

Supporting Material Jupyter Notebooks for:

PyFolding: An open-source software package for graphing, simulation and analysis of the biophysical properties of proteins

Alan R. Lowe^{1,2,3*}, Albert Perez-Riba⁴, Laura S. Itzhaki⁴ and Ewan R.G. Main^{5*}

November 29, 2017

¹London Centre for Nanotechnology
17-19 Gordon Street, London
WC1H 0AH, UK

²Structural and Molecular Biology, University College London,
Gower Street, London,
WC1E 6BT, UK

³Department of Biological Sciences, Birkbeck College, University of London
Malet Street, London
WC1E 7HX, UK

⁴Department of Pharmacology, University of Cambridge
Tennis Court Road, Cambridge
CB2 1PD, UK

⁵School of Biological and Chemical Sciences, Queen Mary University of London
Mile End Road, London
E1 4NS, UK

*corresponding authors (ARL: a.lowe@ucl.ac.uk, ERGM: e.main@qmul.ac.uk)

Contents

1	Generic Data Importer Function Notebook	3
1.1	Loading any data set	4
1.2	Plotting data and fitting to a new model	6
1.2.1	Sinusoidal model with phase and amplitude	7
2	Generating New Models/ Equations Notebook	9
2.1	Loading the Generic Data into PyFolding	10
2.2	Format for model definition	11
2.3	New Model Formatting	12
3	SI Notebook 1 - Fitting equilibrium and kinetic folding data to simple models	15
3.1	Data Format	15
3.1.1	Considerations	16
3.2	Plotting Data	16
3.3	Fitting Data	18
3.3.1	List the models in pyfolding:	18
3.3.2	Print a model and its parameters, etc:	18
3.3.3	Fitting the Data.	19
3.4	Plotting Data to a Fancier Graph	23
3.5	Fit to multiple models	24
4	SI Notebook 2 - Fitting more complicated models to some example data	29
4.1	Data Format	29
4.1.1	Special Considerations	29
4.2	Plotting Chevron phases	30
4.3	Fitting Data To Multiple Models	31
4.3.1	List Models	31
4.3.2	Fit to multiple models	32
4.3.3	Fit to multiple models and output with prettier graphics	38
5	SI Notebook 3 - Performing and automating higher-level calculations such as ΔGs & Phi-values	47
5.1	Data Format	47
5.2	Assign Data to a specific protein object	48
5.3	Fit the Data	49
5.4	Calculating Free Energy of unfolding	55
5.5	Calculating Phi Values	56
5.5.1	As you can see this works. YAY!	56
6	SI Notebook 4 - Fitting multiple datasets with linked equations (share certain parameters between the models)	57
6.1	Data Format	57
6.2	Load and Plot the Data	58
6.3	Custom Plotting	58
6.4	Set temperature	59
6.5	List the Models used	60
6.6	Create a New model - called FastPhase	61

6.7	Setting up a custom Global fit	62
6.8	Display Results from custom Global fit	63
6.9	Plot the Figure of the custom Global fit	64
6.10	Plot custom Global fit with "prettier" graphics	65
6.10.1	1st - we need to add the kinetic results to the protein object	65
6.10.2	2nd - lets get the equilibrium data & fit to a three state model	65
6.10.3	3rd - lets plot both together & print out to pdf	66
7	SI Notebook 5 - Fitting to a HomoZipper/HomoPolymer Ising Model	68
7.1	Data Format	68
7.2	Import Data, assign names and put into a list	69
7.3	Set the Temperature	69
7.4	Plot the data	70
7.5	Print information on the HomoZipper Ising Model	70
7.6	Fitting to a Homopolymer Ising model	71
7.7	Set up Our Topology	72
7.8	Automatic global fitting to the homozipper model	75
8	SI Notebook 6 - Fitting to a HeteroPolymer Ising Model	79
8.1	Data Format	79
8.2	Import Data, assign names and put into a list	80
8.3	Set Temperature	80
8.4	Plot the data	81
8.5	Fitting to a Heteropolymer Ising model	82
8.6	Assign Topology	82
8.7	Plot our Chosen Topology	83
8.8	Fit Data with our chosen Topology	86
9	SI Notebook 7 - Simulating Ising Models	111
9.1	Initialise domains and give them their appropriate values	112
9.2	Build a topology	112
9.3	Generate partition function	113
9.4	Simulate	113
10	SI Notebook 8 - Global fit Equilibrium Curves from a dimeric protein to dimeric equilibrium models	116
10.1	Data Format	116
10.2	Import Data, assign names and put into a list	117
10.3	Set Temperature	117
10.4	Plot Data	117
10.5	Fit Data to 3 state denaturation that unfolds via dimeric I	119
10.6	Automatic global fitting to the Three State Dimeric Intermediate Equilibrium model	120
10.7	Plot the fitted data	123
10.8	Simulate Data	124

1 Generic Data Importer Function Notebook

[Authors] ARL & ERGM

This Notebook aims to show an example of importing generic data into PyFolding and fitting it with a user defined model. Once you have tested it here you can add it to your models.py from the PyFolding installation and setup PyFolding again to incorporate it into your local copy or simply copy it into whichever notebook you are using.

So as always lets load up PyFolding:

```
In [1]: #First off let's load pyfolding & pyplot into this ipython notebook  
        # (pyplot allows us to plot more complex figures of our results):
```

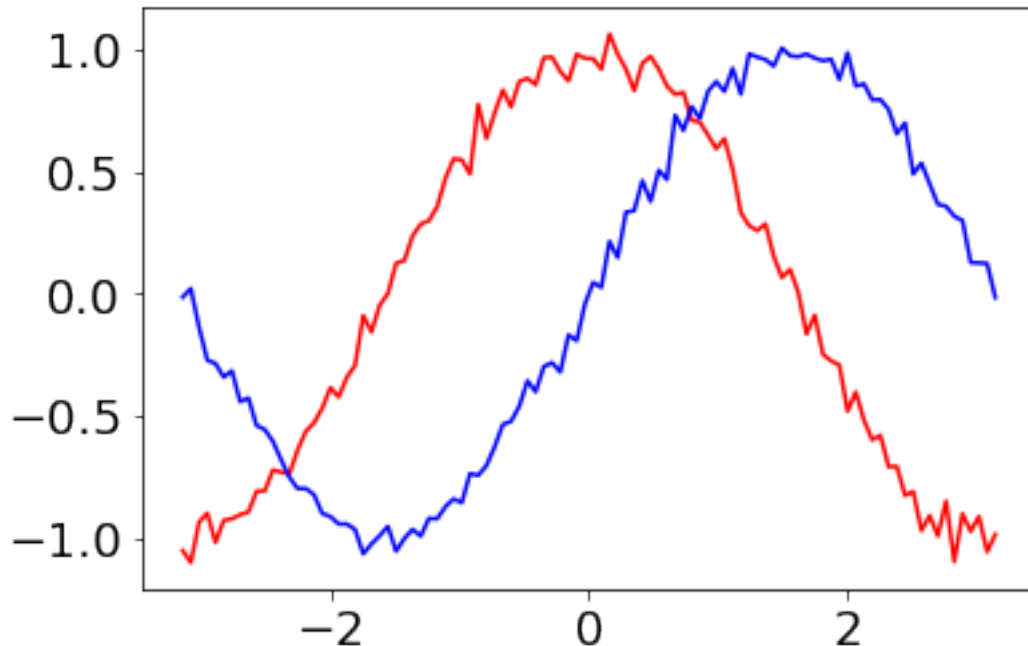
```
%matplotlib inline  
import pyfolding  
from pyfolding import models  
  
# let's use some other libraries also  
import matplotlib.pyplot as plt  
import numpy as np
```

<IPython.core.display.Javascript object>

PyFolding: Jupyter autoscrolling has been disabled

NOTE: This section just autogenerates a CSV file of data to be imported using the generic importer

```
In [2]: from pyfolding import utils  
        from collections import OrderedDict  
        x = np.linspace(-np.pi, np.pi, 100)  
        y_cos = np.cos(x) + np.random.randn(len(x))*0.05  
        y_sin = np.sin(x) + np.random.randn(len(x))*0.05  
  
        plt.figure()  
        plt.plot(x, y_cos, 'r-', x, y_sin, 'b-')  
        plt.show()  
  
        data = OrderedDict([('x',x), ('cos',y_cos), ('sin',y_sin)])  
  
        utils.write_CSV('../examples/waves.csv', data)
```



Writing .csv file (../examples/waves.csv)...

1.1 Loading any data set

Here we will load a data set which contains sine and cosine functions over the range $(-\pi, \pi]$ to demonstrate how PyFolding can deal with arbitrary datasets. We will then plot and fit a simple mode to recover the amplitude and phase of the waves.

```
In [3]: # loading the data - The kinetics of each protein is in one .csv
        # as per PyFolding SI Notebooks 1 and 2
```

```
pth = "../examples/"
wave_data = pyfolding.read_generic_data(pth, "waves.csv")
```

```
In [4]: print wave_data
```

```
<pyfolding.core.GenericData object at 0x113fc3150>
```

We can use this generic object as we would any other

For example, let's extract the datasets found within, and plot x versus y (y defaults to the first label, e.g. 'cos' in this case)

```
In [5]: print wave_data.labels
```

```
['x', 'cos', 'sin']
```

```
In [6]: print wave_data.x
```

```
[-3.14159265 -3.07812614 -3.01465962 -2.9511931  -2.88772658 -2.82426006
 -2.76079354 -2.69732703 -2.63386051 -2.57039399 -2.50692747 -2.44346095
 -2.37999443 -2.31652792 -2.2530614  -2.18959488 -2.12612836 -2.06266184
 -1.99919533 -1.93572881 -1.87226229 -1.80879577 -1.74532925 -1.68186273
 -1.61839622 -1.5549297  -1.49146318 -1.42799666 -1.36453014 -1.30106362
 -1.23759711 -1.17413059 -1.11066407 -1.04719755 -0.98373103 -0.92026451
 -0.856798   -0.79333148 -0.72986496 -0.66639844 -0.60293192 -0.53946541
 -0.47599889 -0.41253237 -0.34906585 -0.28559933 -0.22213281 -0.1586663
 -0.09519978 -0.03173326  0.03173326  0.09519978  0.1586663  0.22213281
  0.28559933  0.34906585  0.41253237  0.47599889  0.53946541  0.60293192
  0.66639844  0.72986496  0.79333148  0.856798   0.92026451  0.98373103
  1.04719755  1.11066407  1.17413059  1.23759711  1.30106362  1.36453014
  1.42799666  1.49146318  1.5549297  1.61839622  1.68186273  1.74532925
  1.80879577  1.87226229  1.93572881  1.99919533  2.06266184  2.12612836
  2.18959488  2.2530614  2.31652792  2.37999443  2.44346095  2.50692747
  2.57039399  2.63386051  2.69732703  2.76079354  2.82426006  2.88772658
  2.9511931   3.01465962  3.07812614  3.14159265]
```

```
In [7]: print wave_data.y
```

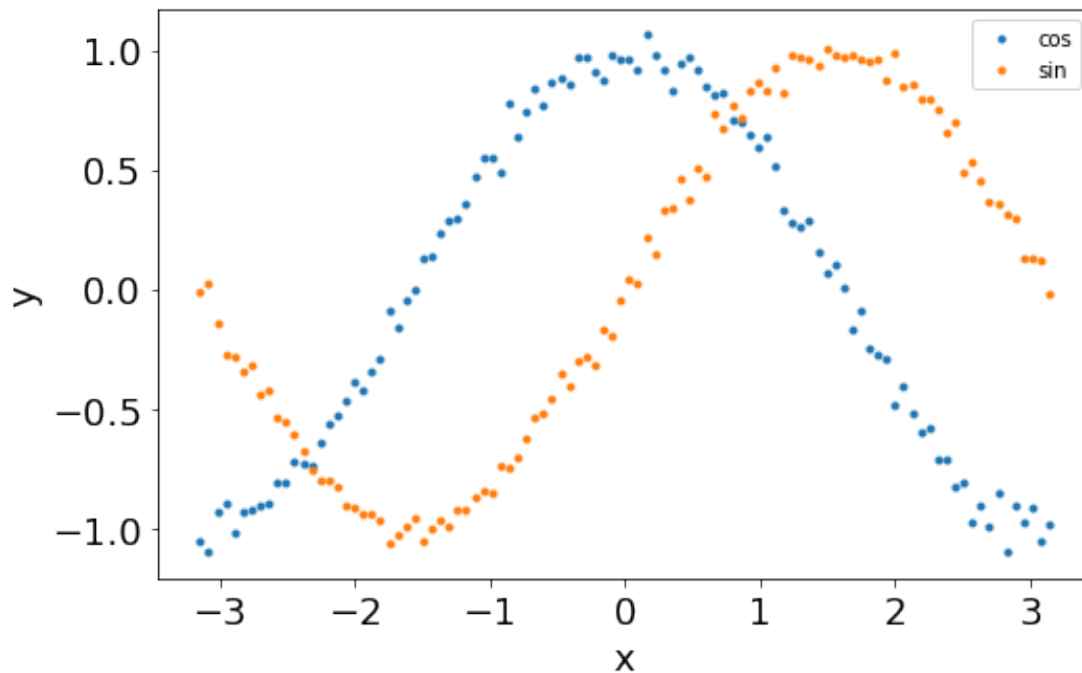
```
[ -1.05184198e+00  -1.10086036e+00  -9.34577866e-01  -8.98684831e-01
 -1.01876844e+00  -9.28422902e-01  -9.21834189e-01  -9.05072298e-01
 -8.94156059e-01  -8.09748407e-01  -8.07323495e-01  -7.21774418e-01
 -7.31338883e-01  -7.37619719e-01  -6.40085902e-01  -5.63761858e-01
 -5.28692200e-01  -4.69408225e-01  -3.84358928e-01  -4.23246500e-01
 -3.42038709e-01  -2.93540838e-01  -8.99022064e-02  -1.56712701e-01
 -4.70741508e-02  2.21520170e-04  1.24822637e-01  1.35374655e-01
 2.36447385e-01  2.84709078e-01  2.98650228e-01  3.56154884e-01
 4.71149537e-01  5.51995373e-01  5.48471932e-01  4.89630950e-01
 7.75252581e-01  6.35591971e-01  7.39330183e-01  8.34019948e-01
 7.64425698e-01  8.68623007e-01  8.81952824e-01  8.55746330e-01
 9.66528664e-01  9.68849809e-01  9.09940500e-01  8.71076422e-01
 9.80475110e-01  9.64203100e-01  9.62147890e-01  9.19320898e-01
 1.06287708e+00  9.79996528e-01  9.19656691e-01  8.31644631e-01
 9.43496470e-01  9.71987785e-01  9.19820064e-01  8.50062902e-01]
```

8.16184298e-01	8.24049638e-01	7.10046130e-01	7.02083818e-01
6.49232729e-01	5.93266790e-01	6.34566975e-01	5.10333001e-01
3.31677692e-01	2.78595731e-01	2.58089568e-01	2.85880648e-01
1.53620903e-01	6.60839402e-02	9.97124703e-02	4.06463880e-03
-1.65095090e-01	-8.79775962e-02	-2.48359633e-01	-2.75328948e-01
-2.92200664e-01	-4.81186925e-01	-4.02666378e-01	-5.18212035e-01
-5.98537310e-01	-5.79979894e-01	-7.08734247e-01	-7.07753630e-01
-8.26412974e-01	-8.10114098e-01	-9.70090371e-01	-9.08403141e-01
-9.92313207e-01	-8.49087297e-01	-1.09725395e+00	-9.01188317e-01
-9.73493283e-01	-9.12334689e-01	-1.05720339e+00	-9.85473702e-01]

1.2 Plotting data and fitting to a new model

Note that we can call plot and fit this to an arbitrary function (i.e. it doesn't have to be anything to do with protein folding) using all of the same tools

In [8]: `wave_data.plot()`



1.2.1 Sinusoidal model with phase and amplitude

Lets define a sinusoidal model with phase and amplitude. We can use this to extract the amplitude and phase of the waves...

```
In [9]: class Wave(pyfolding.FitModel):
        """ Sine wave with amplitude and phase
        """
        def __init__(self):
            pyfolding.FitModel.__init__(self)
            fit_args = self.fit_func_args
            self.params = tuple( [(fit_args[i],i) for i in xrange(len(fit_args))] )
            self.default_params = np.array([1., 0.])

        def fit_func(self, x, amplitude, phase):
            return amplitude*np.sin(x+phase)

In [10]: wave_data.fit_func = Wave
         wave_data.fit()
```

```
=====
Fitting results
=====
```

ID: waves

Model: Wave

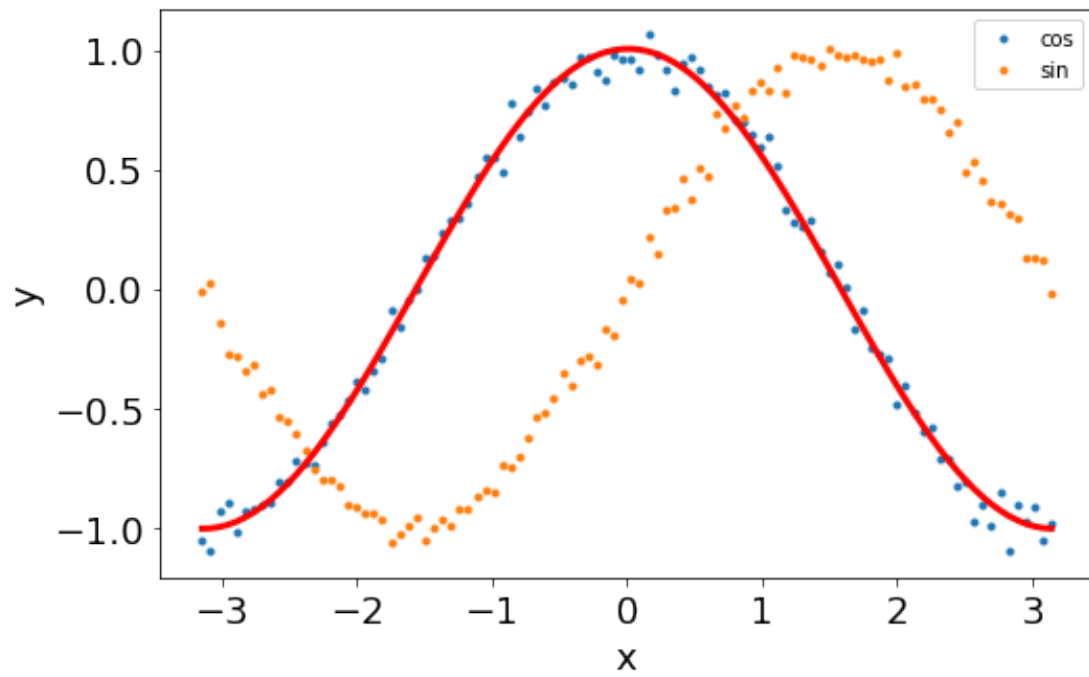
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) amplitude	1.00419 ± 0.00045	95% CI[1.00408,	1.00431]
(f) phase	1.55891 ± 0.00045	95% CI[1.55879,	1.55902]

```
-----
R^2: 0.99389
=====
```

```
In [11]: wave_data.plot()
```

```
In [12]: print np.pi/2
```

```
1.57079632679
```

End of Notebook

2 Generating New Models/ Equations Notebook

[Author] ERGM & ARL

This Notebook aims to show an example of importing generic data into PyFolding and fitting it with a user defined model. Once you have tested it here you can add it to your models.py from the PyFolding installation and setup PyFolding again to incorporate it into your local copy or simply copy it into whichever notebook you are using.

So as always lets load up PyFolding:

```
In [1]: # Load up pyfolding, etc.
        %matplotlib inline
        import pyfolding
        from pyfolding import models

        # let's use some other libraries also
        import matplotlib.pyplot as plt
        import numpy as np

        # Command imports pyfolding models
        from pyfolding.models import *
```

<IPython.core.display.Javascript object>

PyFolding: Jupyter autoscrolling has been disabled

NOTE: The cell below just autogenerates a CSV file of data to be imported using the generic importer

```
In [2]: # Lets create some data

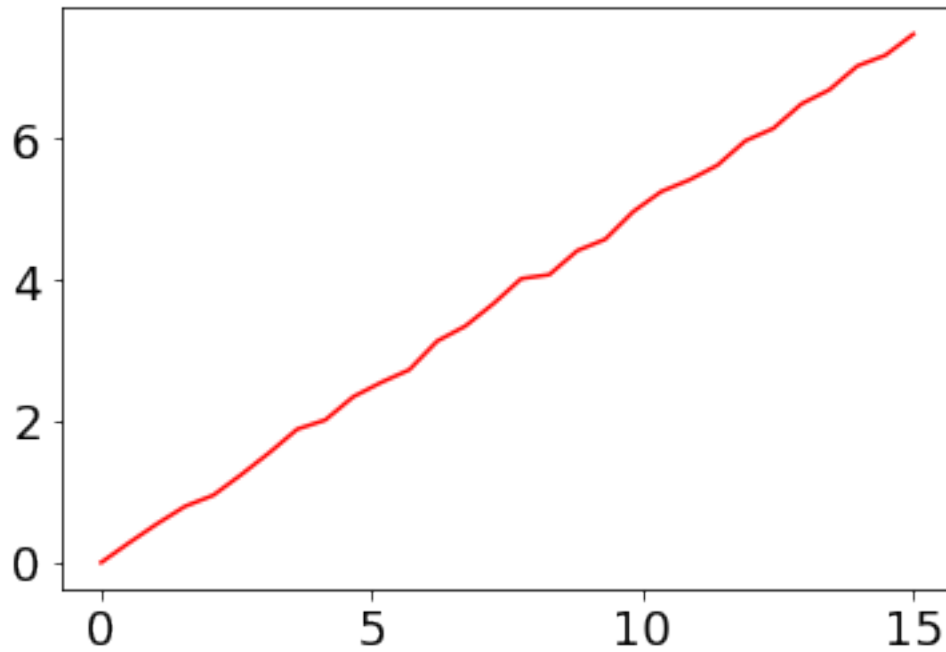
        from pyfolding import utils
        from collections import OrderedDict
        x = np.linspace(0, 15, num=30)
        y_straight_line = 0.5*(x) + np.random.randn(len(x))*0.05

        plt.figure()
        plt.plot(x, y_straight_line, 'r-')
```

```
plt.show()

data = OrderedDict([('x',x),('Straight_line',y_straight_line)])

utils.write_CSV('../examples/test_data.csv', data)
```



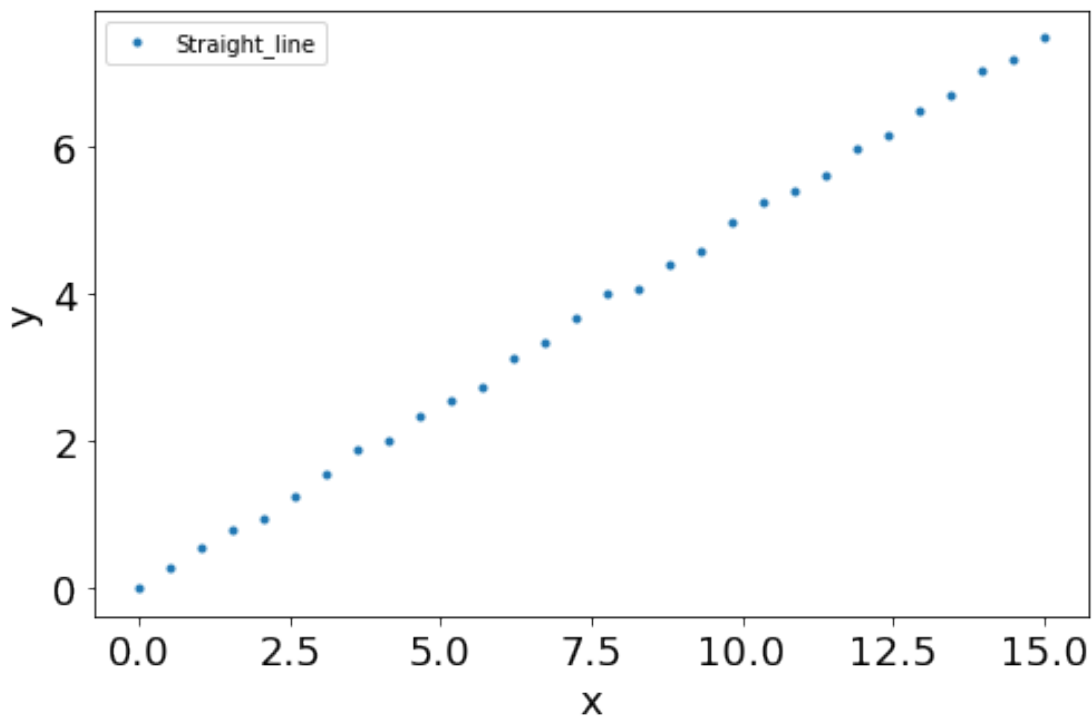
Writing .csv file (../examples/test_data.csv)...

