors**(*hdxm*, *dG*)
Calculate covariances and uncertainty (perr, experimental)

Parameters

- **hdxm** (*HDXMeasurement*) –
- **dG** (ndarray) – Array with dG values.

6.4 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

Return type pd.DataFrame

pyhdx.fileIO.**csv_to_hdxm**(*filepath_or_buffer*, *comment='#'*, ***kwargs*)
Reads a pyhdx .csv file or buffer into a pyhdx.models.HDXMeasurement or pyhdx.models.HDXMeasurementSet object.

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** – Resulting HDXMeasurement object with *r_number* as index

Return type *pyhdx.models.HDXMeasurement*

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** – Resulting Protein object with *r_number* as index

Return type *pyhdx.models.Protein*

`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** (`str` or *pathlib.Path*) – File path of the target file to write.
- **df** (*pd.DataFrame*) – The pandas dataframe to write to the file.
- **fmt** (`str`) – Specify the formatting of the output. Options are '.csv' (machine readable) or 'pprint' (human readable)
- **include_metadata** (`bool` or `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** – Resulting io.StringIO object.

Return type io.StringIO

`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** (`str`) – Specify the formatting of the output. Options are 'csv' (machine readable) or 'pprint' (human readable)
- **include_metadata** (`bool` or `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** – Resulting io.StringIO object.

Return type io.StringIO

`pyhdx.fileIO.load_fitresult`(*fit_dir*)

Load a fitresult into a fitting_torch.TorchSingleFitResult or `TorchBatchFitResult` object

The fit result must be in the format as generated by saving a fit result with *save_fitresult*.

:param fir_dir `str` or `Path`: Fit result directory.

`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) – Time unit of the field 'exposure'. Options are 'h', 'min' or 's'

Returns **full_df** – Peptides as a pandas DataFrame

Return type DataFrame

`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: dG.csv/.txt: Fit output result (dG, 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 str) – Output directory to save fitresult to
- **fit_result** (*pydhx.fittin_torch.TorchFitResult*) – fit result object to save
- **log_lines** (list) – Optional additional lines to write to log file.

6.5 Output

`class pyhdx.output.FitReport(fit_result, title=None, doc=None, add_date=True, temp_dir=None, **kwargs)`

Create .pdf output of a fit result

`class pyhdx.output.LocalThreadExecutor`

`shutdown(wait=True)`

Clean-up the resources associated with the Executor.

It is safe to call this method several times. Otherwise, no other methods can be called after this one.

Parameters **wait** – If True then shutdown will not return until all running futures have finished executing and the resources used by the executor have been reclaimed.

`submit(f, *args, **kwargs)`

Submits a callable to be executed with the given arguments.

Schedules the callable to be executed as fn(*args, **kwargs) and returns a Future instance representing the execution of the callable.

Returns A Future representing the given call.

6.6 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 with thresholds

Return type tuple

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*)

Reduce overlapping intervals to its non-overlapping intveral parts

Author: Brent Pedersen Source: https://github.com/brentp/interlap/blob/3c4a5923c97a5d9a11571e0c9ea5bb7ea4e784ee/interlap.py#L224