2.1 Loading the Generic Data into PyFolding

In [3]: *# Lets load this generic data*

```
pth = "../examples/"
test_data = pyfolding.read_generic_data(pth,"test_data.csv")

# Lets plot it to confirm it is functioning correctly in pyFolding
test_data.plot()
```



```
In [4]: #If we want to set our temperature
```

```
pyfolding.set_temperature(25.0)
```

Set temperature to 25.00°C

(NOTE: Careful, this sets the temperature for all subsequent calculations)

2.2 Format for model definition

In the PyFolding there is a python file that defines all the models that we have inputted into PyFolding called models.py. These have the following format (I have used TwoStateEquilibrium as an example)

```
class TwoStateEquilibrium(core.FitModel):

    """Two state equilibrium denaturation curve - No sloping baseline.

    Folding Scheme:
    N <-> D
```

Params:

```
F = Fraction unfolded
m = m-value
x = denaturant concentration (M)
d50 = denaturant midpoint (M)
R = Universal Gas Constant (kcal.mol-1.K-1)
T = Temperature (Kelvin)
```

Reference:

```
Clarke and Fersht. Engineered disulfide bonds as probes of
the folding pathway of barnase: Increasing the stability
of proteins against the rate of denaturation.
Biochemistry (1993) vol. 32 (16) pp. 4322-4329"""
```

```
def __init__(self):
    core.FitModel.__init__(self)
    fit_args = self.fit_func_args
    self.params = tuple( [(fit_args[i],i) for i in xrange(len(fit_args))] )
    self.default_params = np.array([1.5, 5.])
    self.verified = True

def fit_func(self, x, m, d50):
    F = ( np.exp((m*(x-d50))/core.temperature.RT)) / (1.+np.exp((m*(x-d50))/core.temperature.RT))
    return F

@property
def equation(self):
    return r'F = \frac{\exp( m(x-d_{50})) / RT}{ 1+\exp(m(x-d_{50}))/RT}'
```

2.3 New Model Formatting

The parts that you need to use the make a new equation/model are:

```
class TwoStateEquilibrium(core.FitModel):

def __init__(self):
    core.FitModel.__init__(self)
    fit_args = self.fit_func_args
    self.params = tuple( [(fit_args[i],i) for i in xrange(len(fit_args))] )
    self.default_params = np.array([1.5, 5.])
    self.verified = True
```

```
def fit_func(self, x, m, d50):
    F = ( np.exp((m*(x-d50))/core.temperature.RT)) / (1.+np.exp((m*(x-d50))/core.temperature.RT))
    return F
```

So lets use these and create our own equation and run it in this notebook (no need to modify the folder models.py) to fit the straight line data we generated above:

```
In [5]: # lets give our equation a name
class straightline(core.FitModel):

    # We don't need to touch any of this next part except "self.default_params"
    # "self.default_params" are the initial parameters the model will use so lets change
    # the values in the brackets:
    # "np.array([1., 1.])".

    def __init__(self):
        core.FitModel.__init__(self)
        fit_args = self.fit_func_args
        self.params = tuple( [(fit_args[i],i) for i in xrange(len(fit_args))] )
        self.default_params = np.array([1., 1.])

    # This is our actual equation we will be using,
    # so lets input a straight line i.e.  $y = mx + c$ 
    def fit_func(self, x, gradient, intercept):
        y = (gradient*x)+intercept
        return y
```

```
In [6]: # Lets fit our earlier generated data and see if the model works!
```

```
test_data.fit_func = straightline
test_data.fit()
```

```
=====
Fitting results
=====
```

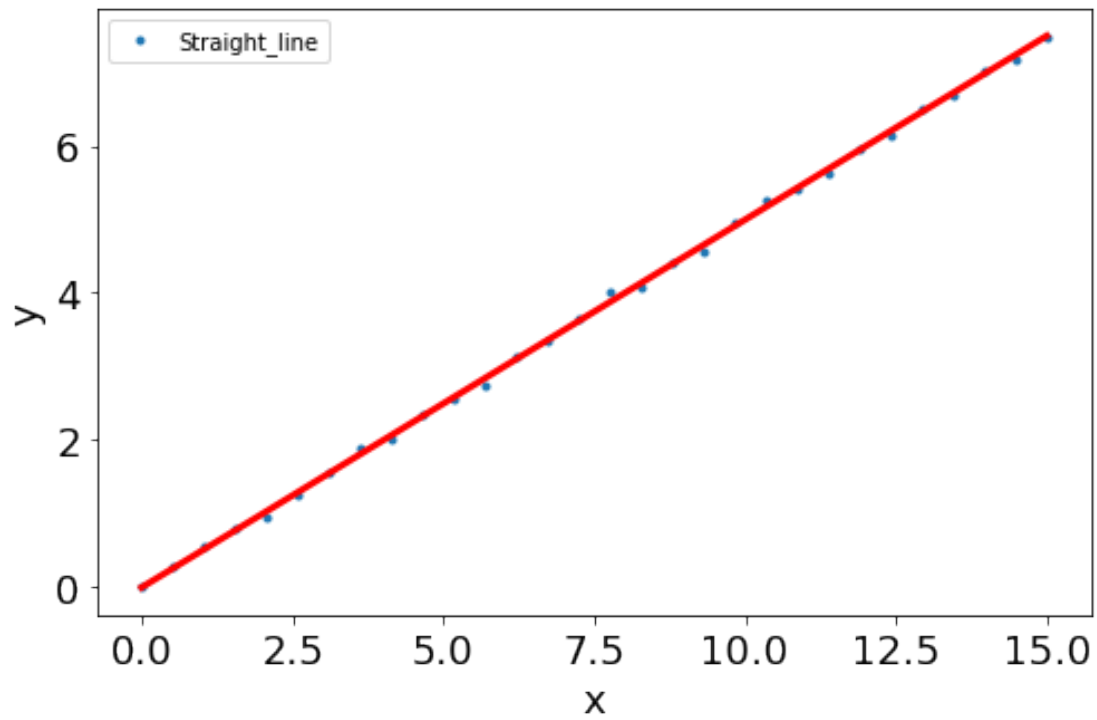
```
ID: test_data
Model: straightline
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C
```

```
(f) gradient      0.50094 ± 0.00014          95% CI[  0.50090,    0.50097]
(f) intercept    -0.01329 ± 0.00120          95% CI[ -0.01359,   -0.01299]
```

R²: 0.99938

=====

```
In [10]: # Now lets plot the data with the fit
test_data.plot()
```



NoteBook End

3 SI Notebook 1 - Fitting equilibrium and kinetic folding data to simple models

[Author] ERGM

In this notebook we will show how equilibrium and kinetic folding data can be imported into a notebook and fitted to folding models. We also show how you can automate fitting data to a series of models and automate using different starting parameters/variables!

If you are less script/computer orientated, you can simply change the data paths/variables, etc for your proteins and re-run the jupyter notebook ("Kernal/Restart & Run all" from the menu above).

3.1 Data Format

Datasets should be in .csv files where:

1. The 1st row should contain the data titles
 2. the 1st column should contain the x-values
 3. The subsequent columns should contain the y-values.
 4. You can have different data sets in different .csv files or all in one (as long as there is only one x-value column).
 5. If you wish to perform global analyses on folding or equilibrium data, the datasets concerned must be in the same .csv
 6. Except for global analyses using the Ising model, here each dataset must have its own .csv
-

Example .csv structure:

[Urea] (M)	Fraction Unfolded FKBP12
0	-0.00207
0.267	0.00307
...	...

First off lets load pyfolding & pyplot into this ipython notebook (pyplot allows us to plot more complex figures of our results):

```
In [1]: # use this command to tell Jupyter to plot figures inline with the text
        %matplotlib inline
```



```
# import pyfolding, the pyfolding models and ising models
import pyfolding
from pyfolding import *

# import the package for plotting, call it plt
import matplotlib.pyplot as plt

# import numpy as well
import numpy as np
```

<IPython.core.display.Javascript object>

PyFolding: Jupyter autoscrolling has been disabled

Now, we need to load some data to analyse.

I will import the equilibrium denaturation & unfolding/folding kinetics of wild-type FKBP12 from:

Main E.R.G., Fulton K.F. & Jackson S.E. (1999) ‘The folding pathway of FKBP12 and characterisation of the transition state.’ Journal of Molecular Biology, 291, 429-444.

3.1.1 Considerations

1. Kinetics data should be entered as rate constants (k) and NOT as the log of the rate constant.
2. There can be no “empty” x-axis cells in the .csv file for kinetics data.

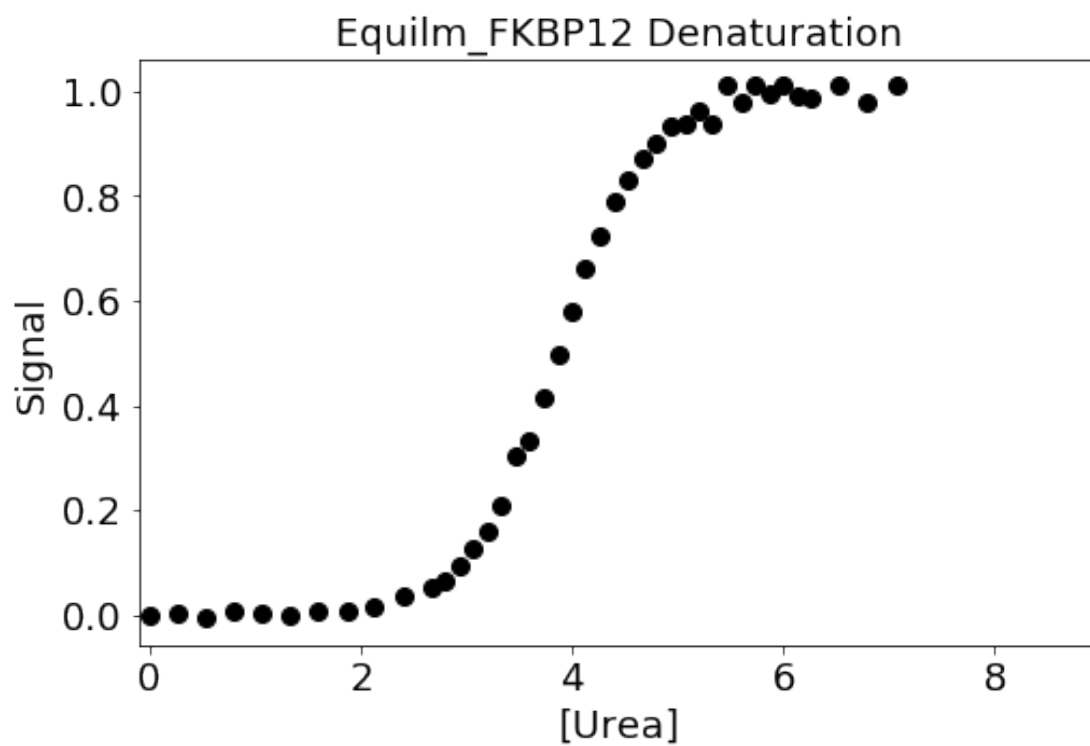
```
In [2]: # start by loading a data set
        # arguments are "path", "filename"

pth = "../examples/FKBP12/"
Equilm_FKBP12 = pyfolding.read_equilibrium_data(pth, "Equilm_FKBP12.csv")
Kinetics_FKBP12 = pyfolding.read_kinetic_data(pth, "Kinetics_FKBP12.csv")
```

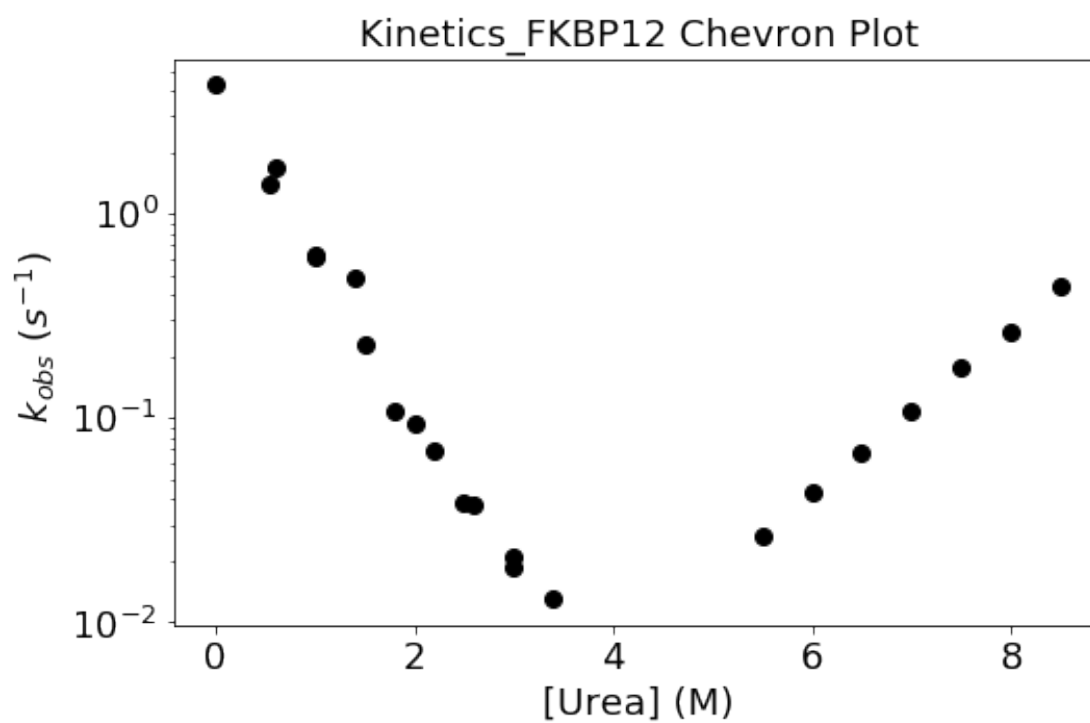
3.2 Plotting Data

OK, now we can try plotting the data to protein folding models.

```
In [3]: # now plot the equilm data
        Equilm_FKBP12.plot()
```



```
In [4]: # now the kinetics  
Kinetics_FKBP12.plot()
```



3.3 Fitting Data

OK, now we can try fitting the data to protein folding models. We will start by fitting the equilibrium data to a two state folding model, without sloping baselines (as the data has been processed as Fraction Unfolded).

3.3.1 List the models in pyfolding:

```
In [5]: # Command imports pyfolding models
        from pyfolding.models import *

        # command lists models
        list_models()

        # After the model name:
        # 'Verified: True' - model rigorously tested and it functions as expected.
        # 'Verified:False' - model tested, but not rigourously.
```

```
Out[5]: [('ChevronPolynomialFit', 'Verified: True'),
          ('HeteropolymerIsingEquilibrium', 'Verified: False'),
          ('HomozipperIsingEquilibrium', 'Verified: True'),
          ('ParallelTwoStateChevron', 'Verified: False'),
          ('ParallelTwoStateUnfoldingChevron', 'Verified: False'),
          ('TemplateModel', 'Verified: False'),
          ('ThreeStateChevron', 'Verified: True'),
          ('ThreeStateDimericIEquilibrium', 'Verified: True'),
          ('ThreeStateEquilibrium', 'Verified: True'),
          ('ThreeStateFastPhaseChevron', 'Verified: True'),
          ('ThreeStateMonoIEquilibrium', 'Verified: True'),
          ('ThreeStateSequentialChevron', 'Verified: True'),
          ('TwoStateChevron', 'Verified: True'),
          ('TwoStateChevronMovingTransition', 'Verified: True'),
          ('TwoStateDimerEquilibrium', 'Verified: True'),
          ('TwoStateEquilibrium', 'Verified: True'),
          ('TwoStateEquilibriumSloping', 'Verified: True')]
```

3.3.2 Print a model and its parameters, etc:

```
In [6]: # print the info
        models.TwoStateEquilibrium().info()
```

$$F = \frac{\exp(m(x - d_{50}))/RT}{1 + \exp(m(x - d_{50}))/RT}$$

Two state equilibrium denaturation curve - No sloping baseline.

Folding Scheme:

N <-> D

Params:

F = Fraction unfolded

m = m-value

x = denaturant concentration (M)

d50 = denaturant midpoint (M)

R = Universal Gas Constant (kcal.mol⁻¹.K⁻¹)

T = Temperature (Kelvin)

Reference:

Clarke and Fersht. Engineered disulfide bonds as probes of the folding pathway of barnase: Increasing the stability of proteins against the rate of denaturation. Biochemistry (1993) vol. 32 (16) pp. 4322-4329

```
In [7]: # Printing the model variables
```

```
# The function has two parts:
# part1 states the model you want to find about: "*****()"
# part2 prints the variables: ".fit_func_args"
```

```
TwoStateEquilibrium().fit_func_args
```

```
Out[7]: ['m', 'd50']
```

3.3.3 Fitting the Data.

Or we can skip straight to fitting the data.

```
In [8]: # Set temperature to 25.00°C
# (NOTE: Careful, this sets the temperature for all subsequent calculations)
pyfolding.set_temperature(25.)

#1st select the fit function and associates it with the data
Equilm_FKBP12.fit_func = models.TwoStateEquilibrium
```

```

#then fit it.
#in the brackets you can define starting values for the variables -
#input in the order printed above with the ".fit_func_args"
Equilm_FKBP12.fit(p0=[2,4])

# We can print the resultant graph using either of these commands:
# 1. "Equilm_FKBP12.plot()" or
# 2. "pyfolding.plot_equilibrium(Equilm_FKBP12)"

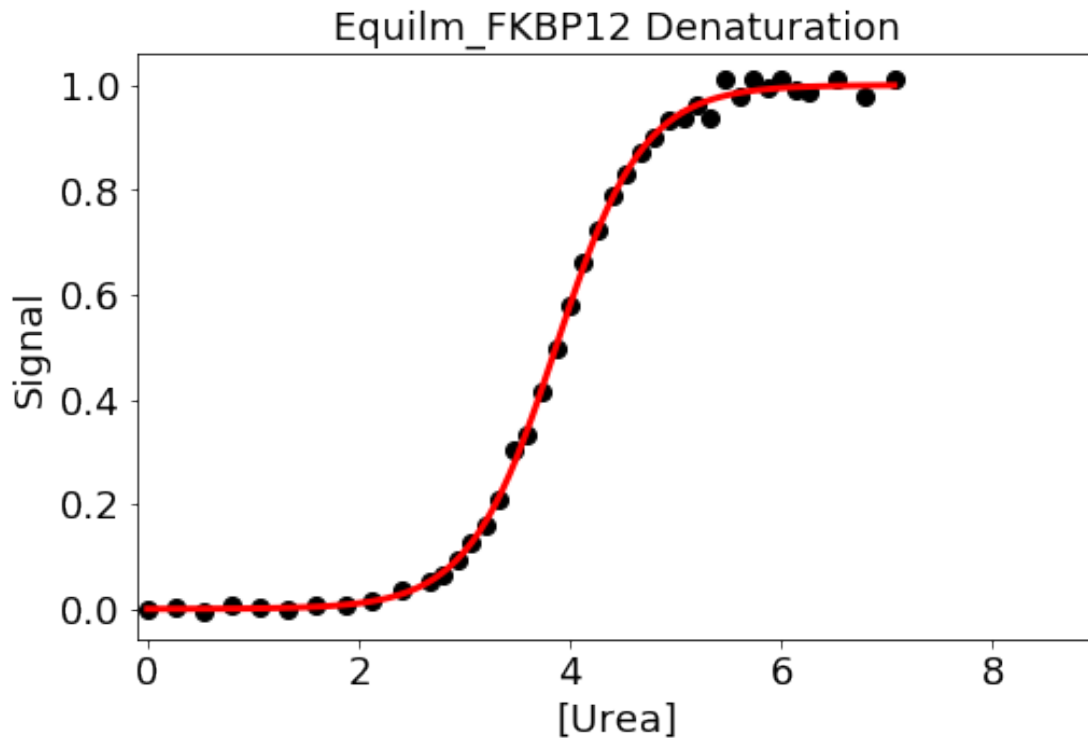
# lets do the first & save out the resultant graph

Equilm_FKBP12.plot(save='/Users/ergm/Desktop/test.pdf')

Set temperature to 25.00°C
(NOTE: Careful, this sets the temperature for all subsequent calculations)
=====
Fitting results
=====
ID: Equilm_FKBP12
Model: TwoStateEquilibrium
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(f) m      1.43404 ± 0.00022      95% CI[  1.43399,   1.43410]
(f) d50    3.86730 ± 0.00007      95% CI[  3.86728,   3.86732]
-----
R^2: 0.99933
=====

```



```
In [9]: # save out the fit
Equilm_FKBP12.save_fit('/users/ergm/test.csv')
```

We can fit the kinetic data separately too, lets first print the equation we want to use:

```
In [10]: # printing the equation for viewing
models.TwoStateChevron().info()

# Lets also see what the input variables are also
models.TwoStateChevron().fit_func_args
```

$$k_{obs} = k_f + k_u$$

where:

$$k_f = k_f^{H_2O} \exp(-m_{kf}x)$$

$$k_u = k_u^{H_2O} \exp(m_{ku}x)$$

thus:

$$k_{obs} = k_f^{H_2O} \exp(-m_{kf}x) + k_u^{H_2O} \exp(m_{ku}x)$$

(1)

Two state chevron plot.

Folding Scheme:
N \leftrightarrow D

Params:

k obs = rate constant of unfolding or refolding at a particular denaturant concentration
kf = rate constant of refolding at a particular denaturant concentration
mf = the gradient of refolding arm of the chevron
ku = rate constant of unfolding at a particular denaturant concentration
mu = the gradient of unfolding arm of the chevron
x = denaturant concentration (M)

Reference:

Jackson SE and Fersht AR. Folding of chymotrypsin inhibitor 2.
1. Evidence for a two-state transition.
Biochemistry (1991) 30(43):10428-10435.

```
Out[10]: ['kf', 'mf', 'ku', 'mu']
```

```
In [11]: # 1st select the fit function and associates it with the data
Kinetics_FKBP12.fit_func = models.TwoStateChevron
```

```
# 2nd fit the data with initial values (input these as per the list of input variables
Kinetics_FKBP12.fit(p0=[4,2,0.0001,1])
```

```
=====
Fitting results
=====
```

ID: Kinetics_FKBP12

Model: TwoStateChevron

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

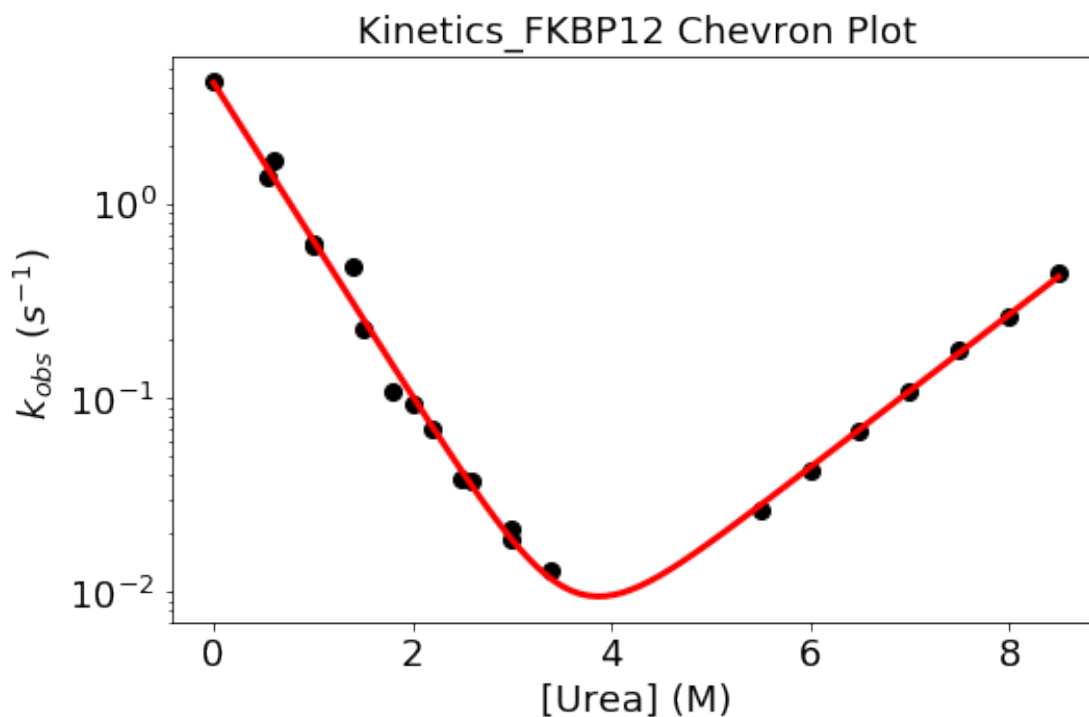
Temperature: 25.00°C

(f) kf	4.21728 \pm 0.05339	95% CI[4.20408,	4.23048]
(f) mf	1.86223 \pm 0.00703	95% CI[1.86049,	1.86396]
(f) ku	0.00019 \pm 0.00001	95% CI[0.00019,	0.00019]
(f) mu	0.90557 \pm 0.00799	95% CI[0.90360,	0.90755]

```
-----
R^2: 0.99234
=====
```

We can print the resultant graph: This is the same as when plotting the Equilm data and the commands are either: `Kinetics_FKBP12.plot()` or `pyfolding.plot_chevron(Kinetics_FKBP12)`
Let's use the first

```
In [12]: Kinetics_FKBP12.plot()
```

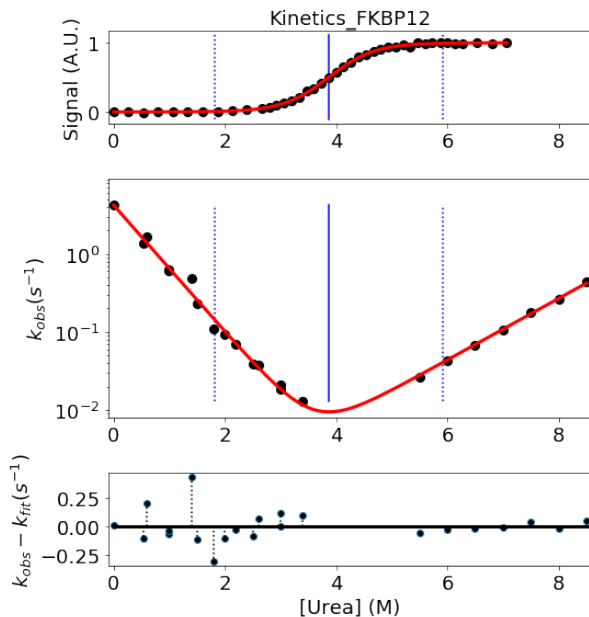


3.4 Plotting Data to a Fancier Graph

We can also plot a fancier graph that shows both Equilm and Kinetics together

```
In [13]: pyfolding.plot_figure(Equilm_FKBP12,
                                Kinetics_FKBP12,
                                display=True,
                                pth= '/users/ergm/desktop', save=True)

# the "pth" and save=True part writes out a pdf of the figure below.
```

Data-set: Equilm_FKBP12

Equilibrium Model: TwoStateEquilibrium

m: 1.43404 ± 0.00022

d50: 3.86730 ± 0.00007

Folding midpoint: 3.87 M

R^2 : 1.00

Kinetic Model: TwoStateChevron

Fit Standard Error: 0.03

kf: $4.22\text{e}+00 \pm 5.34\text{e}-02$

mf: $1.86\text{e}+00 \pm 7.03\text{e}-03$

ku: $1.92\text{e}-04 \pm 1.08\text{e}-05$

mu: $9.06\text{e}-01 \pm 7.99\text{e}-03$

R^2 : 0.99

3.5 Fit to multiple models

```
In [14]: # make a list of models to be used to fit
models_to_fit = [models.TwoStateEquilibrium,
                  models.TwoStateEquilibriumSloping]

# and now lets fit them
for model in models_to_fit:
    Equilm_FKBP12.fit_func = model
    Equilm_FKBP12.fit()
    Equilm_FKBP12.plot()
```

=====
Fitting results
=====

ID: Equilm_FKBP12

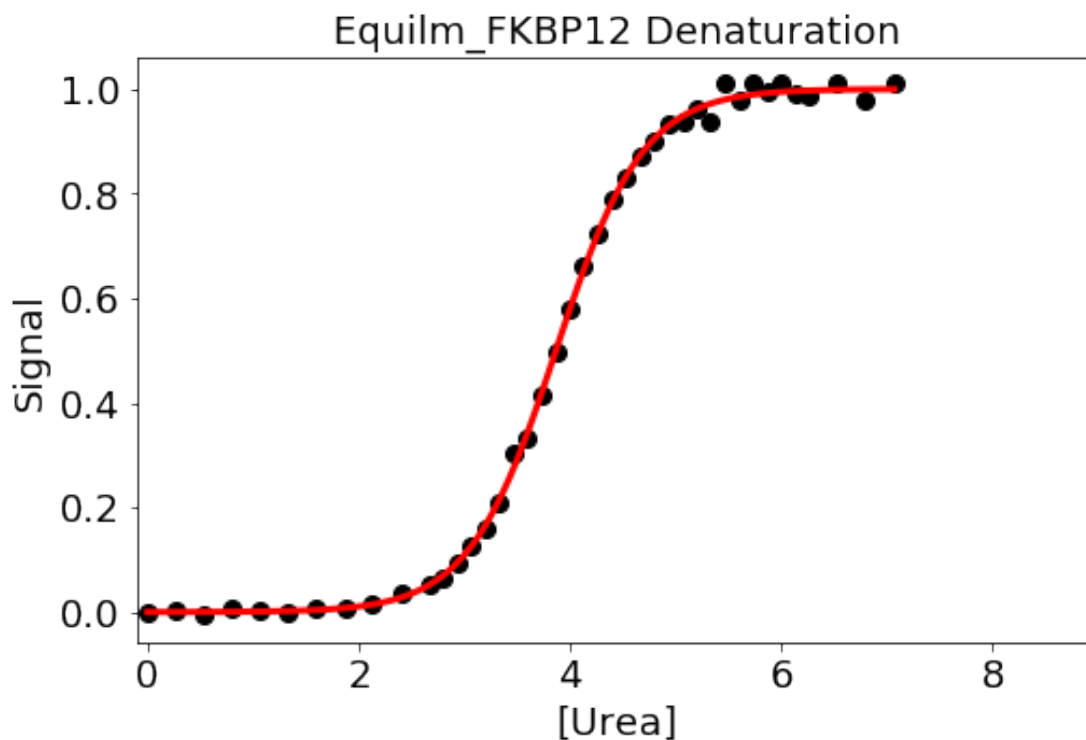
Model: TwoStateEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) m	1.43404 ± 0.00022	95% CI[1.43399,	1.43410]
(f) d50	3.86730 ± 0.00007	95% CI[3.86728,	3.86732]

R²: 0.99933



Fitting results

ID: Equilm_FKBP12

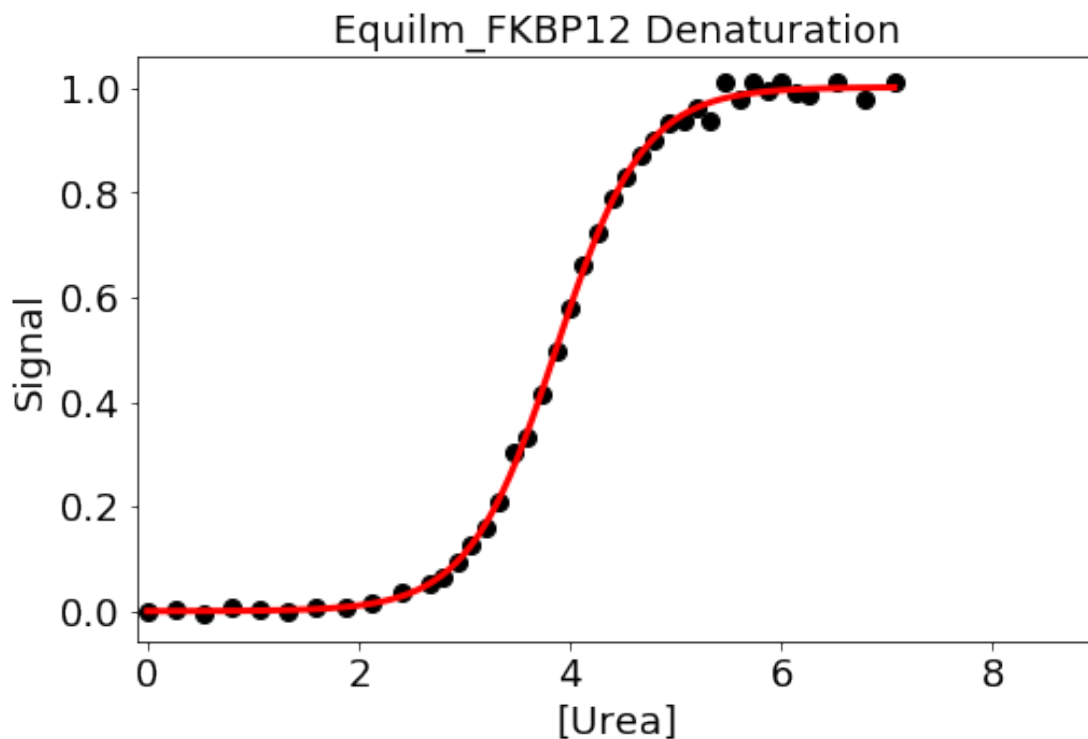
Model: TwoStateEquilibriumSloping

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) α_f	-0.00000 ± 0.00008	95% CI[$-0.00002,$	$0.00002]$
(f) β_f	-0.00005 ± 0.00006	95% CI[$-0.00007,$	$-0.00004]$
(f) α_u	0.99945 ± 0.00051	95% CI[$0.99932,$	$0.99958]$
(f) β_u	0.00035 ± 0.00008	95% CI[$0.00033,$	$0.00037]$
(f) m	1.43029 ± 0.00058	95% CI[$1.43014,$	$1.43043]$
(f) d50	3.86799 ± 0.00021	95% CI[$3.86793,$	$3.86804]$

R²: 0.99933



As you can see, each of the models has found a good fit with the default initial variables.

We can also get PyFolding to iterate over a range of initial parameters used (incase, for example, fitting your data is very dependent on your starting parameters).

```
In [15]: #1st select the fit function and associates it with the data
Equilm_FKBP12.fit_func = models.TwoStateEquilibrium

# iterate over some different parameters
for m in xrange(1,3):
    for d50 in xrange (2,4):
        Equilm_FKBP12.fit(p0=[m,d50])
```

=====

Fitting results

=====

ID: Equilm_FKBP12

Model: TwoStateEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) m	1.43404 ± 0.00022	95% CI[1.43399,	1.43410]
-------	-------------------	---------	----------	----------

(f) d50	3.86730 ± 0.00007	95% CI[3.86728,	3.86732]
---------	-------------------	---------	----------	----------

R^2: 0.99933

=====

=====

Fitting results

=====

ID: Equilm_FKBP12

Model: TwoStateEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) m	1.43404 ± 0.00022	95% CI[1.43399,	1.43410]
-------	-------------------	---------	----------	----------

(f) d50	3.86730 ± 0.00007	95% CI[3.86728,	3.86732]
---------	-------------------	---------	----------	----------

R^2: 0.99933

=====

=====

Fitting results

=====

ID: Equilm_FKBP12

Model: TwoStateEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) m	1.43404 ± 0.00022	95% CI[1.43399,	1.43410]
-------	-------------------	---------	----------	----------

(f) d50	3.86730 ± 0.00007	95% CI[3.86728,	3.86732]
---------	-------------------	---------	----------	----------

R^2: 0.99933

=====

=====

Fitting results

=====

ID: Equilm_FKBP12

Model: TwoStateEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) m	1.43404 ± 0.00022	95% CI[1.43399,	1.43410]
-------	-----------------------	---------	----------	----------

(f) d50	3.86730 ± 0.00007	95% CI[3.86728,	3.86732]
---------	-----------------------	---------	----------	----------

R^2: 0.99933
=====

End of this Notebook.

4 SI Notebook 2 - Fitting more complicated models to some example data

[Author] ERGM

In this notebook we will fit more complicated models to some example data. Here the idea is to show that we can fit multiple models in series to folding data. The models are slightly more complex than two-state folding (Pyfolding SI Notebook 1.) These models can be used with in bought software, but with PyFolding we can automate and is Free!

If you are less script/computer orientated, you can simply change the data paths/variables, etc for your proteins and re-run the jupyter notebook ("Kernal/Restart & Run all" from the menu above).

In this notebook I will be using data from the paper below:

Mapping the Topography of a Protein Energy Landscape

Hutton, R. D., Wilkinson, J., Faccin, M., Sivertsson, E. M., Pelizzola, A.,
Lowe, A. R., Bruscolini, P. & Itzhaki, L. S. J Am Chem Soc (2015) 127, 46: 14610-25

[<http://pubs.acs.org/doi/10.1021/jacs.5b07370>]

4.1 Data Format

See PyFolding SI Notebook 1 for how to set up your .csv files for import

4.1.1 Special Considerations

1. If you wish to perform global analyses on folding data, the datasets of the same WT/mutant concerned must be in the same .csv
2. Kinetics data should be entered as rate constants (k) and NOT as the log of the rate constant.
3. There can be no "empty" cells in the x-axis variables in the .csv file for kinetics data.

Example .csv structure:

urea	k1	k2
8.59	1.72	
8.77	1.96	
9.00	2.41	
0.54	5.44	23.6
0.79	5.94	21.2
1.02	6.56	21.6
...

```
In [1]: # As in previous notebook, lets load pyfolding & pyplot into this ipython notebook
        #(pyplot allows us to plot more complex figures of our results):
```

```
%matplotlib inline
import pyfolding
from pyfolding import models

# let's use some other libraries also
import matplotlib.pyplot as plt
import numpy as np
```

<IPython.core.display.Javascript object>

PyFolding: Jupyter autoscrolling has been disabled

Now, we need to load some data to analyse.

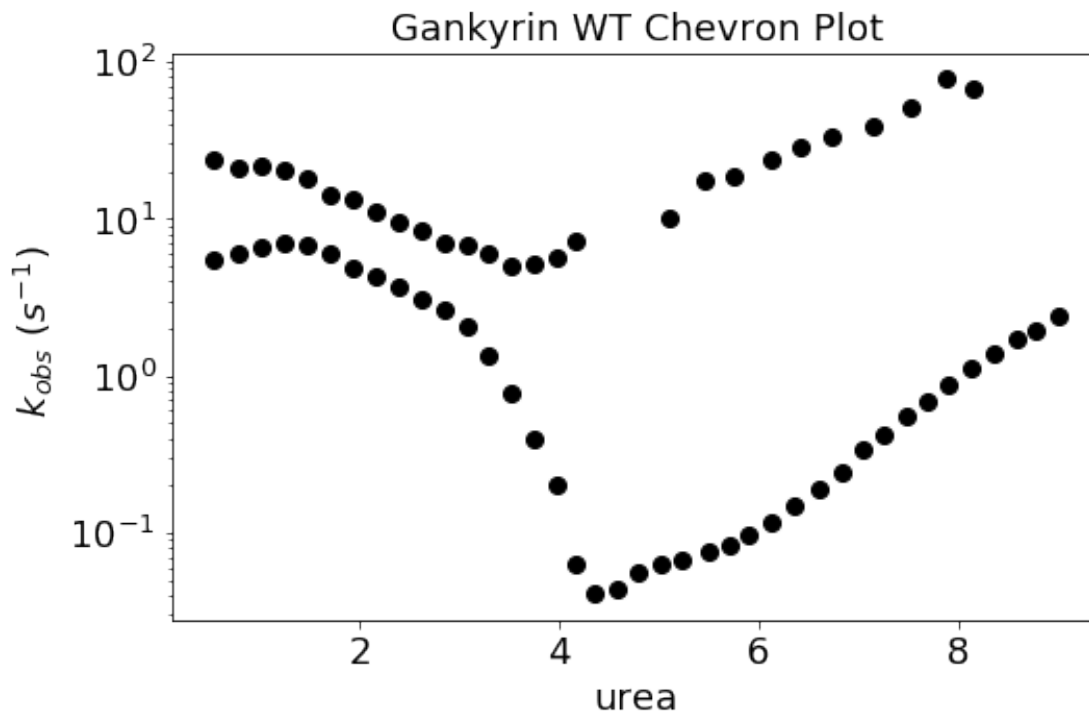
```
In [2]: # loading the data -
pth = "../examples/Gankyrin"
GankyrinChevron = pyfolding.read_kinetic_data(pth, "GankyrinWTChevron.csv")

# let's give this dataset a good name
GankyrinChevron.ID = 'Gankyrin WT'
```

4.2 Plotting Chevron phases

Let's plot the chevron phases:

```
In [3]: # easy plotting of the entire dataset ...
GankyrinChevron.plot()
```



Note that the measurements were performed at 25°C. This is the default for pyFolding. However lets set this to show you how.

```
In [4]: pyfolding.set_temperature(25.0)
```

Set temperature to 25.00°C

(NOTE: Careful, this sets the temperature for all subsequent calculations)

4.3 Fitting Data To Multiple Models

4.3.1 List Models

Lets first list the models that are already in PyFolding.

We can list the models in pyfolding:

```
In [5]: # Command imports pyfolding models
        from pyfolding.models import *

        # command lists models
        list_models()

        # After the model name:
        # 'Verified: True' - the model rigourously tested and it functions as expected.
        # 'Verified:False' - the model has not been rigourously tested.
```

```
Out[5]: [('ChevronPolynomialFit', 'Verified: True'),
          ('HeteropolymerIsingEquilibrium', 'Verified: False'),
          ('HomozipperIsingEquilibrium', 'Verified: True'),
          ('ParallelTwoStateChevron', 'Verified: False'),
          ('ParallelTwoStateUnfoldingChevron', 'Verified: False'),
          ('TemplateModel', 'Verified: False'),
          ('ThreeStateChevron', 'Verified: True'),
          ('ThreeStateDimericIEquilibrium', 'Verified: True'),
          ('ThreeStateEquilibrium', 'Verified: True'),
          ('ThreeStateFastPhaseChevron', 'Verified: True'),
          ('ThreeStateMonoIEquilibrium', 'Verified: True'),
          ('ThreeStateSequentialChevron', 'Verified: True'),
          ('TwoStateChevron', 'Verified: True'),
          ('TwoStateChevronMovingTransition', 'Verified: True'),
          ('TwoStateDimerEquilibrium', 'Verified: True'),
          ('TwoStateEquilibrium', 'Verified: True'),
          ('TwoStateEquilibriumSloping', 'Verified: True')]
```

4.3.2 Fit to multiple models

Now lets fit to multiple models

```
In [9]: # make a list of models to be used to fit in the square brackets
        test_models = [models.TwoStateChevron,
                        models.ThreeStateChevron,
                        models.ThreeStateSequentialChevron,
                        models.ThreeStateFastPhaseChevron,
                        models.TwoStateChevronMovingTransition,
                        models.ChevronPolynomialFit]

        #now tell Pyfolding to fit your data to each model
        for model in test_models:
            GankyrinChevron.fit_func = model    # selects fit function & associates with data
```

```
GankyrinChevron.fit()      # then fit it.
GankyrinChevron.plot()     # plots it. Add (components=True) to plot component k's.
```

=====

Fitting results

=====

ID: Gankyrin WT

Model: TwoStateChevron

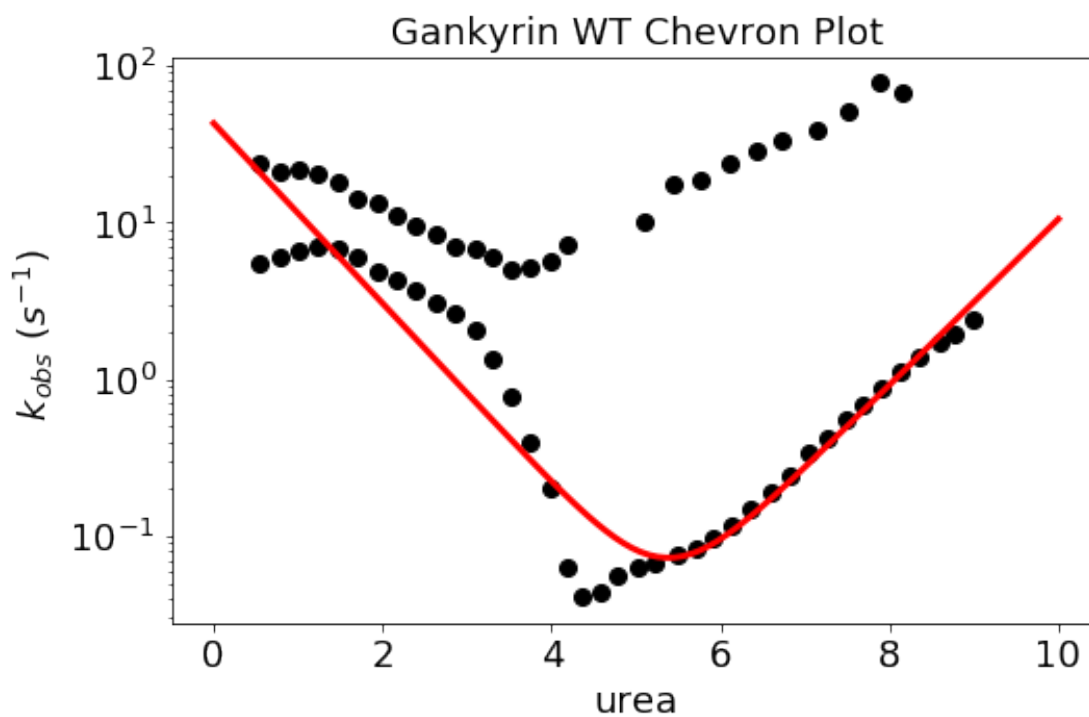
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) kf	42.82617 ± 7.83661	95% CI[40.86319,	44.78916]
(f) mf	1.32179 ± 0.06202	95% CI[1.30625,	1.33733]
(f) ku	0.00005 ± 0.00004	95% CI[0.00004,	0.00006]
(f) mu	1.21849 ± 0.09285	95% CI[1.19524,	1.24175]

R²: 0.89052

=====



=====

Fitting results

=====

ID: Gankyrin WT

Model: ThreeStateChevron

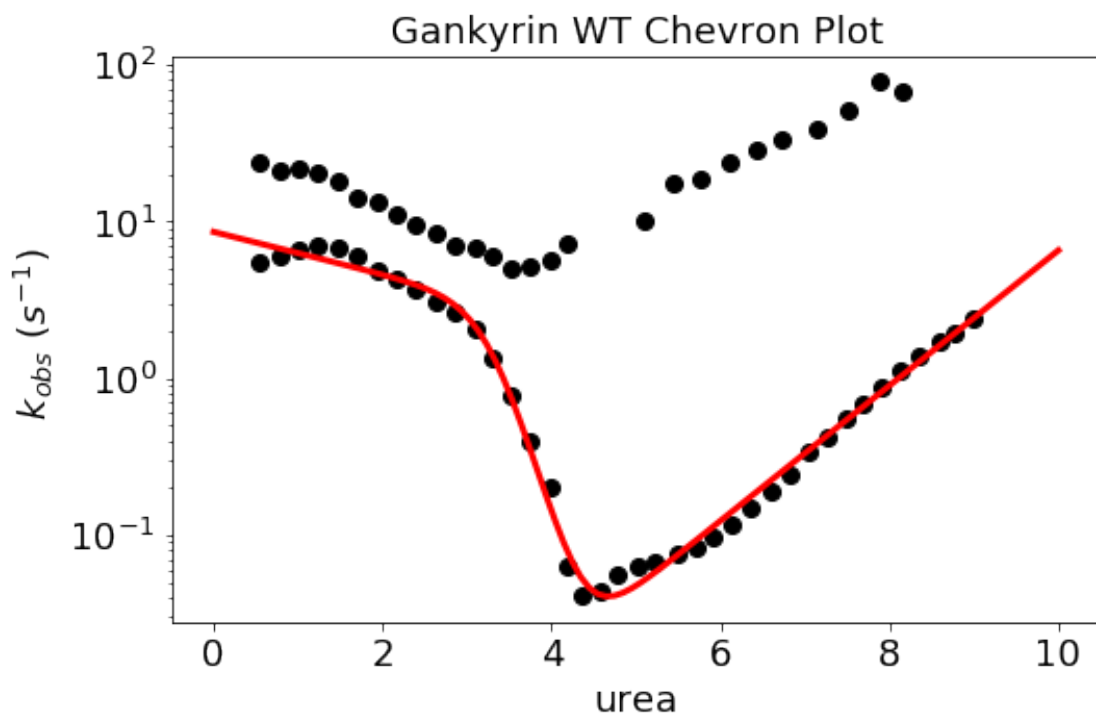
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) kfi	0.00032 ± 0.00001	95% CI[0.00032,	0.00032]
(f) mif	-0.99293 ± 0.00453	95% CI[-0.99406,	-0.99179]
(f) kif	$3088061.04961 \pm 591967.18085$	95% CI[2939906.24233,	3236215.85690]
(f) mi	-5.23401 ± 0.04790	95% CI[-5.24600,	-5.22202]
(f) Kiu	0.00000 ± 0.00000	95% CI[0.00000,	0.00000]
(f) mu	-1.30465 ± 0.01243	95% CI[-1.30776,	-1.30154]