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.

```
>>> 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

Return type 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

SEVEN

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.

7.1 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 mode (*Selector*, default='Manual', options=['Manual', 'Batch'])

Input files label (*String*, default='Input files:')

Input files (*List*, bounds=(0, None), default=[])
HDX input files. Currently only support DynamX format

Batch file label (*String*, default='Batch file (yaml)')

Batch file (*Parameter*)
Batch file input:

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

Back exchange percentage (*Number*, bounds=(0, 100), default=28.0)
Global percentage of back-exchange

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

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

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

Deuterium percentage (*Number*, bounds=(0, 100), default=90.0)
Percentage of deuterium in the labelling 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 measurement(s) (*Action*)
Parse selected peptides for further analysis and apply back-exchange correction

HDX Measurements (*ListSelector*, options=[])
Lists added HDX-MS measurements

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

`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 ()', options=['Half-life ()', 'Association'])
Choose method for determining initial guesses.

Dataset (for bounds) (*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

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

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

G Fit

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.

Guess mode (*Selector*, default='One-to-one', options=['One-to-one', 'One-to-many'])
Use initial guesses for each protein state (one-to-one) or use one initialguess for all protein states (one-to-many)

Guess state (*Selector*, options=[])
Which protein state to use for initial guess when using one-to-many guesses

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

Stop loss (*Number*, bounds=(0, None), default=5e-06)
Threshold loss difference below which to stop fitting.

Stop patience (*Integer*, bounds=(1, None), default=50)
Number of epochs where stop loss should be satisfied before stopping.

Learning rate (*Number*, bounds=(0, None), default=10000.0)
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=200000)
Maximum number of epochs (iterations.

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

Regularizer 2 (sample axis) (*Number*, bounds=(0, None), default=1)
Value of the regularizer along sample axis.

R2 reference (*Selector*, options=[])
Select reference state to use in batch fitting

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

Do Fitting (*Action*)
Start global fitting

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

`class pyhdx.web.controllers.DifferentialControl`(*parent*, ***params*)

Differential HDX

Reference state (*Selector*, options=[])
Which of the states to use as reference

Comparison name (*String*, default='comparison_1')
Name for the comparison table

Add comparison (*Action*)

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

`class pyhdx.web.controllers.ColorTransformControl`(*parent*, ***param*)

Color Transform

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 Quantity (*Selector*, options=[])

Current color transform (*String*, default=”)

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

Number of colours (*Integer*, bounds=(1, 10), default=2)
Number of classification colors.

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

Colormap (*Selector*, options=[])

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

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

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

Color transform name (*String*, default=”)
Name for the color transform to add

Update color transform (*Action*)

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

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

Protein Control

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

File binary (*Parameter*)
Corresponds to file upload value

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

Load structure (*Action*)

Highlight mode (*Selector*, default='Single', options=['Single', 'Range'])

Highlight range (*Range*, default=(1, 2), length=2, step=1)

Highlight value (*Integer*, default=1)

Highlight (*Action*)

Clear highlight (*Action*)

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

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

Graph Control

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