R²: 0.99254

=====



=====

Fitting results

=====

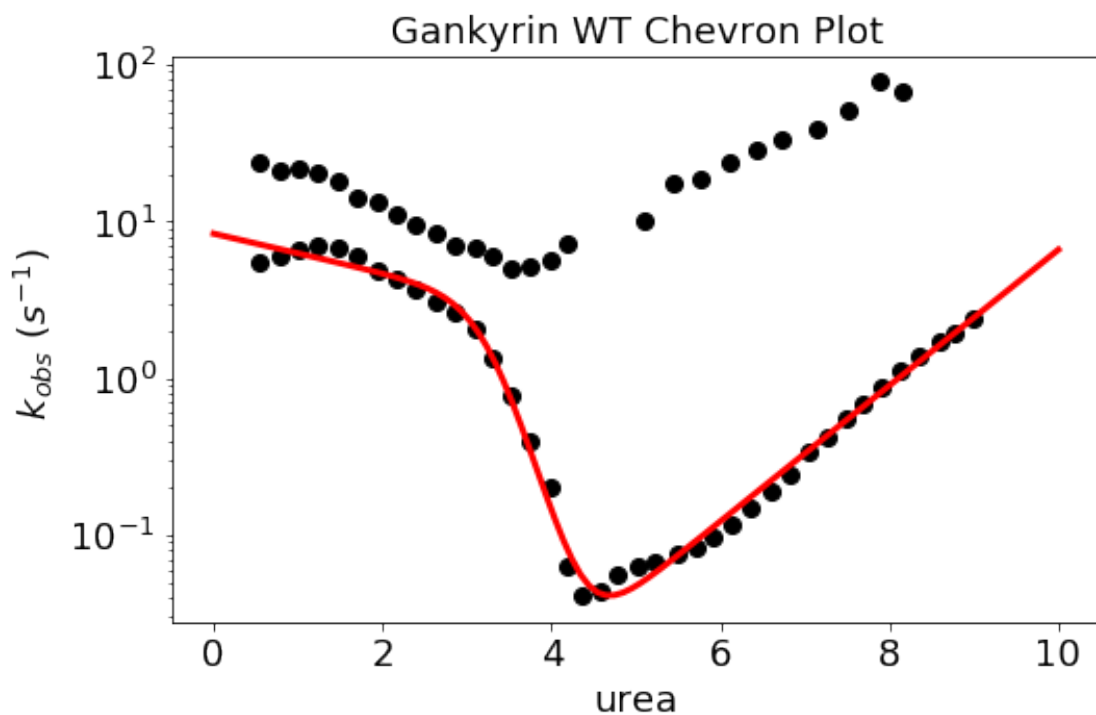
ID: Gankyrin WT

Model: ThreeStateSequentialChevron

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
 Temperature: 25.00°C

(f) kui	$8.35101 \pm \text{inf}$	95% CI[-inf,	inf]
(f) mui	$0.28903 \pm \text{inf}$	95% CI[-inf,	inf]
(c) kiu	10000.00000			
(c) miu	0.00000			
(f) kif	$1976594059.17404 \pm \text{inf}$	95% CI[-inf,	inf]
(f) mif	$3.79062 \pm \text{inf}$	95% CI[-inf,	inf]
(f) kfi	$0.00031 \pm \text{inf}$	95% CI[-inf,	inf]
(f) mfi	$0.99698 \pm \text{inf}$	95% CI[-inf,	inf]

R²: 0.99250



Fitting results

ID: Gankyrin WT

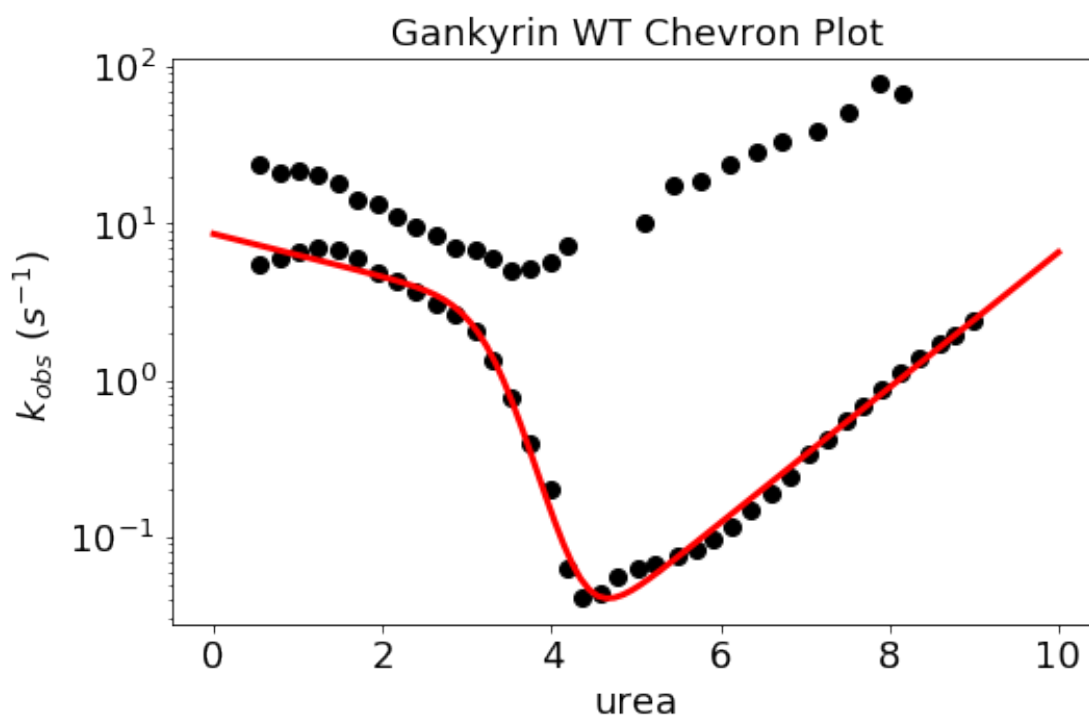
Model: ThreeStateFastPhaseChevron

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) kui	726.43137 ± 328693556.72880	95% CI[-82183999.71540, 82185452.57815]
(f) mui	2.50450 ± 254385.90101	95% CI[-63602.74477, 63607.75378]
(f) kiu	0.00101 ± 456.74252	95% CI[-114.20037, 114.20239]
(f) miu	1.42452 ± 254385.89377	95% CI[-63603.82294, 63606.67198]
(f) kif	6166648.24920 ± 1373603.37261	95% CI[5823200.02911, 6510096.46929]
(f) mif	4.24069 ± 0.05691	95% CI[4.22646, 4.25492]
(f) kfi	0.00032 ± 0.00001	95% CI[0.00032, 0.00032]
(f) mfi	0.99294 ± 0.00484	95% CI[0.99172, 0.99415]

R²: 0.99254



Fitting results

ID: Gankyrin WT

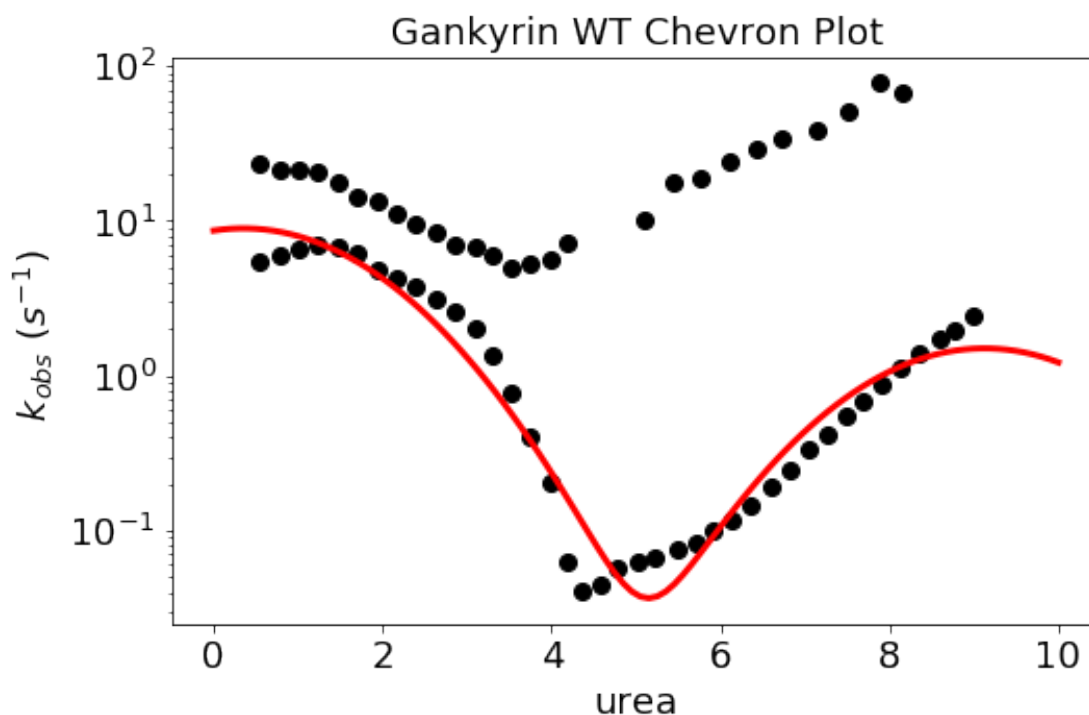
Model: TwoStateChevronMovingTransition

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) ku	0.00000 ± 0.00000	95% CI[0.00000,	0.00000]
(f) mu	4.98449 ± 0.25547	95% CI[4.92052,	5.04845]
(f) kf	8.64128 ± 1.19561	95% CI[8.34192,	8.94064]
(f) mf	0.19051 ± 0.10469	95% CI[0.16429,	0.21672]
(f) m_prime	-0.27314 ± 0.01801	95% CI[-0.27764,	-0.26863]

R²: 0.94901



Fitting results

ID: Gankyrin WT

Model: ChevronPolynomialFit

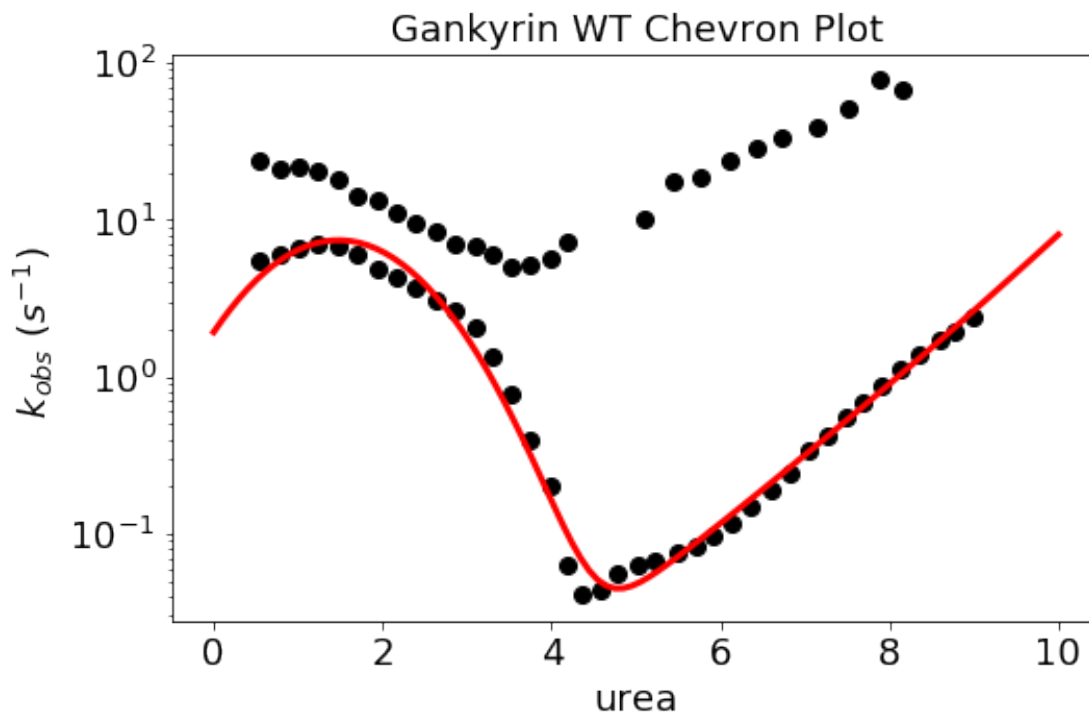
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) ku	0.00062 ± 0.00020	95% CI[0.00056,	0.00067]
(f) mu	0.76331 ± 0.09633	95% CI[0.73920,	0.78742]
(f) mu_prime	0.01845 ± 0.00691	95% CI[0.01672,	0.02018]

(f) kf	1.92293 ± 0.09362	95% CI[1.89950,	1.94636]
(f) mf	1.82503 ± 0.04649	95% CI[1.81339,	1.83666]
(f) mf_prime	-0.61791 ± 0.00971	95% CI[-0.62034,	-0.61548]

R²: 0.98724
=====



4.3.3 Fit to multiple models and output with prettier graphics

We can also fit to multiple models and output the “prettier” graphics (as in PyFolding SI Notebook 1)

```
In [7]: # First load in the Equilm denaturation (as per PyFolding SI Notebook 1)
pth = "../examples/Gankyrin"
GankyrinEquilm = pyfolding.read_equilibrium_data(pth, "GankyrinEquilmDenaturationCurve.csv")

# 1st select the fit function and associates it with the data
```

```

GankyrinEquilm.fit_func = models.TwoStateEquilibriumSloping

# then fit it.
# the brackets enable you to define starting values for the variables see notebook 1
GankyrinEquilm.fit(p0=[-37, 0.1, -15, 0.1, 3,4])

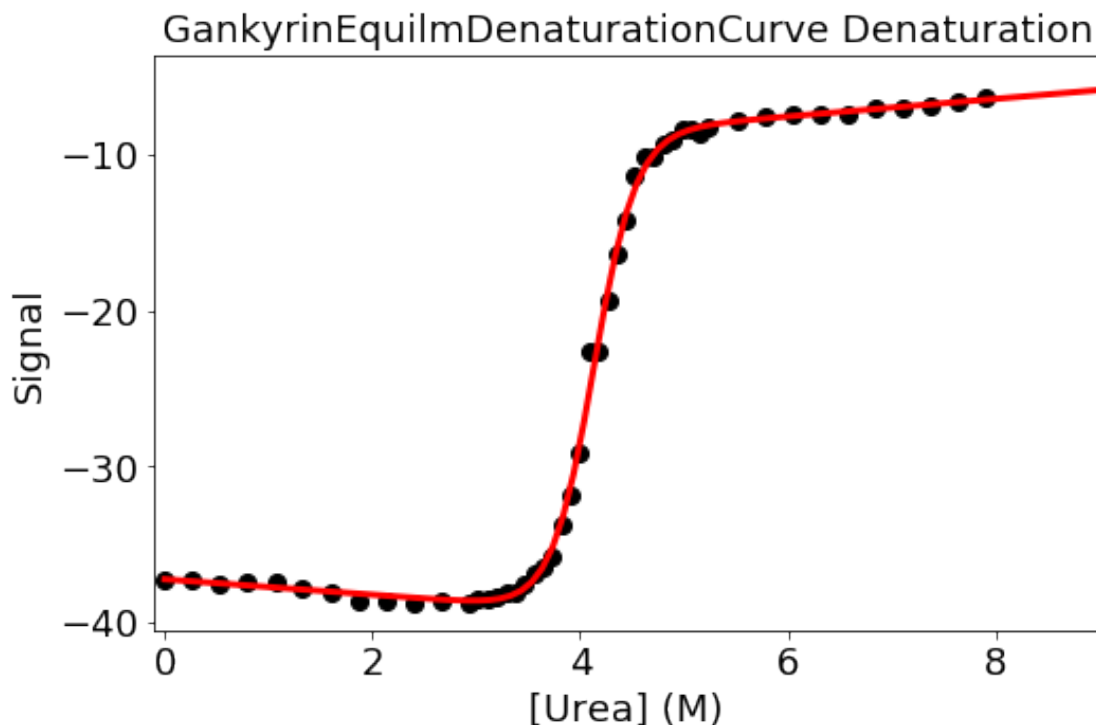
# and plot to check all it OK
GankyrinEquilm.plot()

=====
Fitting results
=====
ID: GankyrinEquilmDenaturationCurve
Model: TwoStateEquilibriumSloping
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(f) alpha_f  -37.25158 ± 0.14271          95% CI[ -37.28741,  -37.21575]
(f) beta_f    -0.49562 ± 0.06439          95% CI[  -0.51179,  -0.47946]
(f) alpha_u   26.33776 ± 0.48279          95% CI[ 26.21654,  26.45898]
(f) beta_u    1.06005 ± 0.09104          95% CI[  1.03719,   1.08291]
(f) m         2.90885 ± 0.05757          95% CI[  2.89439,   2.92330]
(f) d50       4.12424 ± 0.00486          95% CI[  4.12302,   4.12546]

-----
R^2: 0.99869
=====

```

```
In [8]: # again make a list of models to be used to fit in the square brackets
test_models = [models.TwoStateChevron,
               models.ThreeStateChevron,
               models.ThreeStateSequentialChevron,
               models.ThreeStateFastPhaseChevron,
               models.TwoStateChevronMovingTransition,
               models.ChevronPolynomialFit]

#now tell Pyfolding to fit your data to each model
for model in test_models:
    GankyrinChevron.fit_func = model # 1st select fit function & associate with data
    GankyrinChevron.fit() # then fit it.
    pyfolding.plot_figure(GankyrinEquilm, GankyrinChevron, display=True) # then plot
```

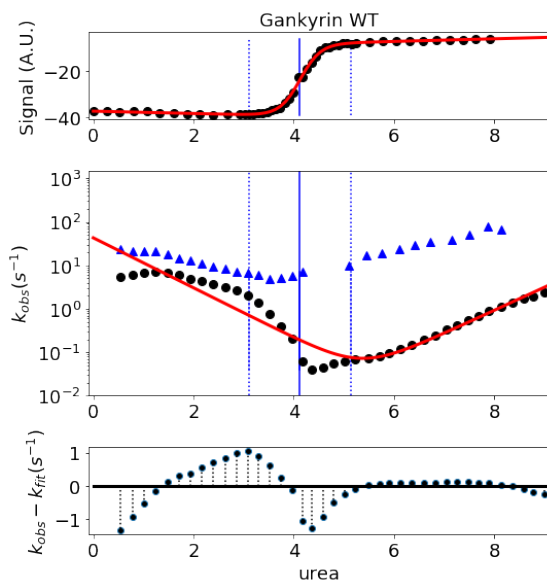
```
=====
Fitting results
=====
```

```
ID: Gankyrin WT
Model: TwoStateChevron
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C
```

```
(f) kf    42.82617 ± 7.83661          95% CI[ 40.86319, 44.78916]
(f) mf     1.32179 ± 0.06202         95% CI[ 1.30625, 1.33733]
```

(f) ku	0.00005 ± 0.00004	95% CI[0.00004,	0.00006]
(f) mu	1.21849 ± 0.09285	95% CI[1.19524,	1.24175]

R²: 0.89052



Data-set: GankyrinEquilmDenaturationCurve

Equilibrium Model: TwoStateEquilibriumSloping
 α_f : -37.25158 ± 0.14271
 β_f : -0.49562 ± 0.06439
 α_u : 26.33776 ± 0.48279
 β_u : 1.06005 ± 0.09104
 m : 2.90885 ± 0.05757
 $d50$: 4.12424 ± 0.00486
 Folding midpoint: 4.12 M
 R^2 : 1.00

Kinetic Model: TwoStateChevron
 Fit Standard Error: 0.09
 k_f : $4.28e+01 \pm 7.84e+00$
 m_f : $1.32e+00 \pm 6.20e-02$
 k_u : $5.37e-05 \pm 3.76e-05$
 μ_u : $1.22e+00 \pm 9.29e-02$
 R^2 : 0.89

Fitting results

ID: Gankyrin WT

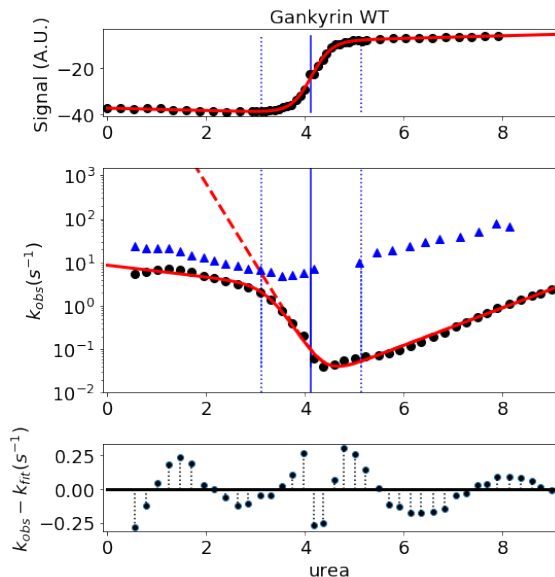
Model: ThreeStateChevron

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) kfi	0.00032 ± 0.00001	95% CI[0.00032,	0.00032]
(f) mif	-0.99293 ± 0.00453	95% CI[-0.99406,	-0.99179]
(f) kif	$3088061.04961 \pm 591967.18085$	95% CI[2939906.24233,	3236215.85690]
(f) mi	-5.23401 ± 0.04790	95% CI[-5.24600,	-5.22202]
(f) Kiu	0.00000 ± 0.00000	95% CI[0.00000,	0.00000]
(f) mu	-1.30465 ± 0.01243	95% CI[-1.30776,	-1.30154]

R²: 0.99254



Data-set: GankyrinEquilmDenaturationCurve

Equilibrium Model: TwoStateEquilibriumSloping

α_f : -37.25158 \pm 0.14271

β_f : -0.49562 \pm 0.06439

α_u : 26.33776 \pm 0.48279

β_u : 1.06005 \pm 0.09104

m: 2.90885 \pm 0.05757

d50: 4.12424 \pm 0.00486

Folding midpoint: 4.12 M

R^2 : 1.00

Kinetic Model: ThreeStateChevron

Fit Standard Error: 0.02

kfi: 3.19e-04 \pm 1.02e-05

mif: -9.93e-01 \pm 4.53e-03

kif: 3.09e+06 \pm 5.92e+05

mi: -5.23e+00 \pm 4.79e-02

Kiu: 2.77e-06 \pm 5.12e-07

mu: -1.30e+00 \pm 1.24e-02

R^2 : 0.99

Fitting results

ID: Gankyrin WT

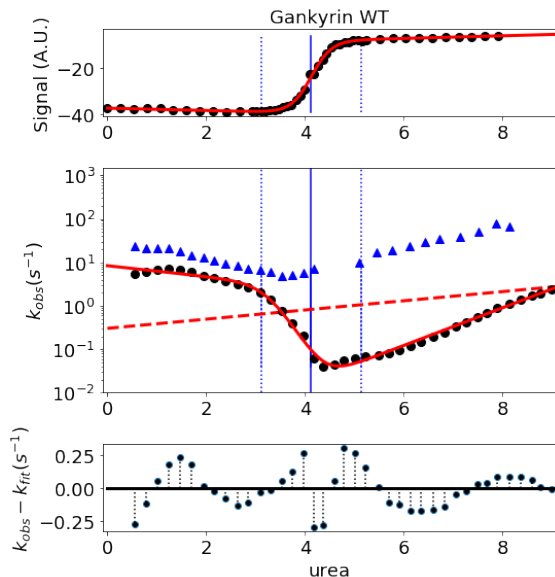
Model: ThreeStateSequentialChevron

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) kui	8.35101 \pm inf	95% CI[-inf,	inf]
(f) mui	0.28903 \pm inf	95% CI[-inf,	inf]
(c) kiu	10000.00000			
(c) miu	0.00000			
(f) kif	1976594059.17404 \pm inf	95% CI[-inf,	inf]
(f) mif	3.79062 \pm inf	95% CI[-inf,	inf]
(f) kfi	0.00031 \pm inf	95% CI[-inf,	inf]
(f) mfi	0.99698 \pm inf	95% CI[-inf,	inf]

R^2 : 0.99250



Data-set: GankyrinEquilmDenaturationCurve

Equilibrium Model: TwoStateEquilibriumSloping

α_f : -37.25158 \pm 0.14271

β_f : -0.49562 \pm 0.06439

α_u : 26.33776 \pm 0.48279

β_u : 1.06005 \pm 0.09104

m: 2.90885 \pm 0.05757

d50: 4.12424 \pm 0.00486

Folding midpoint: 4.12 M

R^2 : 1.00

Kinetic Model: ThreeStateSequentialChevron

Fit Standard Error: 0.02

k_{ui}: 8.35e+00 \pm inf

μ _{ui}: 2.89e-01 \pm inf

k_{iu}: 1.00e+04 \pm 0.00e+00

μ _{iu}: 0.00e+00 \pm 0.00e+00

k_{if}: 1.98e+09 \pm inf

μ _{if}: 3.79e+00 \pm inf

k_{fi}: 3.09e-04 \pm inf

μ _{fi}: 9.97e-01 \pm inf

R^2 : 0.99

Fitting results

ID: Gankyrin WT

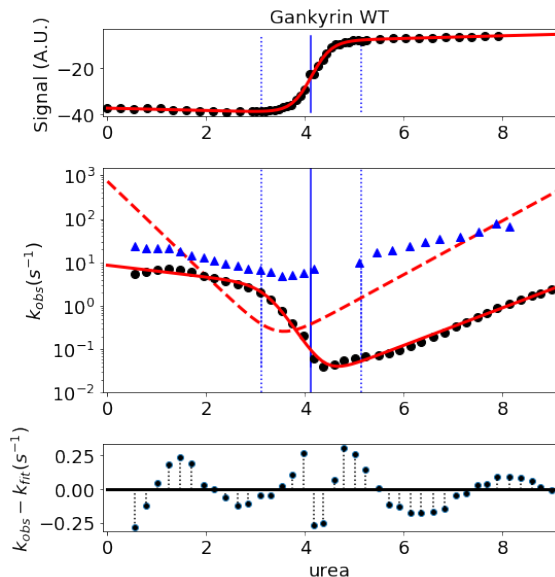
Model: ThreeStateFastPhaseChevron

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) kui	726.43137 \pm 328693556.72880	95% CI[-82183999.71540, 82185452.57815]
(f) μ _{ui}	2.50450 \pm 254385.90101	95% CI[-63602.74477, 63607.75378]
(f) k _{iu}	0.00101 \pm 456.74252	95% CI[-114.20037, 114.20239]
(f) μ _{iu}	1.42452 \pm 254385.89377	95% CI[-63603.82294, 63606.67198]
(f) k _{if}	6166648.24920 \pm 1373603.37261	95% CI[5823200.02911, 6510096.46929]
(f) μ _{if}	4.24069 \pm 0.05691	95% CI[4.22646, 4.25492]
(f) k _{fi}	0.00032 \pm 0.00001	95% CI[0.00032, 0.00032]
(f) μ _{fi}	0.99294 \pm 0.00484	95% CI[0.99172, 0.99415]

R^2 : 0.99254



Data-set: GankyrinEquilmDenaturationCurve

Equilibrium Model: TwoStateEquilibriumSloping

α_f : -37.25158 \pm 0.14271

β_f : -0.49562 \pm 0.06439

α_u : 26.33776 \pm 0.48279

β_u : 1.06005 \pm 0.09104

m: 2.90885 \pm 0.05757

d50: 4.12424 \pm 0.00486

Folding midpoint: 4.12 M

R^2 : 1.00

Kinetic Model: ThreeStateFastPhaseChevron

Fit Standard Error: 0.02

k_{ui}: 7.26e+02 \pm 3.29e+08

μ_i : 2.50e+00 \pm 2.54e+05

k_{iu}: 1.01e-03 \pm 4.57e+02

μ_u : 1.42e+00 \pm 2.54e+05

k_{if}: 6.17e+06 \pm 1.37e+06

m_{if}: 4.24e+00 \pm 5.69e-02

k_{fi}: 3.19e-04 \pm 1.09e-05

m_{fi}: 9.93e-01 \pm 4.84e-03

R^2 : 0.99

Fitting results

ID: Gankyrin WT

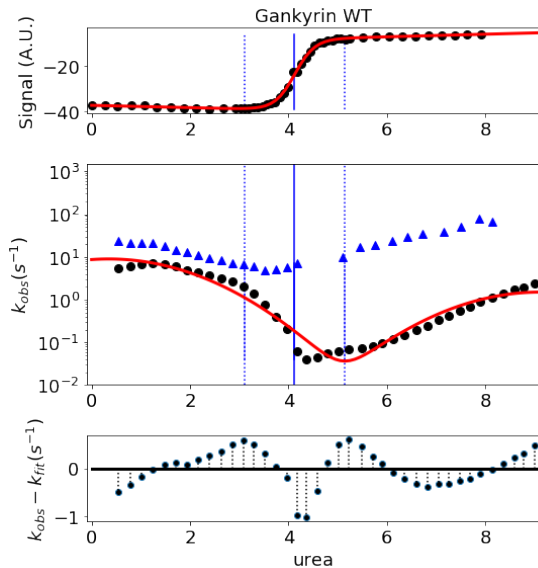
Model: TwoStateChevronMovingTransition

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) ku	0.00000 \pm 0.00000	95% CI[0.00000,	0.00000]
(f) mu	4.98449 \pm 0.25547	95% CI[4.92052,	5.04845]
(f) kf	8.64128 \pm 1.19561	95% CI[8.34192,	8.94064]
(f) mf	0.19051 \pm 0.10469	95% CI[0.16429,	0.21672]
(f) m_prime	-0.27314 \pm 0.01801	95% CI[-0.27764,	-0.26863]

R^2 : 0.94901



Data-set: GankyrinEquilmDenaturationCurve

Equilibrium Model: TwoStateEquilibriumSloping

α_f : -37.25158 ± 0.14271

β_f : -0.49562 ± 0.06439

α_u : 26.33776 ± 0.48279

β_u : 1.06005 ± 0.09104

m : 2.90885 ± 0.05757

$d50$: 4.12424 ± 0.00486

Folding midpoint: 4.12 M

R^2 : 1.00

Kinetic Model: TwoStateChevronMovingTransition

Fit Standard Error: 0.06

k_u : $1.99e-10 \pm 1.80e-10$

μ : $4.98e+00 \pm 2.55e-01$

k_f : $8.64e+00 \pm 1.20e+00$

m_f : $1.91e-01 \pm 1.05e-01$

m_{prime} : $-2.73e-01 \pm 1.80e-02$

R^2 : 0.95

Fitting results

ID: Gankyrin WT

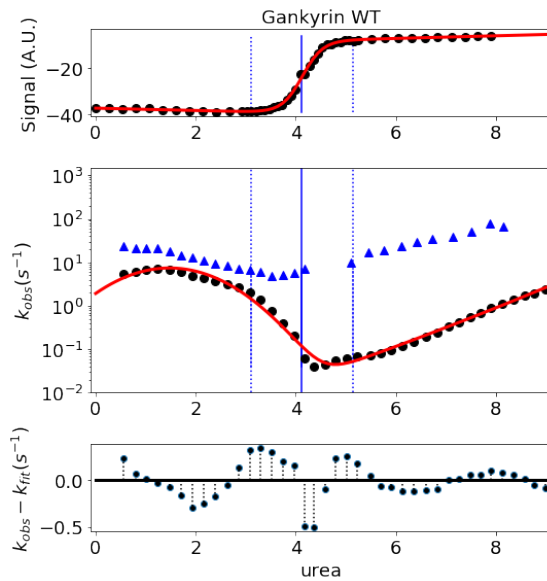
Model: ChevronPolynomialFit

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(f) k_u	0.00062 ± 0.00020	95% CI[$0.00056,$	$0.00067]$
(f) μ	0.76331 ± 0.09633	95% CI[$0.73920,$	$0.78742]$
(f) μ_{prime}	0.01845 ± 0.00691	95% CI[$0.01672,$	$0.02018]$
(f) k_f	1.92293 ± 0.09362	95% CI[$1.89950,$	$1.94636]$
(f) m_f	1.82503 ± 0.04649	95% CI[$1.81339,$	$1.83666]$
(f) m_{prime}	-0.61791 ± 0.00971	95% CI[$-0.62034,$	$-0.61548]$

R^2 : 0.98724



Data-set: GankyrinEquilmDenaturationCurve

Equilibrium Model: TwoStateEquilibriumSloping

$\alpha_f: -37.25158 \pm 0.14271$

$\beta_f: -0.49562 \pm 0.06439$

$\alpha_u: 26.33776 \pm 0.48279$

$\beta_u: 1.06005 \pm 0.09104$

$m: 2.90885 \pm 0.05757$

$d50: 4.12424 \pm 0.00486$

Folding midpoint: 4.12 M

$R^2: 1.00$

Kinetic Model: ChevronPolynomialFit

Fit Standard Error: 0.03

$k_u: 6.15e-04 \pm 2.01e-04$

$\mu: 7.63e-01 \pm 9.63e-02$

$\mu_{prime}: 1.84e-02 \pm 6.91e-03$

$k_f: 1.92e+00 \pm 9.36e-02$

$m_f: 1.83e+00 \pm 4.65e-02$

$m_f_{prime}: -6.18e-01 \pm 9.71e-03$

$R^2: 0.99$

End of this Notebook.

5 SI Notebook 3 - Performing and automating higher-level calculations such as ΔG s & Phi-values

[Author] ERGM

In this notebook we will show how imported and fitted equilibrium and kinetic folding data (as per PyFolding SI Notebooks 1 and 2) can be associated with a particular protein (say WT or a mutant). This enables higher higher-level calculations such as ΔG s & Phi-value analysis to be achieved (also shown here).

If you are less script/computer orientated, you can simply change the data paths/variables, etc for your proteins and re-run the jupyter notebook ("Kernal/Restart & Run all" from the menu above).

5.1 Data Format

Please see PyFolding SI Notebooks 1 and 2 for the format your data has to be in to enable this type of analysis.

```
In [1]: # First off lets load pyfolding & pyplot into this ipython notebook
        #(pyplot allows us to plot more complex figures of our results):
```

```
%matplotlib inline
```

```
# import pyfolding, the pyfolding models and ising models
```

```
import pyfolding
```

```
from pyfolding import *
```

```
# import the package for plotting, call it plt
```

```
import matplotlib.pyplot as plt
```

```
# import numpy as well
```

```
import numpy as np
```

```
<IPython.core.display.Javascript object>
```

PyFolding: Jupyter autoscrolling has been disabled

Now, we need to load some data to analyse. In this notebook I will be using data that I have digitized from the papers below:

Folding of chymotrypsin inhibitor 2. 1. Evidence for a two-state transition.

Jackson, S. E. & Fersht, A. R.

Biochemistry 30, 10428-10435 (1991).

Structure of the hydrophobic core in the transition state for folding of chymotrypsin inhibitor 2: a critical test of the protein engineering method of analysis.

Jackson, S. E., elMasry, N. & Fersht, A. R.

Biochemistry 32, 11270-11278 (1993).

[<http://pubs.acs.org/doi/abs/10.1021/bi00107a010> & <http://pubs.acs.org/doi/abs/10.1021/bi00093a002>]

```
In [2]: # start by loading a data set
        # arguments are "path", "filename"
        # We will do Equilm and Kinetics all together

pth = "../examples/CI2/"

#load CI2 WT equilm and kinetics
EquilmWT_CI2 = pyfolding.read_equilibrium_data(pth,"CI2_WT_Equilm.csv")
KineticsWT_CI2 = pyfolding.read_kinetic_data(pth,"CI2_WT_Kinetics.csv")

# load CI2 Mutant VA38
EquilmVA38_CI2 = pyfolding.read_equilibrium_data(pth,"CI2_VA38_Equilm.csv")
KineticsVA38_CI2 = pyfolding.read_kinetic_data(pth,"CI2_VA38_Kinetics.csv")

# load CI2 Mutant VA66
EquilmVA66_CI2 = pyfolding.read_equilibrium_data(pth,"CI2_VA66_Equilm.csv")
KineticsVA66_CI2 = pyfolding.read_kinetic_data(pth,"CI2_VA66_Kinetics.csv")

# load CI2 Mutant IV48
EquilmIV48_CI2 = pyfolding.read_equilibrium_data(pth,"CI2_IV48_Equilm.csv")
KineticsIV48_CI2 = pyfolding.read_kinetic_data(pth,"CI2_IV48_Kinetics.csv")
```

OK, to perform and automate higher-level calculations such as ΔG s & Phi-value analysis

5.2 Assign Data to a specific protein object

1st we need to assign data to a specific protein by making a “protein object”

```

In [3]: #assign WT
WT = pyfolding.Protein(ID="CI2_WT")      # creating CI2_WT protein object
WT.equilibrium = EquilmWT_CI2             # assigning equilm data to the CI2_WT protein ob
WT.chevron = KineticsWT_CI2               # assigning kinetic data to the CI2_WT protein o

#assign VA38
VA38 = pyfolding.Protein(ID="CI2_VA38")  # creating CI2_WT protein object
VA38.equilibrium = EquilmVA38_CI2         # assigning equilm data to the CI2_WT protein ob
VA38.chevron = KineticsVA38_CI2           # assigning kinetic data to the CI2_WT protein o

#assign VA66
VA66 = pyfolding.Protein(ID="CI2_VA66")  # creating CI2_WT protein object
VA66.equilibrium = EquilmVA66_CI2         # assigning equilm data to the CI2_WT protein ob
VA66.chevron = KineticsVA66_CI2           # assigning kinetic data to the CI2_WT protein o

#assign IV48
IV48 = pyfolding.Protein(ID="CI2_IV48")  # creating CI2_WT protein object
IV48.equilibrium = EquilmIV48_CI2         # assigning equilm data to the CI2_WT protein ob
IV48.chevron = KineticsIV48_CI2           # assigning kinetic data to the CI2_WT protein o

# lets put these in a python list for later on!
Proteins = [WT,
             VA38,
             VA66,
             IV48]

In [4]: # make a python list of the data so we can fit data together and check everything
        #loaded correctly

Equilm_curves = [EquilmWT_CI2,
                  EquilmVA38_CI2,
                  EquilmVA66_CI2,
                  EquilmIV48_CI2]

Kinetic_chevrons = [KineticsWT_CI2,
                    KineticsVA38_CI2,
                    KineticsVA66_CI2,
                    KineticsIV48_CI2]

```

5.3 Fit the Data

Now we need to fit the data. In a similar manner to previous notebooks, we can save time by getting PyFolding to automate the fitting all our data to the model we want:

```
In [5]: # Set temperature to 25.00°C
        # (NOTE: Careful, this sets the temperature for all subsequent calculations)
        pyfolding.set_temperature(25.)

        # fit all the data automatically

        # first lets fit the equilibrium denaturations (we will be using the lists we defined ab
        for c in Equilm_curves:
            c.fit_func = models.TwoStateEquilibrium
            c.fit(p0=[3, 2])
```

```
Set temperature to 25.00°C
(NOTE: Careful, this sets the temperature for all subsequent calculations)
=====
Fitting results
=====
ID: CI2_WT_Equilm
Model: TwoStateEquilibrium
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(f) m      1.86581 ± 0.00075      95% CI[  1.86563,   1.86600]
(f) d50    3.98689 ± 0.00013      95% CI[  3.98686,   3.98693]
-----
R^2: 0.99809
=====
```

```
=====
Fitting results
=====
ID: CI2_VA38_Equilm
Model: TwoStateEquilibrium
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(f) m      2.00158 ± 0.00122      95% CI[  2.00127,   2.00188]
(f) d50    3.74752 ± 0.00019      95% CI[  3.74748,   3.74757]
-----
R^2: 0.99720
=====
```

```
=====
Fitting results
=====
ID: CI2_VA66_Equilm
Model: TwoStateEquilibrium
```

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(f) m	1.74742 ± 0.00193	95% CI[1.74693,	1.74790]
(f) d50	1.45999 ± 0.00042	95% CI[1.45989,	1.46010]

R²: 0.99391
=====

=====

Fitting results

=====

ID: CI2_IV48_Equilm
Model: TwoStateEquilibrium
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(f) m	1.96731 ± 0.00168	95% CI[1.96689,	1.96773]
(f) d50	3.40891 ± 0.00026	95% CI[3.40884,	3.40897]

R²: 0.99645
=====

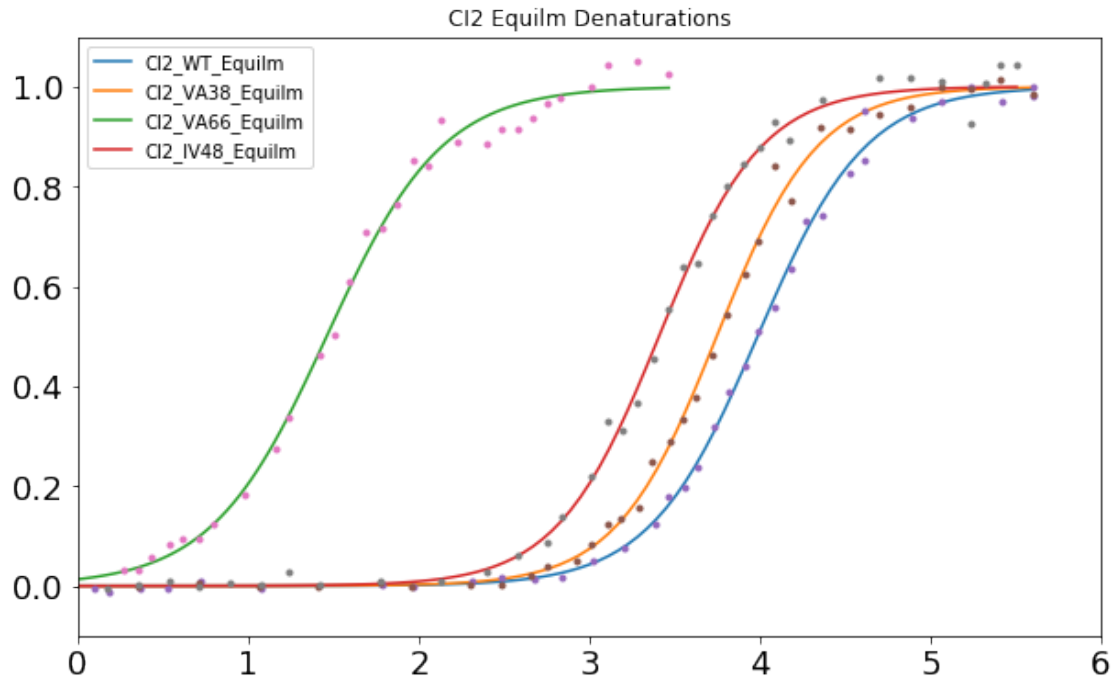
In [6]: *# Before we go further lets check everything looks good!*

the following commands plot all the Equilm curves & fits on one plot

```
plt.figure(figsize=(10,6))
for c in Equilm_curves:
    plt.plot(c.results.x_fit, c.results.y_fit, '-')
for c in Equilm_curves:
    plt.plot(c.x, c.y, '.')
```

This is to make the plot look good!

```
plt.legend([c.ID for c in Equilm_curves], loc='best') # plots a legend & 'loc' command
plt.title("CI2 Equilm Denaturations")                # plots a title.
plt.ylim([-0.1, 1.1])                                # y axis from 0 to 8
plt.xlim([0, 6])                                       # x axis from 0 to 5
plt.show()
```



```
In [14]: # Now lets fit the kinetics
         for c in Kinetic_chevrons:
             c.fit_func = models.TwoStateChevron
             c.fit()
```

=====

Fitting results

=====

ID: CI2_WT_Kinetics
Model: TwoStateChevron
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(f) kf	57.74792 ± 0.08261	95% CI[57.72761,	57.76823]
(f) mf	1.90608 ± 0.00111	95% CI[1.90581,	1.90636]
(f) ku	0.00013 ± 0.00000	95% CI[0.00013,	0.00013]
(f) mu	1.29050 ± 0.00248	95% CI[1.28989,	1.29111]

R²: 0.99939

=====

=====

Fitting results

=====

ID: CI2_VA38_Kinetics
 Model: TwoStateChevron
 Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
 Temperature: 25.00°C

(f) kf	70.14623 ± 0.05693	95% CI[70.13226,	70.16019]
(f) mf	1.81407 ± 0.00055	95% CI[1.81393,	1.81420]
(f) ku	0.00037 ± 0.00000	95% CI[0.00037,	0.00037]
(f) mu	1.37146 ± 0.00121	95% CI[1.37116,	1.37175]

 R^2: 0.99960
 =====

=====

Fitting results

=====

ID: CI2_VA66_Kinetics
 Model: TwoStateChevron
 Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
 Temperature: 25.00°C

(f) kf	9.26632 ± 0.02154	95% CI[9.26103,	9.27160]
(f) mf	2.33134 ± 0.00486	95% CI[2.33015,	2.33253]
(f) ku	0.04530 ± 0.00070	95% CI[0.04512,	0.04547]
(f) mu	1.41589 ± 0.00482	95% CI[1.41471,	1.41707]

 R^2: 0.99325
 =====

=====

Fitting results

=====

ID: CI2_IV48_Kinetics
 Model: TwoStateChevron
 Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
 Temperature: 25.00°C

(f) kf	9.92141 ± 0.03494	95% CI[9.91292,	9.92990]
(f) mf	2.50768 ± 0.00590	95% CI[2.50624,	2.50911]
(f) ku	0.01212 ± 0.00017	95% CI[0.01208,	0.01216]
(f) mu	1.11975 ± 0.00319	95% CI[1.11897,	1.12052]

 R^2: 0.99578
 =====

```
In [8]: # Again, before we go further lets check everything looks good!
```

```
# the following commands plot all the kinetics on one plot
```

```
plt.figure(figsize=(10,6))
```

```
for c in Kinetic_chevrons:
```

```
    plt.plot(c.results.x_fit, c.results.y_fit, '-')
```

```
for c in Kinetic_chevrons:
```

```
    plt.plot(c.x, c.y, '.')
```

```
# as before this is to make the plot look good!
```

```
plt.legend([c.ID for c in Kinetic_chevrons], loc='best')
```

```
# plots a legend & 'loc' comma
```

```
plt.title("CI2")
```

```
# plots a title.
```

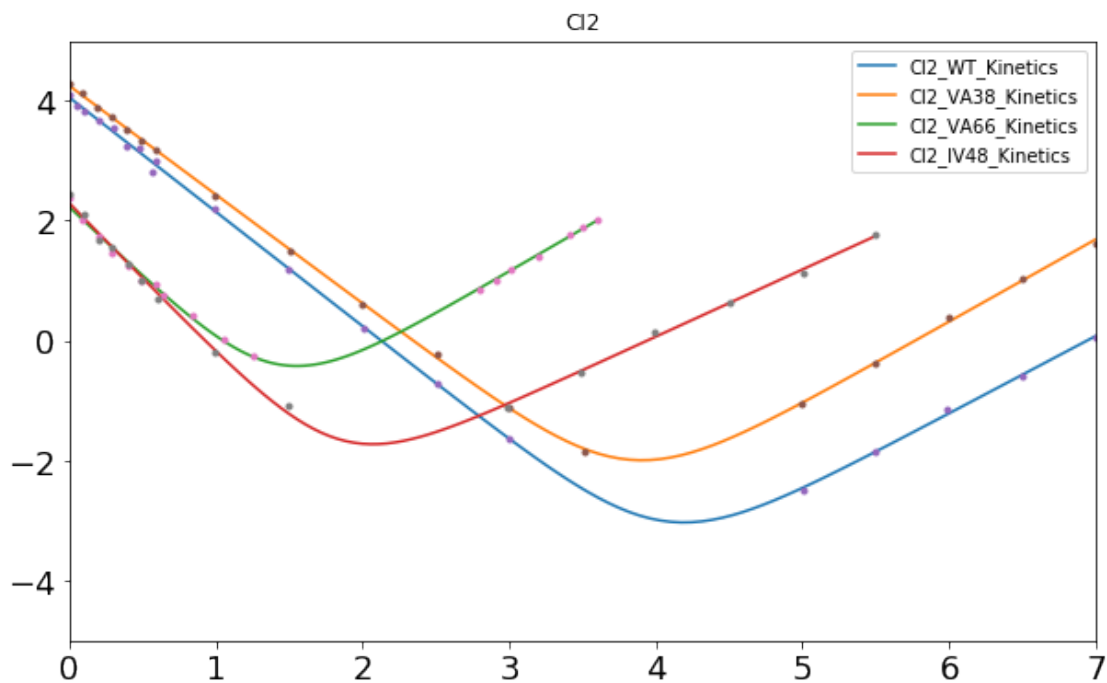
```
plt.ylim([-5, 5])
```

```
# y axis from 0 to 8
```

```
plt.xlim([0, 7])
```

```
# x axis from 0 to 5
```

```
plt.show()
```