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

File Export

<outdated docstring> 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

Additional GUI elements on:

pyhdx.web.base.ControlPanel: **parent**, **_excluded**

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

Figure Export

Figure (*Selector*, default='scatter', options=['scatter', 'linear_bars', 'rainbowclouds'])

Table (*Selector*, options=[])
which table data to use for figure

Selection (*Selector*, options=[])
for scatter / rainbowclouds, which fit to use

Groupby (*Selector*, options=[])
for linear bars, how to group the bars

Reference (*Selector*, options=[])

Figure format (*Selector*, default='png', options=['png', 'pdf', 'svg', 'eps'])

Number of columns (*Integer*, bounds=(1, 4), default=2)
Number of columns in subfigure

Aspect ratio (*Number*, default=3.0)
Subfigure aspect ratio

Figure width (mm) (*Number*, bounds=(50, None), default=160.0)
Width of the output figure

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

`class pyhdx.web.controllers.SessionManagerControl`(*parent*, ***params*)

Session Manager

Session file (*Parameter*)

Load session (*Action*)

Reset session (*Action*)

Additional GUI elements on:

`pyhdx.web.base.ControlPanel`: **parent**, **_excluded**

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pyhdx.fileIO, 44
pyhdx.fitting, 38
pyhdx.fitting_torch, 42
pyhdx.models, 29
pyhdx.output, 46
pyhdx.support, 47

INDEX

A

add_alignment() (*pyhdx.models.HDXMeasurementSet method*), 33
apply_interval() (*pyhdx.models.Coverage method*), 30
apply_interval() (*pyhdx.models.CoverageSet method*), 30
array_intersection() (*in module pyhdx.models*), 37
autowrap() (*in module pyhdx.support*), 47

B

block_length (*pyhdx.models.Coverage property*), 30

C

calc_rfu() (*pyhdx.models.HDXTimepoint method*), 34
check_bounds() (*in module pyhdx.fitting*), 39
chi_squared (*pyhdx.fitting.EmptyResult attribute*), 38
colors_to_pymol() (*in module pyhdx.support*), 47
ColorTransformControl (*class in pyhdx.web.controllers*), 53
contiguous_regions() (*in module pyhdx.models*), 37
Coverage (*class in pyhdx.models*), 29
coverage (*pyhdx.models.HDXMeasurement attribute*), 31
CoverageSet (*class in pyhdx.models*), 30
csv_to_dataframe() (*in module pyhdx.fileIO*), 44
csv_to_hdxm() (*in module pyhdx.fileIO*), 44
csv_to_protein() (*in module pyhdx.fileIO*), 44

D

d_exp (*pyhdx.models.HDXMeasurement property*), 31
d_exp (*pyhdx.models.HDXMeasurementSet attribute*), 33
d_exp (*pyhdx.models.HDXTimepoint property*), 34
data (*pyhdx.models.HDXMeasurement attribute*), 31
dataframe_to_file() (*in module pyhdx.fileIO*), 45
dataframe_to_stringio() (*in module pyhdx.fileIO*), 45
DeltaGFit (*class in pyhdx.fitting_torch*), 42
dG (*pyhdx.fitting_torch.TorchFitResult property*), 42
DifferentialControl (*class in pyhdx.web.controllers*), 53

E

EmptyResult (*class in pyhdx.fitting*), 38
estimate_errors() (*in module pyhdx.fitting_torch*), 44
eval() (*pyhdx.fitting_torch.TorchFitResult method*), 43
exchanges (*pyhdx.models.HDXMeasurementSet property*), 33
exposure (*pyhdx.models.HDXTimepoint attribute*), 34
exposures (*pyhdx.models.PeptideMasterTable property*), 36

F

FigureExportControl (*class in pyhdx.web.controllers*), 56
FileExportControl (*class in pyhdx.web.controllers*), 56
fit_gibbs_global() (*in module pyhdx.fitting*), 39
fit_gibbs_global_batch() (*in module pyhdx.fitting*), 39
fit_gibbs_global_batch_aligned() (*in module pyhdx.fitting*), 40
fit_kinetics() (*in module pyhdx.fitting*), 40
fit_rates() (*in module pyhdx.fitting*), 41
fit_rates_half_time_interpolate() (*in module pyhdx.fitting*), 41
fit_rates_weighted_average() (*in module pyhdx.fitting*), 41
FitControl (*class in pyhdx.web.controllers*), 52
FitReport (*class in pyhdx.output*), 46
forward() (*pyhdx.fitting_torch.DeltaGFit method*), 42

G

gen_subclasses() (*in module pyhdx.support*), 47
generate_output() (*pyhdx.fitting_torch.TorchFitResult static method*), 43
GenericFitResult (*class in pyhdx.fitting*), 38
get() (*pyhdx.models.HDXMeasurementSet method*), 33
get_bounds() (*in module pyhdx.fitting*), 41
get_d() (*pyhdx.fitting.KineticsFitResult method*), 38
get_data() (*pyhdx.models.PeptideMasterTable method*), 36

get_dcalc() (pyhdx.fitting_torch.TorchFitResult method), 43
get_k_int() (pyhdx.models.Protein method), 37
get_masks() (pyhdx.models.CoverageSet method), 30
get_p() (pyhdx.fitting.KineticsFitResult method), 38
get_param() (pyhdx.fitting.KineticsFitResult method), 38
get_peptide_mse() (pyhdx.fitting_torch.TorchFitResult method), 43
get_residue_mse() (pyhdx.fitting_torch.TorchFitResult method), 43
get_sections() (pyhdx.models.Coverage method), 30
get_squared_errors() (pyhdx.fitting_torch.TorchFitResult method), 43