Now lets perform and automate some higher-level calculations such as Delta Gs & Phi-value analysis :

5.4 Calculating Free Energy of unfolding

```
In [9]: # Lets start with the free energy of unfolding ( $\Delta G_D-N$ ) [calculated from the equilibrium m-values]
# we can get an individual values for each protein by calling up our protein objects we
# Lets try it for WT
```

```
print WT.deltaG
```

```
7.43880166046
```

```
In [10]: # Now we can automate with the list we created at the start of this notebook {in cell (1)}
# from the list in cell (3) the order should be: WT, VA38, VA66 & IV48
```

```
for c in Proteins:
    print "{0:s} --> delta G: {1:.2f}".format(c.ID, c.deltaG)
```

```
CI2_WT --> delta G: 7.44
CI2_VA38 --> delta G: 7.50
CI2_VA66 --> delta G: 2.55
CI2_IV48 --> delta G: 6.71
```

Protein	delta G
WT	7.53
VA38	7.54
VA66	2.57
IV48	6.80

The values from the paper are:

```
In [11]: # We can also do calculations on any of the data we have.
# Lets calculate some DDG values for VA66

# Stability change in water between VA66 and WT (i.e. DDG from Equilibrium curves)
ddg = WT.deltaG - VA66.deltaG

# Free Energy change between Denatured and Transition State between VA66 and WT
#(i.e. DDG from folding rates in water)

ddts = constants.RT * np.log(WT.kf_H2O / VA66.kf_H2O)
```



```
print ddg, ddts
```

```
4.88758792146 0.354099720978
```

5.5 Calculating Phi Values

```
In [12]: # Lets finish by calculating some Phi Values
        # we can get an individual value by calling up our protein objects we defined
        # at the start of this notebook {in cell (3)}
        # Lets try it for the mutant VA38
```

```
from pyfolding.phi import phi
```

```
PhiValue = phi(WT, VA66)
```

```
print PhiValue
```

```
0.0724487675042
```

```
In [13]: # Now we can automate with the list we created also at the start of this notebook {in c
        # from the list in cell (3) the order should be a Phi value for VA38, IV48 & IA48
        mut_proteins = Proteins[1:]
```

```
for c in mut_proteins:
    c_phi_value = phi(WT, c)
    print "{0:s} --> Phi value: {1:.2f}".format(c.ID, c_phi_value)
```

```
CI2_VA38 --> Phi value: 0.45
```

```
CI2_VA66 --> Phi value: 0.07
```

```
CI2_IV48 --> Phi value: 0.46
```

5.5.1 As you can see this works. YAY!

However, the final Phi values are different from the original CI2 paper due to errors from digitizing the data. CI2_VA38 -> Phi value: 0.45 CI2_VA66 -> Phi value: 0.23 CI2_IV48 -> Phi value: 0.29

For example, VA38 - has a very small difference in stability from WT. Thus when digitising this causes a large error in the reported Phi values.

End of this Notebook.

6 SI Notebook 4 - Fitting multiple datasets with linked equations (share certain parameters between the models)

[Authors] ERGM, ARL

In this notebook, let's try to fit a complicated model to some example data. Here the idea is to fit two models to two datasets, but share some of the parameters between the two models. In real terms this is difficult to do in most bought fitting software, but we have the full power of a programming language at our disposal.

We can use the GlobalFit object to do exactly what we want. Let's start by getting some nice data digitised from this paper:

Structural insights into an equilibrium folding intermediate of an archaeal ankyrin repeat protein. Low C, Weininger U, Neumann P, Klepsch M, Lilie H, Stubbs MT, and Balbach J
PNAS (2008) 105: 3779-3784

[<http://www.pnas.org/cgi/doi/10.1073/pnas.0710657105>]

Remember, if you are less script/computer orientated, you can simply change the data paths/variables, etc for your proteins and re-run the jupyter notebook ("Kernal/Restart & Run all" from the menu above).

6.1 Data Format

Please see PyFolding SI Notebooks 1 and 2 for the format your data has to be in to enable this type of analysis.

```
In [1]: # First off let's load pyfolding & pyplot into this ipython notebook
        # (pyplot allows us to plot more complex figures of our results):
```

```
%matplotlib inline
import pyfolding
from pyfolding import models

# let's use some other libraries also
import matplotlib.pyplot as plt
import numpy as np
```

```
<IPython.core.display.Javascript object>
```

PyFolding: Jupyter autoscrolling has been disabled

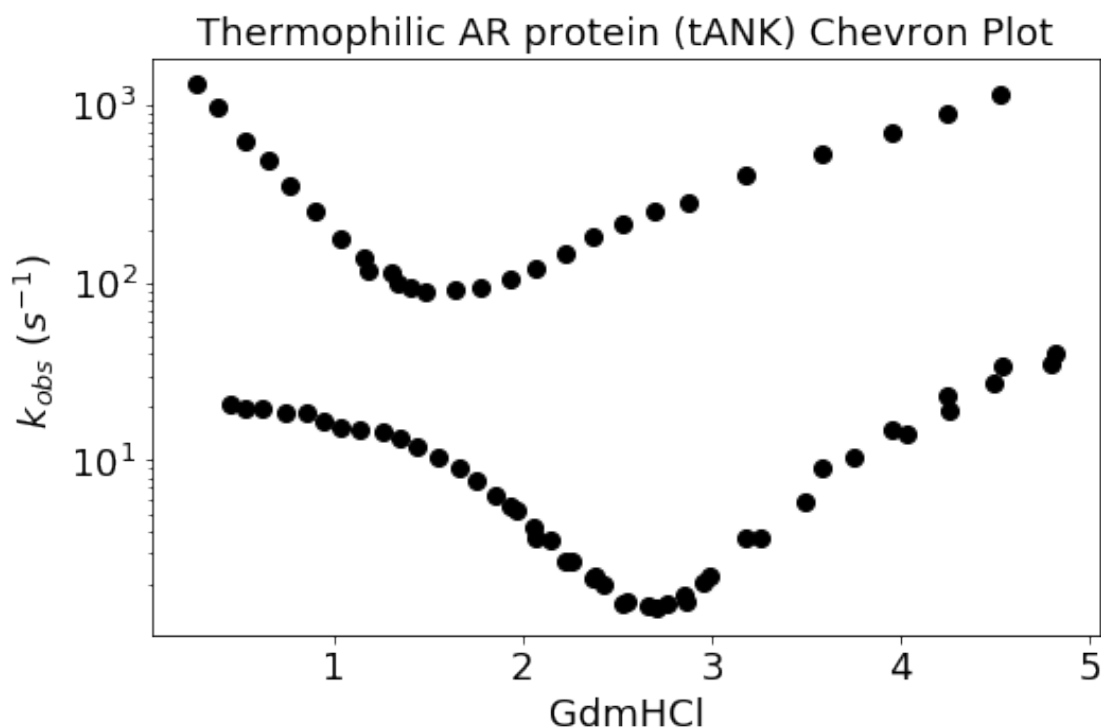
6.2 Load and Plot the Data

```
In [2]: # loading the data - The kinetics of each protein is in one .csv as per
        #PyFolding SI Notebooks 1 and 2
```

```
pth = "../examples/tANK/"
tANK_all = pyfolding.read_kinetic_data(pth, "ChevronAll.csv")
```

```
In [3]: # let's give this dataset a good name
        tANK_all.ID = 'Thermophilic AR protein (tANK)'
```

```
In [4]: # easy plotting of the entire dataset ...
        tANK_all.plot()
```



6.3 Custom Plotting

```
In [5]: # ...or, custom plotting
```

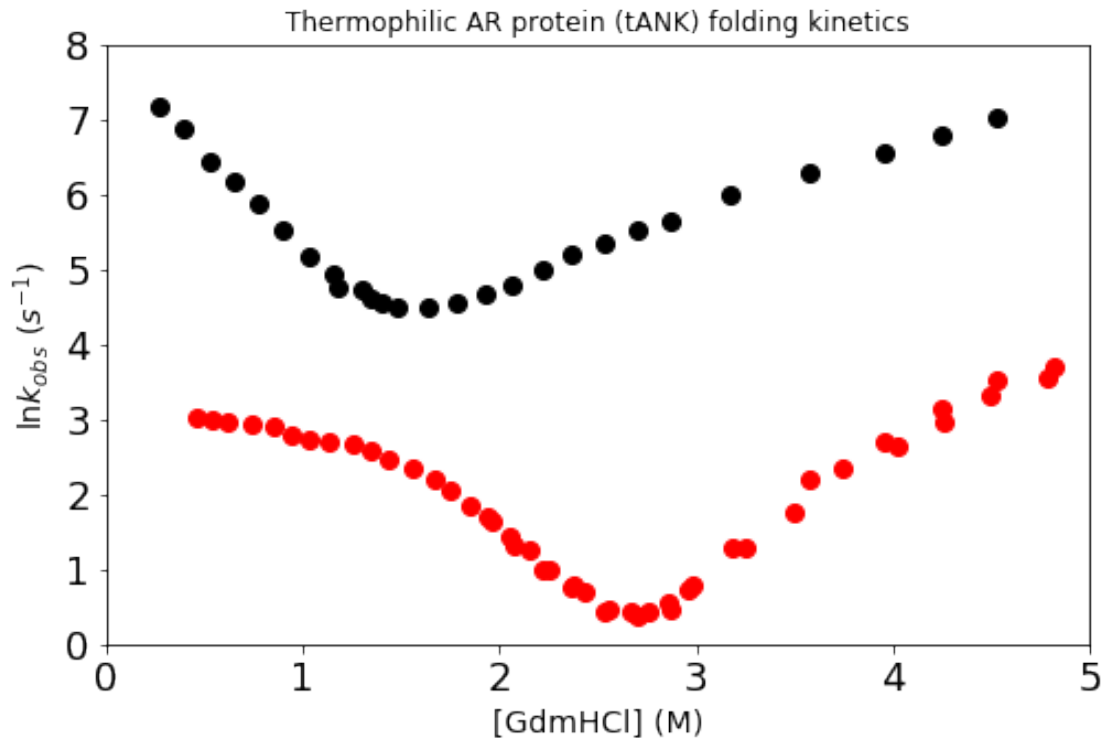
```
k1_x, k1_y = tANK_all.chevron('k1') # defines the slower chevron rates
k2_x, k2_y = tANK_all.chevron('k2') # defines the faster chevron rates

plt.figure(figsize=(8,5)) # makes the figure 8cm by 5cm
plt.plot(k1_x, k1_y, 'ro', markersize=8) # red filled circles, sized '8'
plt.plot(k2_x, k2_y, 'ko', markersize=8) # black filled circles, sized '8'
```

```

plt.rc('xtick', labelsizes=14)           # fontsize of the x tick labels
plt.rc('ytick', labelsizes=14)           # fontsize of the y tick labels
plt.ylim([0, 8])                         # y axis from 0 to 8
plt.xlim([0, 5])                         # x axis from 0 to 5
plt.grid(False)                          # no grid on the graph
plt.xlabel('[GdmHCl] (M)', fontsize=14)  # x axis title with fontsize
plt.ylabel(r'$\ln k_{obs}$ (s$^{-1}$)', fontsize=14) # y axis title with fontsize
plt.title('Thermophilic AR protein (tANK) folding kinetics') # Plot title
plt.show()

```



6.4 Set temperature

Note that the measurements were performed at 15°C, so we need to adjust the temperature:

```
In [6]: pyfolding.set_temperature(15.0)
```

Set temperature to 15.00°C

(NOTE: Careful, this sets the temperature for all subsequent calculations)

6.5 List the Models used

In the paper they decided that their data showed that their protein was folding via a three state mechanism. So lets try fitting the digitized data to this model - three state chevron with fast phase. To do this, we need to share the parameters like this:

1. Fit a “two-state chevron” model to the fast phase, and share those params with:
2. A “three-state with fast phase” model for the slower phase

First of all lets print the information on each model:

```
In [7]: models.TwoStateChevron().info()
```

$$k_{obs} = k_f + k_u$$

where:

$$\begin{aligned} k_f &= k_f^{H_2O} \exp(-m_{kf}x) \\ k_u &= k_u^{H_2O} \exp(m_{ku}x) \end{aligned} \tag{2}$$

thus:

$$k_{obs} = k_f^{H_2O} \exp(-m_{kf}x) + k_u^{H_2O} \exp(m_{ku}x)$$

Two state chevron plot.

Folding Scheme:
N <-> D

Params:

k obs = rate constant of unfolding or refolding at a particular denaturant conce
kf = rate constant of refolding at a particular denaturant concentration
mf = the gradient of refolding arm of the chevron
ku = rate constant of unfolding at a a particular denaturant concentration
mu = the gradient of unfolding arm of the chevron
x = denaturant concentration (M)

Reference:

Jackson SE and Fersht AR. Folding of chymotrypsin inhibitor 2.
1. Evidence for a two-state transition.
Biochemistry (1991) 30(43):10428-10435.

```
In [8]: models.ThreeStateFastPhaseChevron().info()
```

$$k_{obs} = \frac{k_{fi} + k_{if}}{(1 + 1/K_{iu})}$$

where:

$$\begin{aligned} k_{fi} &= k_{fi}^{H_2O} \exp(m_{fi}x) \\ k_{if} &= k_{if}^{H_2O} \exp(-m_{if}x) \\ k_{iu} &= k_{iu}^{H_2O} \exp(m_{iu}x) \\ k_{ui} &= k_{ui}^{H_2O} \exp(-m_{ui}x) \\ K_{iu} &= \frac{k_{iu}}{k_{iu} + k_{ui}} \end{aligned} \quad (3)$$

Three state chevron with single intermediate.

Folding Scheme: N <-> I <-> D

Params:

k obs = rate constant of unfolding or refolding at a particular denaturant conce
kfi = microscopic rate constant for the conversion of folded to intermediate
kif = microscopic rate constant for the conversion of intermediate to folded
kiu = microscopic rate constant for the conversion of intermediate to unfolded
kui = microscopic rate constant for the conversion of unfolded to intermediate
Kiu = equilibrium constant for the rapid equilibration between intermediate & un
mfi = m-value associated with the kinetic transition between folded & intermedia
mif = m-value associated with the kinetic transition between intermediate & fold
miu = m-value associated with the kinetic transition between intermediate & unfo
mui = m-value associated with the kinetic transition between unfolded & intermed
x = denaturant concentration (M)

Reference:

Parker et al. An integrated kinetic analysis of
intermediates and transition states in protein folding reactions.
Journal of molecular biology (1995) vol. 253 (5) pp. 771-86

6.6 Create a New model - called FastPhase

To fit the data to two equations at once where certain parameters are shared, we need to create a fast phase model, which is essentially a two state chevron, but where the parameters represent the folding and unfolding of the intermediate. We can do this *on-the-fly* by using the template model to create a new FastPhase** model:**

```
In [9]: class FastPhase(pyfolding.FitModel):
        """ Fast phase model
        """
        def __init__(self):
            pyfolding.FitModel.__init__(self)
            fit_args = self.fit_func_args
            self.params = tuple( [(fit_args[i],i) for i in xrange(len(fit_args))] )
            self.default_params = np.array([100., 1.3480, 5e-4, 1.])
            self.verified = True

        def fit_func(self, x, kui, mui, kiu, miu):
            k_obs = kui*np.exp(-mui*x) + kiu*np.exp(miu*x)
            return k_obs

        def error_func(self, y):
            return np.log(y)
```

6.7 Setting up a custom Global fit

```
In [10]: # We first need to differentiate the data into the two phases we will fit
k1_x, k1_y = tANK_all.chevron('k1') # defines the slower chevron rates
k2_x, k2_y = tANK_all.chevron('k2') # defines the faster chevron rates
```

```
# start by setting up a GlobalFit object
global_fit = pyfolding.GlobalFit()
global_fit.fit_funcs = (models.ThreeStateFastPhaseChevron, FastPhase)
global_fit.x = (k1_x, k2_x) # remember, k1 defines the slower chevron
global_fit.y = (k1_y, k2_y) # & k2 defines are faster chevron
global_fit.ID = ['slow_phase', 'fast_phase']
global_fit.shared = ['kui', 'mui', 'kiu', 'miu']
global_fit.initialise()
```

```
In [11]: # find out the order of params for p0
print global_fit.params.keys()
```

```
['mui', 'miu', 'kui', 'kiu', 'kif_{slow_phase}', 'mif_{slow_phase}', 'kfi_{slow_phase}', 'mfi_{s
```

```
In [12]: # set some reasonable starting parameters for the fit
p0 = np.array([1.7, 0.01, 1860., 2.7, 2652.0, 2.8, 14., 1.])
bounds = ((-5., -5., 0., 0., 0., -5., 0., -5.), (5., 5., 1e4, 1e4, 1e4, 5., 1e4, 5.))

# run the fit and get the results
_,_ = global_fit.fit(p0=p0, bounds=bounds)
results = global_fit.results
```

6.8 Display Results from custom Global fit

Display the results from the fitting. Note that in the results table, (f) means a free-variable, (s) is shared and (c) is constant.

```
In [13]: for r in results:
          r.display()
```

```
=====
Fitting results
=====
```

```
ID: slow_phase
```

```
Model: ThreeStateFastPhaseChevron
```

```
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
```

```
Temperature: 15.00°C
```

(s) kui	2652.18218 ± 54.43026	95% CI[2638.45729, 2665.90706]
(s) mui	2.82889 ± 0.02696	95% CI[2.82210, 2.83569]
(s) kiu	14.30673 ± 0.37100	95% CI[14.21318, 14.40028]
(s) miu	0.99906 ± 0.00828	95% CI[0.99698, 1.00115]
(f) kif	1860.56417 ± 63.30515	95% CI[1844.60144, 1876.52691]
(f) mif	2.73831 ± 0.01748	95% CI[2.73391, 2.74272]
(f) kfi	0.01057 ± 0.00038	95% CI[0.01047, 0.01066]
(f) mfi	1.76794 ± 0.00936	95% CI[1.76558, 1.77030]

```
-----
R^2: 0.97224
=====
```

```
=====
Fitting results
=====
```

```
ID: fast_phase
```

```
Model: FastPhase
```

```
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
```

```
Temperature: 15.00°C
```

(s) kui	2652.18218 ± 52.85223	95% CI[2638.84949, 2665.51487]
(s) mui	2.82889 ± 0.02618	95% CI[2.82229, 2.83550]
(s) kiu	14.30673 ± 0.36024	95% CI[14.21585, 14.39760]
(s) miu	0.99906 ± 0.00804	95% CI[0.99704, 1.00109]

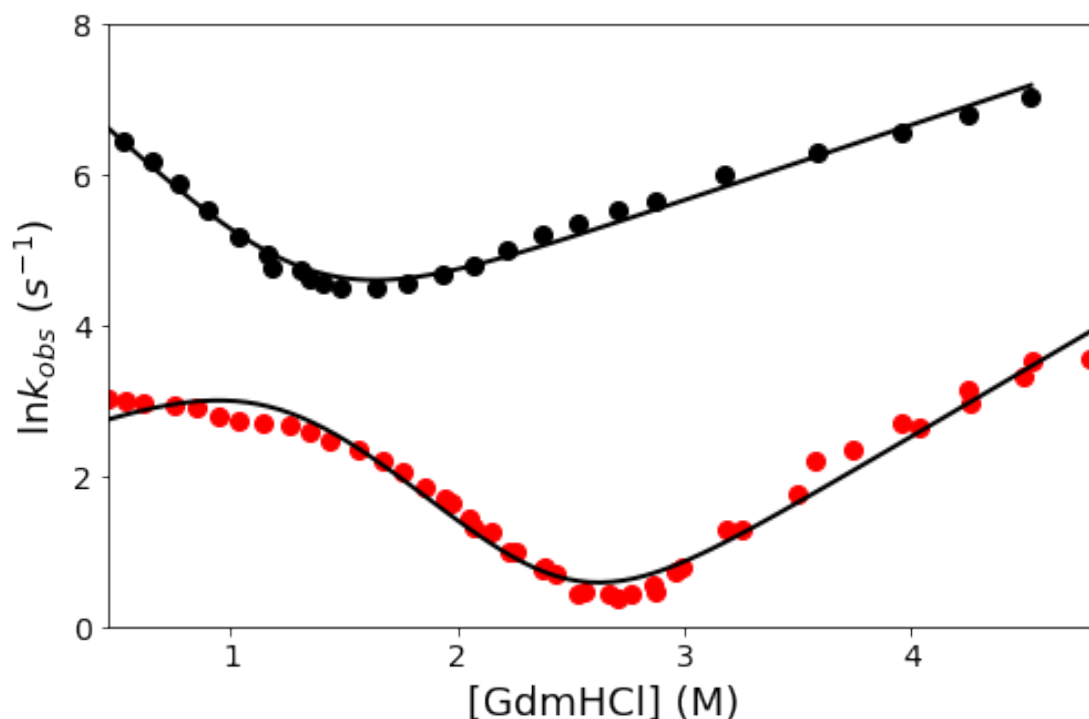
```
-----
R^2: 0.98521
=====
```


6.9 Plot the Figure of the custom Global fit

```
In [14]: # now that we've fitted the data, plot a figure
plt.figure(figsize=(8,5))
plt.plot(k1_x, k1_y, 'ro', markersize=8)
plt.plot(k2_x, k2_y, 'ko', markersize=8)

# plot each of the chevrons...
for r in results:
    plt.plot(r.x_fit, r.y_fit, 'k-', linewidth=2)

plt.xlim([min(k1_x), max(k1_x)])
plt.rc('xtick', labels=18) # fontsize of the x tick labels
plt.rc('ytick', labels=18) # fontsize of the y tick labels
plt.grid(False)
plt.xlabel('[GdmHCl] (M)', fontsize=18)
plt.ylabel(r'$\ln k_{obs}$ (s$^{-1}$)', fontsize=18)
plt.ylim([0, 8])
plt.show()
```



6.10 Plot custom Global fit with "prettier" graphics

We can also output this with the “prettier” graphics showing chevron & equilibrium curves together (as in PyFolding SI Notebook 1)

6.10.1 1st - we need to add the kinetic results to the protein object

```
In [15]: tANK_all.results = results[0]
         tANK_all.components = {'fast': np.exp(results[1].y_fit)}
         tANK_all.fit_func = models.ThreeStateFastPhaseChevron
```

Warning: overwriting fit result for <pyfolding.core.Chevron object at 0x114753a50>

6.10.2 2nd - lets get the equilibrium data & fit to a three state model

```
In [16]: #1st load the data by adding the equilibrium
         tank_Equilm = pyfolding.read_equilibrium_data(pth, "tANK_equilmFluorscence.csv")

         #then select the fit function and associate it with the data
         tank_Equilm.fit_func = models.ThreeStateEquilibrium

         # Fit it with parameters (as the defaults ones are not close enough)
         tank_Equilm.fit(p0=[350,2,25,18.5,11.6,34,12.6])

         # Plot the figure to check
         tank_Equilm.plot()
```

=====

Fitting results

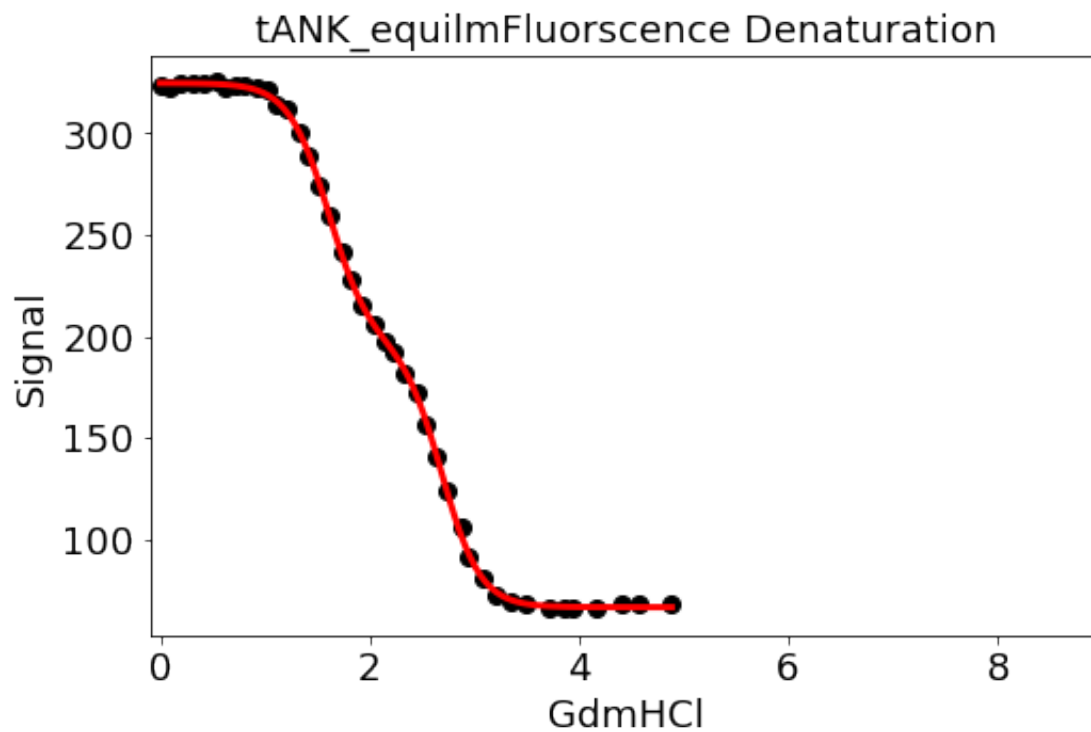
=====

ID: tANK_equilmFluorscence
Model: ThreeStateEquilibrium
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 15.00°C

(f) Y_N	324.75027 ± 0.74190	95% CI[324.56451,	324.93603]
(f) Y_I	197.11446 ± 3.56063	95% CI[196.22293,	198.00599]
(f) Y_D	66.56156 ± 0.75910	95% CI[66.37149,	66.75162]
(f) DGni	4.77711 ± 0.21538	95% CI[4.72318,	4.83104]
(f) m_ni	2.96838 ± 0.14905	95% CI[2.93106,	3.00570]
(f) DGid	8.33336 ± 0.44659	95% CI[8.22154,	8.44518]

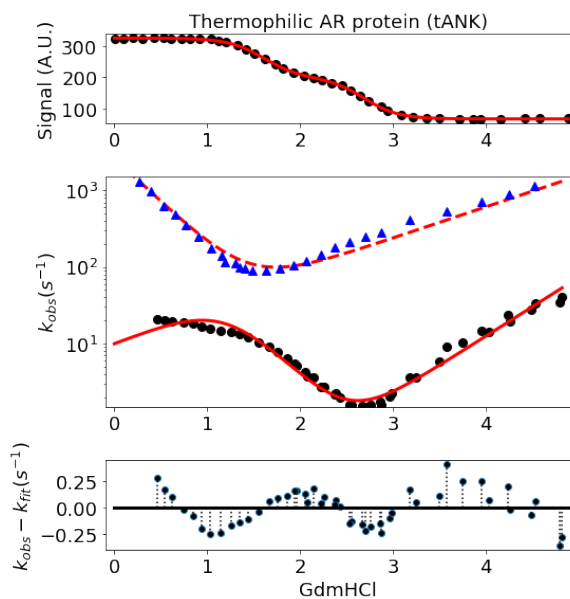
(f) m_{id} 3.09530 ± 0.15594 95% CI[3.05626, 3.13435]

R²: 0.99983
=====



6.10.3 3rd - lets plot both together & print out to pdf

```
In [17]: pyfolding.plot_figure(tank_Equilm, tANK_all, display=True)
```



Data-set: tANK_equilmFluorscence

Equilibrium Model: ThreeStateEquilibrium

Y_N : 324.75027 \pm 0.74190

Y_I : 197.11446 \pm 3.56063

Y_D : 66.56156 \pm 0.75910

DG_{Ni}: 4.77711 \pm 0.21538

m_{Ni} : 2.96838 \pm 0.14905

DG_{Id}: 8.33336 \pm 0.44659

m_{Id} : 3.09530 \pm 0.15594

R^2 : 1.00

Kinetic Model: ThreeStateFastPhaseChevron

Fit Standard Error: 0.02

k_{ui} : 2.65e+03 \pm 5.44e+01

m_{ui} : 2.83e+00 \pm 2.70e-02

k_{iu} : 1.43e+01 \pm 3.71e-01

m_{iu} : 9.99e-01 \pm 8.28e-03

k_{if} : 1.86e+03 \pm 6.33e+01

m_{if} : 2.74e+00 \pm 1.75e-02

k_{fi} : 1.06e-02 \pm 3.82e-04

m_{fi} : 1.77e+00 \pm 9.36e-03

R^2 : 0.97

End of this Notebook.

7 SI Notebook 5 - Fitting to a HomoZipper/HomoPolymer Ising Model

[Author] ERGM

In this notebook we will show how equilibrium folding data can be imported into a notebook and fitted to Homopolymer Ising Model.

If you are less script/computer orientated, you can simply change the data paths and other parameters for your proteins and re-run the jupyter notebook ("Kernal/Restart & Run all" from the menu above).

7.1 Data Format

Please see PyFolding SI Notebooks 1 and 2 for the format your data has to be in to enable this type of analysis. Remember for Ising Model Analysis here each protein dataset (equilibrium denaturation curve) must have its own .csv

First off lets load pyfolding & pyplot into this ipython notebook (pyplot allows us to plot more complex figures of our results):

```
In [1]: # use this command to tell Jupyter to plot figures inline with the text
        %matplotlib inline

        # import pyfolding, the pyfolding models and ising models
        import pyfolding
        from pyfolding import *

        # import the package for plotting, call it plt
        import matplotlib.pyplot as plt

        # import numpy as well
        import numpy as np
```

<IPython.core.display.Javascript object>

PyFolding: Jupyter autoscrolling has been disabled

Now, we need to load some data to analyse. I will import the equilibrium denaturation of a series of CTPRa proteins from:

Phillips, J. J., Javadi, Y., Millership, C. & Main, E. R. Modulation of the multistate folding of designed TPR proteins through intrinsic and extrinsic factors. Protein Sci 21, 327-338 (2012).

I will load 7 denaturation curves that correspond to the following proteins:

Protein	Filename	No. of Helices (repeating unit)
CTPRa2	H2_25C.csv	5
CTPRa3	H3_25C.csv	7
CTPRa4	H4_25C.csv	9
CTPRa5	H5_25C.csv	11
CTPRa6	H6_25C.csv	13
CTPRa8	H8_25C.csv	17
CTPRa10	H10_25C.csv	21

7.2 Import Data, assign names and put into a list

We will load all of the data together, as follows:

```
In [2]: # arguments are "path", "filename"
pth = "../examples/CTPRan"

#this is a set of commands to automate loading the data for each proteins
fn = ["H2_25C.csv", "H3_25C.csv", "H4_25C.csv",
      "H5_25C.csv", "H6_25C.csv", "H8_25C.csv", "H10_25C.csv"]

#Here we are loading all the curves in a list called "proteins" and assigning them names
proteins = [pyfolding.read_equilibrium_data(pth,f) for f in fn]

# also store the number of helices for each protein
n_helices = [5, 7, 9, 11, 13, 17, 21]
```

7.3 Set the Temperature

```
In [3]: # Set temperature to 25.00°C
# (NOTE: Careful, this sets the temperature for all subsequent calculations)
pyfolding.set_temperature(25.)
```

Set temperature to 25.00°C

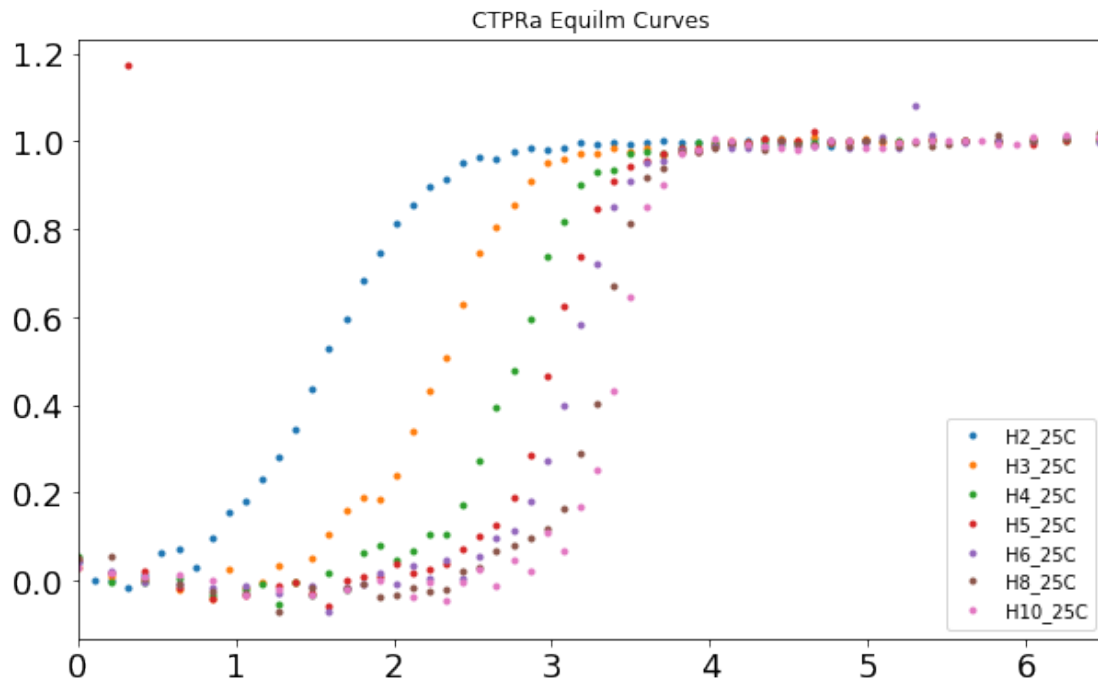
(NOTE: Careful, this sets the temperature for all subsequent calculations)

7.4 Plot the data

In [4]: *# the following commands plot all the data curves on one plot*

```
plt.figure(figsize=(10,6))
for c in proteins:
    plt.plot(c.x, c.y, '.')

# the following commands plot all the data curves on one plot, where "loc" command deter
plt.legend([c.ID for c in proteins], loc='best')
plt.title("CTPRa Equilm Curves")
plt.xlim([0, 6.5]) # x axis from 0 to 6.5
plt.show()
```



7.5 Print information on the HomoZipper Ising Model

In [5]: *# printing the equation for viewing*

```
models.HomozipperIsingEquilibrium().info()
```

the partition function (q) and thus fraction of folded protein (f) of n arrayed repeats are given by:

$$q = 1 + \frac{\kappa([\kappa\tau]^{n+1} - [n+1]\kappa\tau - n)}{(\kappa\tau + 1)^2}$$

$$f = \frac{1}{n} \sum_{i=0}^n i \frac{(n-i+1)\kappa^i \tau^{i-1}}{q}$$
(4)

where:

$$\kappa(x) = \exp \frac{-G_i}{RT} = \exp \frac{-G_{i,H_2O} + m_i x}{RT}$$

$$\tau(x) = \exp \frac{-G_{i,i+1}}{RT}$$

Homopolymer Zipper Ising model

Params:

q = partition function
f = fraction of folded protein
Kappa = equilibrium constant of folding for a given repeating unit
Tau = equilibrium constant of association between 2 repeating units
n = number of repeating units
x = denaturant concentration (M)
Gi = intrinsic stability (folding energy) of a repeating unit i
mi = denaturant sensitivity of the intrinsic stability of a repeating unit i
Gi,i+1 = interface interaction energy between 2 repeating units
R = Universal Gas Constant (kcal.mol⁻¹.K⁻¹)
T = Temperature (Kelvin)

Reference:

Aksel and Barrick. Analysis of repeat-protein folding using nearest-neighbor statistical mechanical models.
Methods in enzymology (2009) vol. 455 pp. 95-125

7.6 Fitting to a Homopolymer Ising model

We need to define a topology for each protein, this is essentially the domains of the protein in order (N->C)

Module	Abbreviation	Defined in Ising as
Module, type 1	r	RepeatDomain

Module	Abbreviation	Defined in Ising as
Module, type 2	Mr	MutantRepeatDomain
Module, type 3	h	HelixDomain
Module, type 4	l	LoopDomain
Module, type 5	MI	MutantLoopDomain
Cap Module	c	CapDomain
Mut. Cap Module	Mc	MutantCapDomain
Decoupled Cap Domain	DC	DecoupleCapDomain

Protein topology can be defined via the following Ising motifs For Homopolymer fitting Helix, Repeat & loop motif functions are identical. We will need the differing types when using the Heteropolymer in the next Notebook. The nomenclature stems from our CTPR proteins analysis, as their repeated motifs can be defined in terms of either repeated TPRs, repeated helices and/or loops.

Each motif is defined as having:

1. an intrinsic stability term (G_i)
2. a denaturation dependence term (m_i) - which is associated with G_i
3. an interface energy term (G_{ij}) - defined as the interaction between folded units i and $i-1$, i.e. the interface of the i unit with the N-terminal unit $i-1$ to it.
4. The Ising model automatically takes into account the first N-terminal repeat does not have an interface term $\{ij\}$ associated with it.

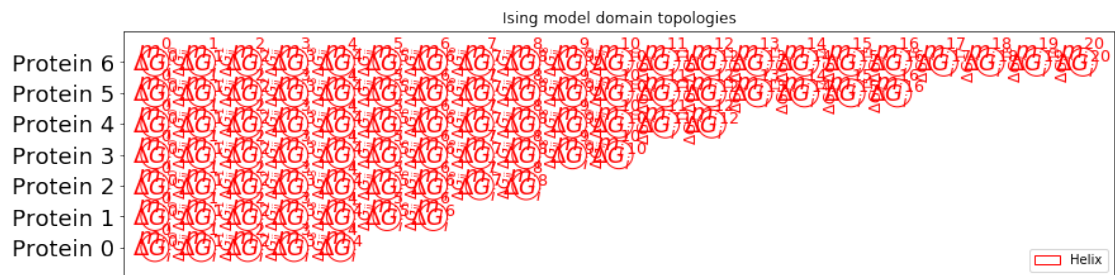
7.7 Set up Our Topology

```
In [6]: # set up the protein topologies automatically
        h = ising.HelixDomain

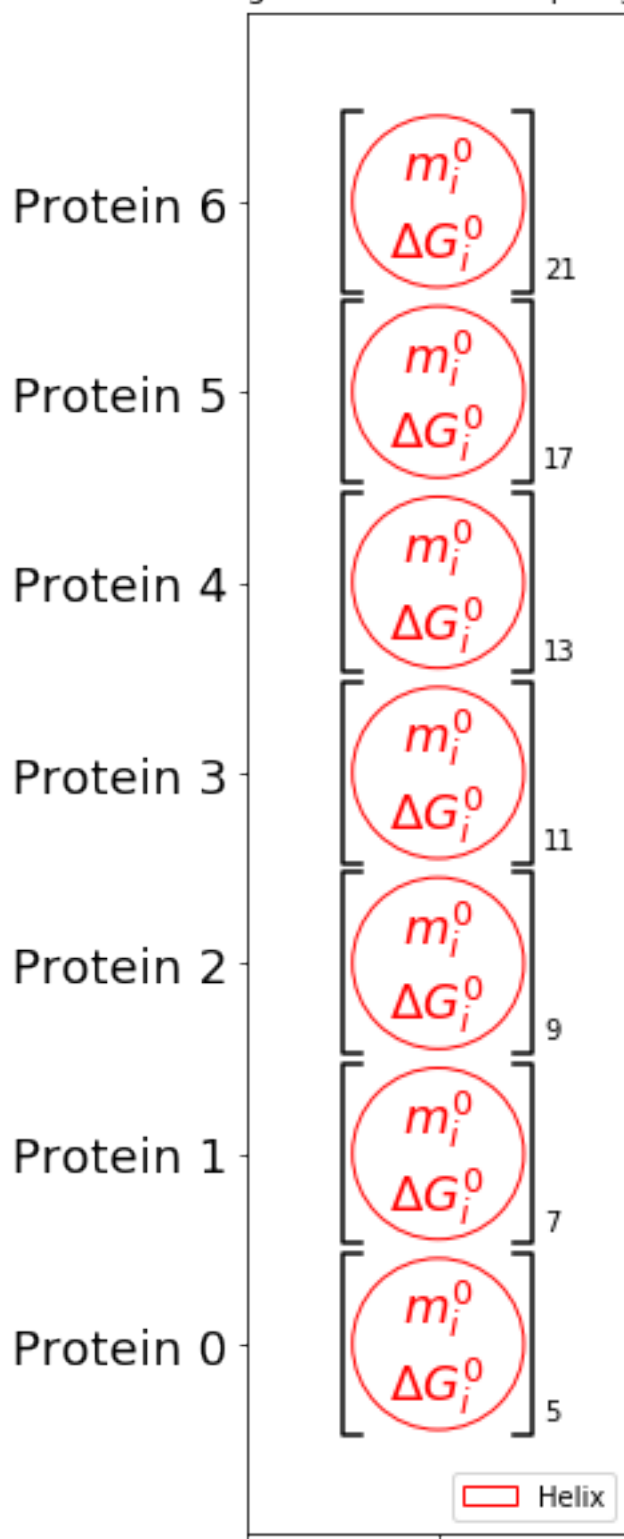
        # We have already defined the number of helices for each protein
        # in cell (3) above as "n_helices", so:
        topology = [[h]*n for n in n_helices]

        # plot them either in an expanded form
        ising.plot_domains(topology, collapse=False)

        # or in a collapsed form
        ising.plot_domains(topology, collapse=True)
```



Ising model domain topologies



7.8 Automatic global fitting to the homozipper model

```
In [7]: # use a one-liner and save out the curve fits via: save='/Users/ergm/Desktop/test.csv'
        r_homo = ising.fit_homopolymer(proteins, n_helices, save='/Users/ergm/Desktop/test.csv')
```

```
=====
Fitting results
=====
```

ID: H2_25C

Model: HomozipperIsingEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(c) n	5.00000		
(s) DG_intrinsic	3.52517 ± 0.00796	95% CI[3.52315, 3.52719]
(s) m_intrinsic	-0.45188 ± 0.00090	95% CI[-0.45211, -0.45166]
(s) DG_interface	-5.29438 ± 0.01119	95% CI[-5.29721, -5.29154]

R²: 0.99659
=====

```
=====
Fitting results
=====
```

ID: H3_25C

Model: HomozipperIsingEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(c) n	7.00000		
(s) DG_intrinsic	3.52517 ± 0.00796	95% CI[3.52315, 3.52719]
(s) m_intrinsic	-0.45188 ± 0.00090	95% CI[-0.45211, -0.45166]
(s) DG_interface	-5.29438 ± 0.01119	95% CI[-5.29721, -5.29154]

R²: 0.99637
=====

```
=====
Fitting results
=====
```

ID: H4_25C

Model: HomozipperIsingEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
 Temperature: 25.00°C

(c) n	9.00000		
(s) DG_intrinsic	3.52517 ± 0.00796	95% CI[3.52315, 3.52719]
(s) m_intrinsic	-0.45188 ± 0.00090	95% CI[-0.45211, -0.45166]
(s) DG_interface	-5.29438 ± 0.01119	95% CI[-5.29721, -5.29154]

 R^2: 0.99366
 =====

=====

Fitting results

=====

ID: H5_25C
 Model: HomozipperIsingEquilibrium
 Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
 Temperature: 25.00°C

(c) n	11.00000		
(s) DG_intrinsic	3.52517 ± 0.00796	95% CI[3.52315, 3.52719]
(s) m_intrinsic	-0.45188 ± 0.00090	95% CI[-0.45211, -0.45166]
(s) DG_interface	-5.29438 ± 0.01119	95% CI[-5.29721, -5.29154]

 R^2: 0.87746
 =====

=====

Fitting results

=====

ID: H6_25C
 Model: HomozipperIsingEquilibrium
 Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
 Temperature: 25.00°C

(c) n	13.00000		
(s) DG_intrinsic	3.52517 ± 0.00796	95% CI[3.52315, 3.52719]
(s) m_intrinsic	-0.45188 ± 0.00090	95% CI[-0.45211, -0.45166]
(s) DG_interface	-5.29438 ± 0.01119	95% CI[-5.29721, -5.29154]

 R^2: 0.99559
 =====

=====

Fitting results

```

=====
ID: H8_25C
Model: HomozipperIsingEquilibrium
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(c) n          17.00000
(s) DG_intrinsic  3.52517 ± 0.00796      95% CI[  3.52315,   3.52719]
(s) m_intrinsic  -0.45188 ± 0.00090      95% CI[ -0.45211,  -0.45166]
(s) DG_interface -5.29438 ± 0.01119      95% CI[ -5.29721,  -5.29154]
-----
R^2: 0.99526
=====

```

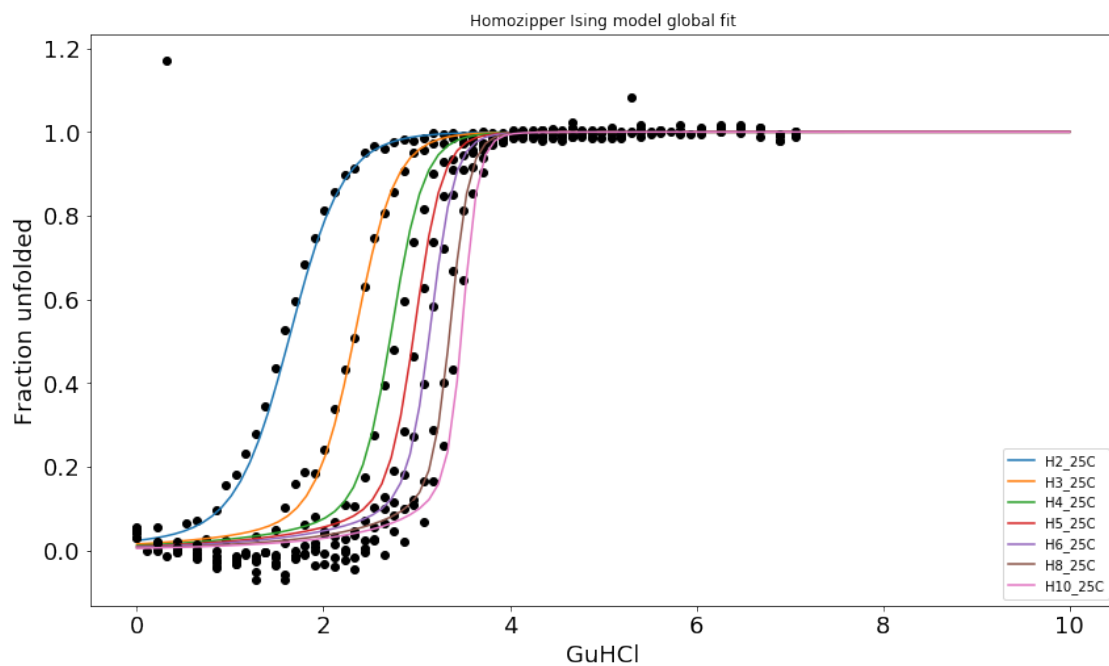
Fitting results

```

=====
ID: H10_25C
Model: HomozipperIsingEquilibrium
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(c) n          21.00000
(s) DG_intrinsic  3.52517 ± 0.00796      95% CI[  3.52315,   3.52719]
(s) m_intrinsic  -0.45188 ± 0.00090      95% CI[ -0.45211,  -0.45166]
(s) DG_interface -5.29438 ± 0.01119      95% CI[ -5.29721,  -5.29154]
-----
R^2: 0.99435
=====

```



End of this Notebook.