get_state() (pyhdx.models.PeptideMasterTable method), 36
get_tensors() (pyhdx.models.HDXMeasurement method), 31
get_tensors() (pyhdx.models.HDXMeasurementSet method), 33
GraphControl (class in pyhdx.web.controllers), 55
grouper() (in module pyhdx.support), 47
guess_deltaG() (pyhdx.models.HDXMeasurement method), 32
guess_deltaG() (pyhdx.models.HDXMeasurementSet method), 33

H

hdx_intersection() (in module pyhdx.models), 37
HDXMeasurement (class in pyhdx.models), 31
HDXMeasurementSet (class in pyhdx.models), 33
HDXTimepoint (class in pyhdx.models), 34
hex_to_rgb() (in module pyhdx.support), 47

I

index (pyhdx.models.Coverage property), 30
index (pyhdx.models.CoverageSet property), 31
InitialGuessControl (class in pyhdx.web.controllers), 51

K

KineticsFitResult (class in pyhdx.fitting), 38

L

load_fitresult() (in module pyhdx.fileIO), 45
LocalThreadExecutor (class in pyhdx.output), 46

M

make_color_array() (in module pyhdx.support), 47
make_monomer() (in module pyhdx.support), 47
module
    pyhdx.fileIO, 44
    pyhdx.fitting, 38
    pyhdx.fitting_torch, 42
    pyhdx.models, 29
    pyhdx.output, 46
    pyhdx.support, 47
mse_loss (pyhdx.fitting_torch.TorchFitResult property), 43
multi_otsu() (in module pyhdx.support), 47

N

name (pyhdx.models.HDXMeasurement property), 32
name (pyhdx.models.HDXTimepoint property), 35
Np (pyhdx.models.Coverage property), 29
Np (pyhdx.models.HDXMeasurement property), 31
Nr (pyhdx.models.Coverage property), 29
Nr (pyhdx.models.HDXMeasurement property), 31
Nt (pyhdx.models.HDXMeasurement property), 31

O

output (pyhdx.fitting.KineticsFitResult property), 38

P

params (pyhdx.fitting.EmptyResult attribute), 38
PeptideFileInputControl (class in pyhdx.web.controllers), 49
PeptideMasterTable (class in pyhdx.models), 35
peptides (pyhdx.models.HDXMeasurement attribute), 32
percent_coverage (pyhdx.models.Coverage property), 30
pH (pyhdx.models.HDXMeasurement property), 32
pprint_df_to_file() (in module pyhdx.support), 47
Protein (class in pyhdx.models), 37
ProteinControl (class in pyhdx.web.controllers), 54
pyhdx.fileIO
    module, 44
pyhdx.fitting
    module, 38
pyhdx.fitting_torch
    module, 42
pyhdx.models
    module, 29
pyhdx.output
    module, 46
pyhdx.support
    module, 47

R

r_number (pyhdx.models.Coverage property), 30
rate (pyhdx.fitting.KineticsFitResult property), 39
RatesFitResult (class in pyhdx.fitting), 39

read_dynamx() (*in module pyhdx.fileIO*), 45
reduce_inter() (*in module pyhdx.support*), 47
redundancy (*pyhdx.models.Coverage property*), 30
reg_loss (*pyhdx.fitting_torch.TorchFitResult property*), 43
regularization_percentage (*pyhdx.fitting_torch.TorchFitResult property*), 43
rfu_peptides (*pyhdx.models.HDXMeasurement property*), 32
rfu_peptides (*pyhdx.models.HDXTimepoint property*), 35
rfu_residues (*pyhdx.models.HDXMeasurement property*), 32
rfu_residues (*pyhdx.models.HDXMeasurementSet property*), 34
rfu_residues (*pyhdx.models.HDXTimepoint property*), 35
rgb_to_hex() (*in module pyhdx.support*), 48
run_optimizer() (*in module pyhdx.fitting*), 42

S

s_r_mask (*pyhdx.models.CoverageSet property*), 31
save_fitresult() (*in module pyhdx.fileIO*), 46
scale() (*in module pyhdx.support*), 48
select() (*pyhdx.models.PeptideMasterTable method*), 36
series_to_pymol() (*in module pyhdx.support*), 48
SessionManagerControl (*class in pyhdx.web.controllers*), 57
set_backexchange() (*pyhdx.models.PeptideMasterTable method*), 36
set_control() (*pyhdx.models.PeptideMasterTable method*), 36
shutdown() (*pyhdx.output.LocalThreadExecutor method*), 46
state (*pyhdx.models.HDXMeasurement attribute*), 32
state (*pyhdx.models.HDXTimepoint attribute*), 35
states (*pyhdx.models.PeptideMasterTable property*), 37
submit() (*pyhdx.output.LocalThreadExecutor method*), 46

T

tau (*pyhdx.fitting.KineticsFitResult property*), 39
temperature (*pyhdx.models.HDXMeasurement property*), 32
timepoints (*pyhdx.models.HDXMeasurement attribute*), 32
timepoints (*pyhdx.models.HDXMeasurementSet attribute*), 34
to_file() (*pyhdx.fitting_torch.TorchFitResult method*), 43
to_file() (*pyhdx.fitting_torch.TorchFitResultSet method*), 43
to_file() (*pyhdx.models.HDXMeasurement method*), 33
to_file() (*pyhdx.models.HDXMeasurementSet method*), 34
to_file() (*pyhdx.models.Protein method*), 37
TorchFitResult (*class in pyhdx.fitting_torch*), 42
TorchFitResultSet (*class in pyhdx.fitting_torch*), 43
total_loss (*pyhdx.fitting_torch.TorchFitResult property*), 43
try_wrap() (*in module pyhdx.support*), 48

W

weighted_average() (*pyhdx.models.HDXTimepoint method*), 35

X

X (*pyhdx.models.Coverage attribute*), 29
X_norm (*pyhdx.models.Coverage property*), 29

Z

Z (*pyhdx.models.Coverage attribute*), 29
Z_norm (*pyhdx.models.Coverage property*), 29