8 SI Notebook 6 - Fitting to a HeteroPolymer Ising Model

[Author] ERGM

In this notebook we will show how equilibrium folding data can be imported into a notebook and fitted to Heteropolymer Ising Model.

If you are less script/computer orientated, you can simply change the data paths and other parameters for your proteins and re-run the jupyter notebook (“Kernal/Restart & Run all” from the menu above).

8.1 Data Format

Please see PyFolding SI Notebooks 1 and 2 for the format your data has to be in to enable this type of analysis. Remember for Ising Model Analysis here each protein dataset (equilibrium denaturation curve) must have its own .csv

First off lets load pyfolding & pyplot into this ipython notebook (pyplot allows us to plot more complex figures of our results):

```
In [2]: # use this command to tell Jupyter to plot figures inline with the text
        %matplotlib inline
```

```
# import pyfolding, the pyfolding models and ising models
import pyfolding
from pyfolding import models, ising
```

```
# import the package for plotting, call it plt
import matplotlib.pyplot as plt
```

```
<IPython.core.display.Javascript object>
```

PyFolding: Jupyter autoscrolling has been disabled

Now, we need to load some data to analyse. I will import the equilibrium denaturation curves of a series of CTPRn variants proteins taken from:

Millership C., Phillips J.J. & Main E.R.G. (2016) “Ising model reprogramming of a repeat protein equilibrium unfolding pathway.” Journal of Molecular Biology, 428 (9A), 1804-1817.

I will load 12 denaturation curves (two repeats of each expt) that correspond to the following proteins:

Protein	Filename	No. of Helices	Truncation
CTPR2ΔA	CTPR2APhos1_10C.csv	4	N-Terminal helix removed
CTPR2ΔA	CTPR2APhos2_10C.csv	4	N-Terminal helix removed
CTPR2ΔS	CTPR2SPhos1_10C.csv	4	C-Terminal helix removed
CTPR2ΔS	CTPR2SPhos2_10C.csv	4	C-Terminal helix removed
CTPR2	CTPR2Phos1_10C.csv	5	none
CTPR2	CTPR2Phos1_10C.csv	5	none
CTPR3ΔA	CTPR3APhos1_10C.csv	6	N-Terminal helix removed
CTPR3ΔA	CTPR3APhos2_10C.csv	6	N-Terminal helix removed
CTPR3ΔS	CTPR3SPhos1_10C.csv	6	C-Terminal helix removed
CTPR3ΔS	CTPR3SPhos2_10C.csv	6	C-Terminal helix removed
CTPR3	CTPR3Phos1_10C.csv	7	none
CTPR3	CTPR3Phos1_10C.csv	7	none

8.2 Import Data, assign names and put into a list

```
In [3]: # start by loading a data set
# arguments are "path", "filename"
pth = "../examples/CTPRn"
CTPR2A1 = pyfolding.read_equilibrium_data(pth, "CTPR2APhos1_10C.csv")
CTPR2A2 = pyfolding.read_equilibrium_data(pth, "CTPR2APhos2_10C.csv")
CTPR2S1= pyfolding.read_equilibrium_data(pth, "CTPR2SPhos1_10C.csv")
CTPR2S2 = pyfolding.read_equilibrium_data(pth, "CTPR2SPhos2_10C.csv")
CTPR2_1= pyfolding.read_equilibrium_data(pth, "CTPR2Phos1_10C.csv")
CTPR2_2 = pyfolding.read_equilibrium_data(pth, "CTPR2Phos2_10C.csv")
CTPR3A1 = pyfolding.read_equilibrium_data(pth, "CTPR3APhos1_10C.csv")
CTPR3A2 = pyfolding.read_equilibrium_data(pth, "CTPR3APhos2_10C.csv")
CTPR3S1 = pyfolding.read_equilibrium_data(pth, "CTPR3SPhos1_10C.csv")
CTPR3S2 = pyfolding.read_equilibrium_data(pth, "CTPR3SPhos2_10C.csv")
CTPR3_1 = pyfolding.read_equilibrium_data(pth, "CTPR3Phos1_10C.csv")
CTPR3_2 = pyfolding.read_equilibrium_data(pth, "CTPR3Phos2_10C.csv")

# make a python list of the data
curves = [CTPR2A1,CTPR2A2,
           CTPR2S1,CTPR2S2,
           CTPR2_1,CTPR2_2,
           CTPR3A1,CTPR3A2,
           CTPR3S1,CTPR3S2,
           CTPR3_1, CTPR3_2]
```

8.3 Set Temperature

Here the expts were conducted at 10 degrees C, so lets set this up

```
In [4]: # set the temperature to 10 degrees C
```

```
pyfolding.set_temperature(10.0)
```

Set temperature to 10.00°C

(NOTE: Careful, this sets the temperature for all subsequent calculations)

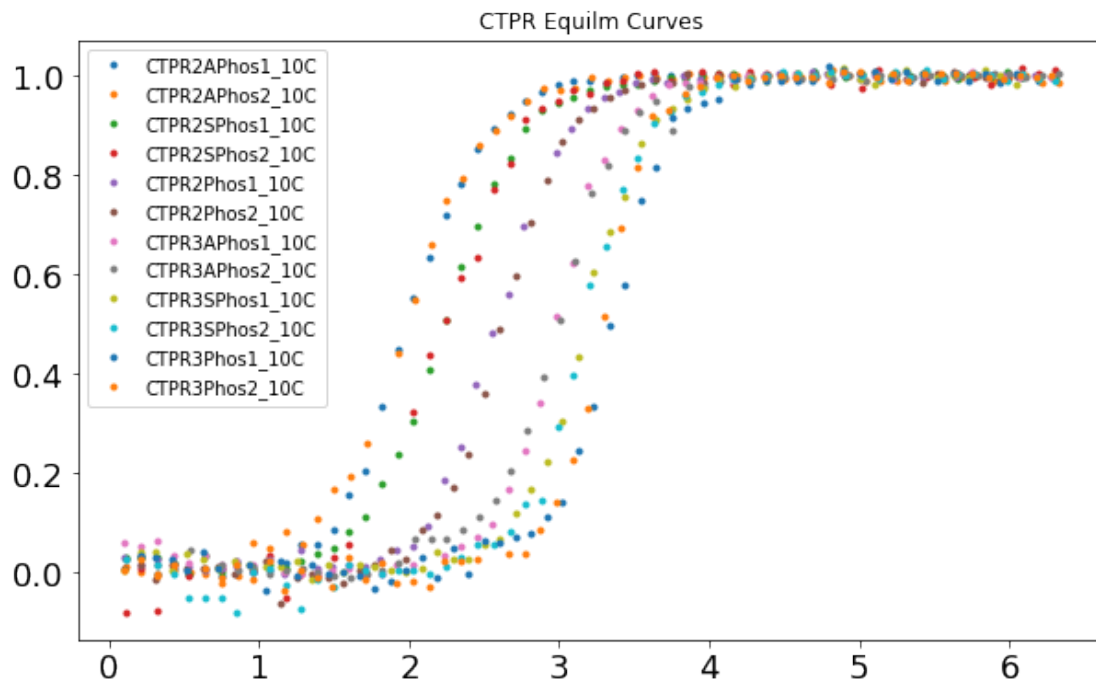
8.4 Plot the data

```
In [4]: # Lets check everything was loaded correctly by plotting all the data curves on one plot
```

```
plt.figure(figsize=(10,6))
for c in curves:
    plt.plot(c.x, c.y, '.')
```

```
# the following commands plot all the data curves on one plot,
# where "loc" command determines where the legend goes.
```

```
plt.legend([c.ID for c in curves], loc='best')
plt.title("CTPR Equilm Curves")
plt.show()
```



8.5 Fitting to a Heteropolymer Ising model

8.6 Assign Topology

We need to define a topology for each protein, this is essentially the domains of the protein in order (N->C)

Module	Abbreviation	Defined in Ising as
Module, type 1	r	RepeatDomain
Module, type 2	Mr	MutantRepeatDomain
Module, type 3	h	HelixDomain
Module, type 4	l	LoopDomain
Module, type 5	Ml	MutantLoopDomain
Cap Module	c	CapDomain
Mut. Cap Module	Mc	MutantCapDomain
Decoupled Cap Domain	DC	DecoupleCapDomain

Protein topology can be defined via the following Ising motifs All modules functions are identical, except for the Decoupled Cap Domain and Cap domains. 1. The Decoupled Cap domain has been coded such that there is no interface energy between itself and any domain N-terminal to it.

2. Cap domains are essentially identical to the other domains, except for clarity their interface energy term (G_{ij}) is not displayed when used in fitting. They should only be used at the C-terminus.

With regards to the other domains, you simply use them to define different modules present in your protein. The nomenclature stems from our various CTPR proteins analysis, as their repeated motifs can be defined in terms of either repeated TPRs, repeated helices and/or loops.

Each motif is defined as having:

1. an intrinsic stability term for each module (G_i)
2. a denaturation dependence term (m_i) - which is associated with G_i
3. an interface energy term (G_{ij}) - defined as the interaction between folded units i and $i+1$, i.e. the interface of the i unit with the unit C terminal to it ($i+1$).
4. The Ising model automatically takes into account the C-terminal repeat does not have an interface term $\{ij\}$ associated with it.

Except:

1. the Cap Motifs/Domains (Cap + Mutant Cap) do not print their interface term $\{ij\}$ for clarity.
2. Thus the Cap domain should only be used when you have a C-terminal deletion.
3. The Decoupled Cap Domain do not use an interface term (hence "decoupled")

Lets list the motifs already present in the Ising model to check that the above is correct!

```
In [5]: # This command lists the motifs already present in the Ising model.
        ising.list_models()
```

```
Out [5]: ['CapDomain',
          'DecoupleCapDomain',
          'HelixDomain',
          'LoopDomain',
          'MutantCapDomain',
          'MutantLoopDomain',
          'MutantRepeatDomain',
          'RepeatDomain',
          'RepeatDomain_mij']
```

8.7 Plot our Chosen Topology

```
In [6]: #Define shorthand for Ising modules:
```

```
    r = ising.RepeatDomain
    h = ising.HelixDomain
    Mr = ising.MutantRepeatDomain
    c = ising.CapDomain
    Mc = ising.MutantCapDomain
    l = ising.LoopDomain
    Ml = ising.MutantLoopDomain
```

```
#lets try topology A first, but need to add in double as two curves of each protein
```

```
topology1 = [[r,r,Mr,c],[r,r,Mr,c],
              [h,r,r,r],[h,r,r,r],
              [h,r,r,Mr,c],[h,r,r,Mr,c],
              [r,r,r,r,Mr,c],[r,r,r,r,Mr,c],
              [h,r,r,r,r,r],[h,r,r,r,r,r],
              [h,r,r,r,r,Mr,c],[h,r,r,r,r,Mr,c]]
```

```
# plot them either in an expanded form
```

```
ising.plot_domains(topology1, collapse=False)
```

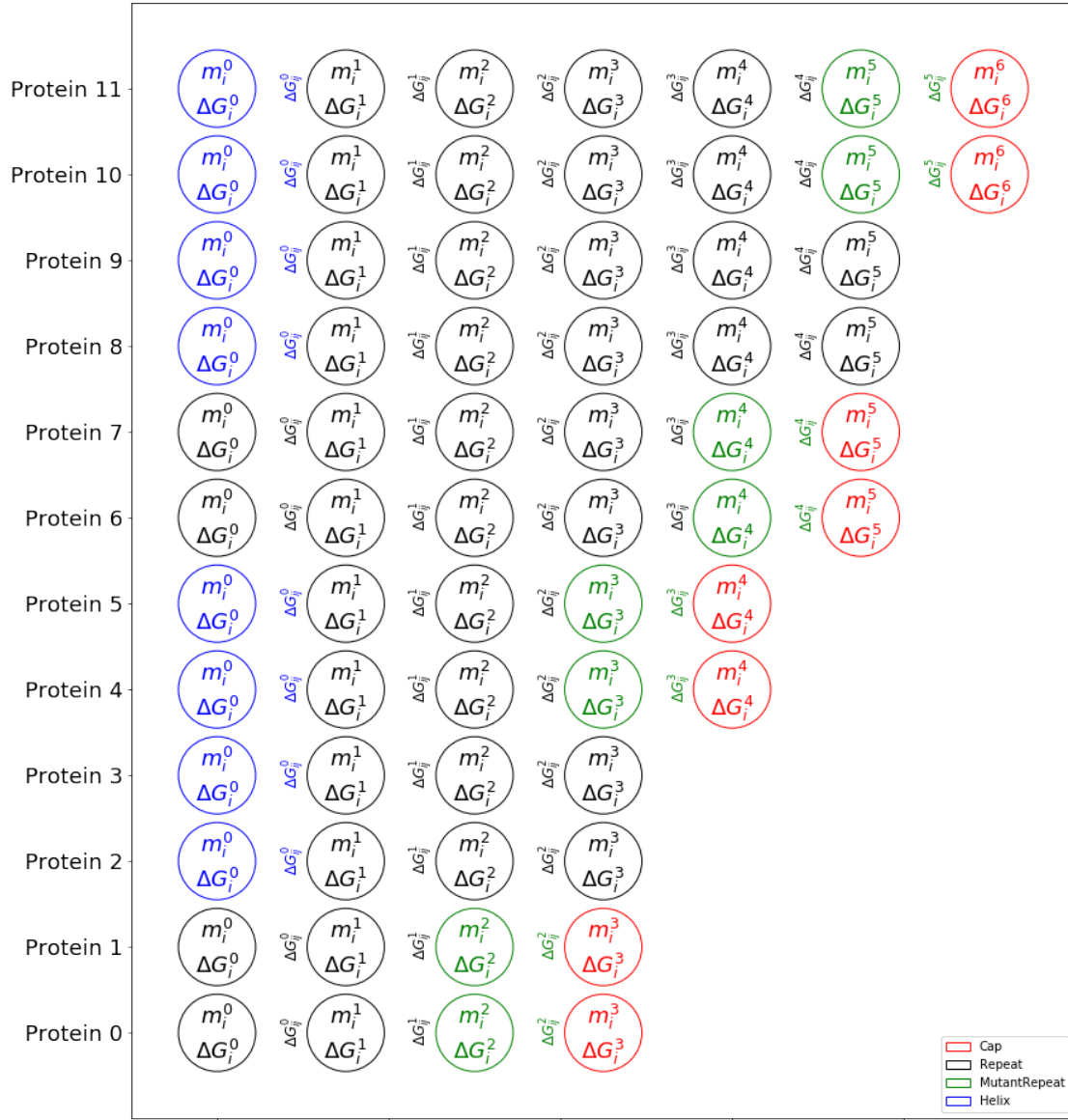
```
# or in a collapsed form
```

```
ising.plot_domains(topology1, collapse=True)
```

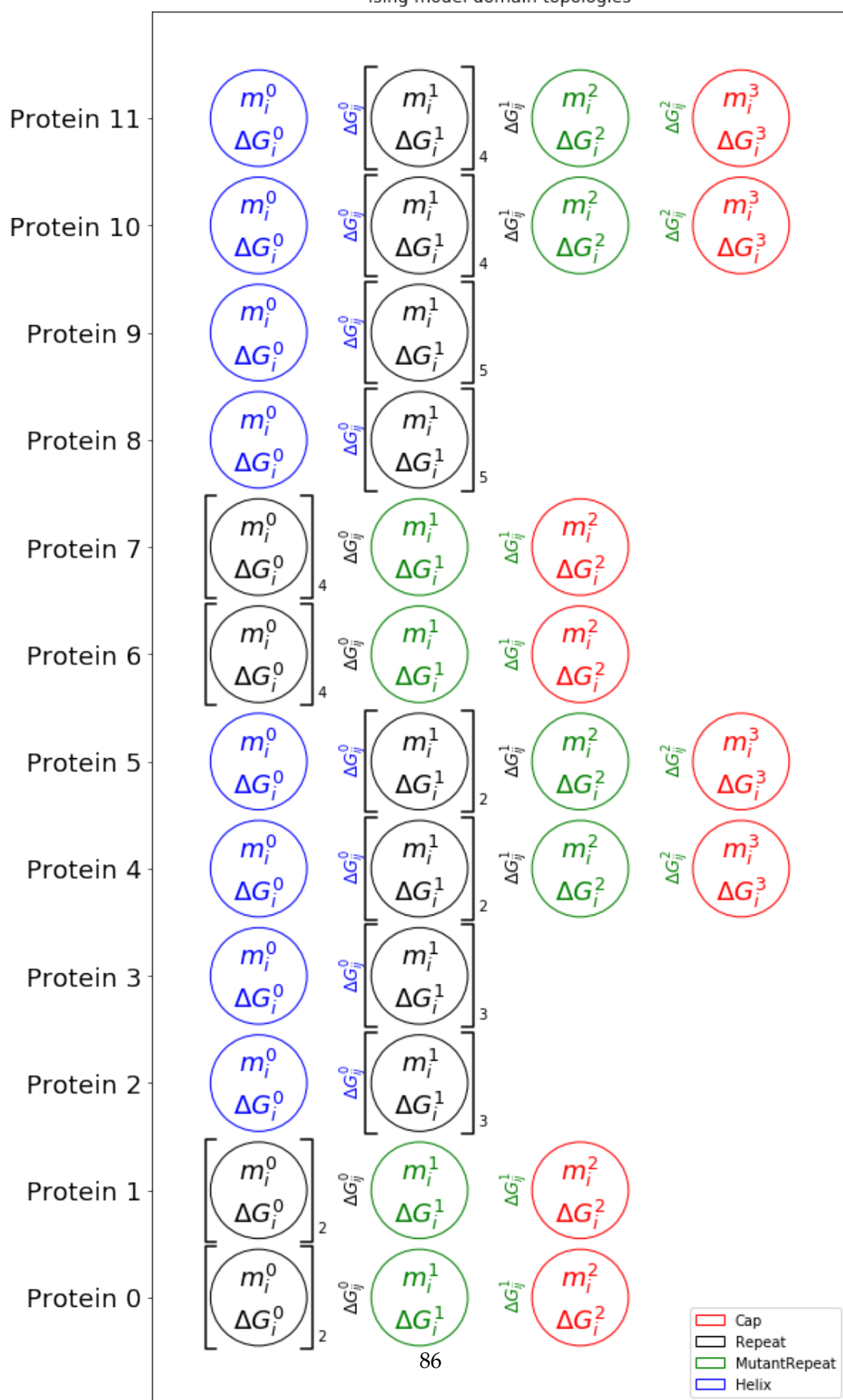
```
# NOTE: You can save the domain image by using the save keyword like this:
```

```
# ising.plot_domains([topology], collapse=True, save="/Users/ergm/Desktop/domains.pdf")
```

Ising model domain topologies



Ising model domain topologies



8.8 Fit Data with our chosen Topology

```
In [8]: # note: popsize is the population size for the optimiser (bigger means more likelihood of finding the best fit)
# start with a popsize of 10, increase to 100-1000 for better fits:
# curves defines the data you will fit, topologies define what it fits to
# ising.fit_heteropolymer(curves, topologies, popsize=10)

# for example, let's just fit the CTPRn datasets to topology 1:

ising.fit_heteropolymer(curves[0:12], topology1[0:12], popsize=10, maxiter= 10000)

# NOTE: You can save the fits using the save keyword like this:
# ising.fit_heteropolymer(curves[0:12], topology1[0:12], popsize=10, maxiter= 10000
#                               , save="/Users/ergm/Desktop/FitResults")
```

Appending 12 curves to GlobalFitIsing...

```
+ added CTPR2Aphos1_10C with topology ['Repeat', 'Repeat', 'MutantRepeat', 'Cap']
+ added CTPR2Aphos2_10C with topology ['Repeat', 'Repeat', 'MutantRepeat', 'Cap']
+ added CTPR2SPhos1_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR2SPhos2_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR2Phos1_10C with topology ['Helix', 'Repeat', 'Repeat', 'MutantRepeat', 'Cap']
+ added CTPR2Phos2_10C with topology ['Helix', 'Repeat', 'Repeat', 'MutantRepeat', 'Cap']
+ added CTPR3Aphos1_10C with topology ['Repeat', 'Repeat', 'Repeat', 'Repeat', 'MutantRepeat', 'Cap']
+ added CTPR3Aphos2_10C with topology ['Repeat', 'Repeat', 'Repeat', 'Repeat', 'MutantRepeat', 'Cap']
+ added CTPR3SPhos1_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR3SPhos2_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR3Phos1_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'MutantRepeat']
+ added CTPR3Phos2_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'MutantRepeat']
```

Performing global optimisation of Ising model (12 curves, Population size: 10, Tolerance: 1.00E-08)

```
- Fitting in progress (Iteration: 100, Convergence: 2.89931E-08, Timing: 2.32s per iteration)
- Fitting in progress (Iteration: 200, Convergence: 2.27786E-08, Timing: 2.26s per iteration)
- Fitting in progress (Iteration: 300, Convergence: 2.73710E-08, Timing: 2.44s per iteration)
- Fitting in progress (Iteration: 400, Convergence: 3.07232E-08, Timing: 2.34s per iteration)
- Fitting in progress (Iteration: 500, Convergence: 1.49071E-07, Timing: 9.31s per iteration)
- Fitting in progress (Iteration: 600, Convergence: 2.56571E-07, Timing: 13.23s per iteration)
- Fitting in progress (Iteration: 700, Convergence: 5.54906E-07, Timing: 13.57s per iteration)
- Fitting in progress (Iteration: 800, Convergence: 4.74560E-07, Timing: 2.42s per iteration)
- Fitting in progress (Iteration: 900, Convergence: 7.91965E-07, Timing: 2.15s per iteration)
- Fitting in progress (Iteration: 1000, Convergence: 2.57162E-06, Timing: 2.03s per iteration)
- Fitting in progress (Iteration: 1100, Convergence: 6.80320E-06, Timing: 2.03s per iteration)
- Fitting in progress (Iteration: 1200, Convergence: 1.37043E-05, Timing: 2.03s per iteration)
- Fitting in progress (Iteration: 1300, Convergence: 3.58656E-05, Timing: 2.03s per iteration)
- Fitting in progress (Iteration: 1400, Convergence: 8.46884E-05, Timing: 2.31s per iteration)
- Fitting in progress (Iteration: 1500, Convergence: 2.48855E-04, Timing: 2.25s per iteration)
```

- Fitting in progress (Iteration: 1600, Convergence: 5.63081E-04, Timing: 2.04s per iteration)
- Fitting in progress (Iteration: 1700, Convergence: 1.16502E-03, Timing: 2.26s per iteration)
- Fitting in progress (Iteration: 1800, Convergence: 3.73319E-03, Timing: 2.27s per iteration)
- Fitting in progress (Iteration: 1900, Convergence: 9.52760E-03, Timing: 2.37s per iteration)
- Fitting in progress (Iteration: 2000, Convergence: 2.83025E-02, Timing: 2.42s per iteration)
- Fitting in progress (Iteration: 2100, Convergence: 7.43200E-02, Timing: 2.40s per iteration)
- Fitting in progress (Iteration: 2200, Convergence: 1.38931E-01, Timing: 2.42s per iteration)
- Fitting in progress (Iteration: 2300, Convergence: 3.92334E-01, Timing: 2.18s per iteration)
- Fitting in progress (Iteration: 2400, Convergence: 8.44591E-01, Timing: 2.24s per iteration)

Fitting results (NOTE: Careful with the errors here):

Fitting results

ID: CTPR2APhos1_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405,	3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644,	-5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569,	-0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393,	5.21801]
(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632,	-9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255,	0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377,	4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021,	-1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435,	7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565,	-9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898,	0.07392]

R²: 0.99891

Fitting results

ID: CTPR2APhos2_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405,	3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644,	-5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569,	-0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393,	5.21801]

(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99894

Fitting results

ID: CTPR2SPhos1_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]
(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99946

Fitting results

ID: CTPR2SPhos2_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]

(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99492

Fitting results

ID: CTPR2Phos1_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]
(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99864

Fitting results

ID: CTPR2Phos2_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]

(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99897

Fitting results

ID: CTPR3APhos1_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]
(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99848

Fitting results

ID: CTPR3APhos2_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]

(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99813

Fitting results

ID: CTPR3SPhos1_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]
(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99877

Fitting results

ID: CTPR3SPhos2_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]

(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99731

Fitting results

ID: CTPR3Phos1_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]
(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99722

Fitting results

ID: CTPR3Phos2_10C

Model: Heteropolymer Ising Model

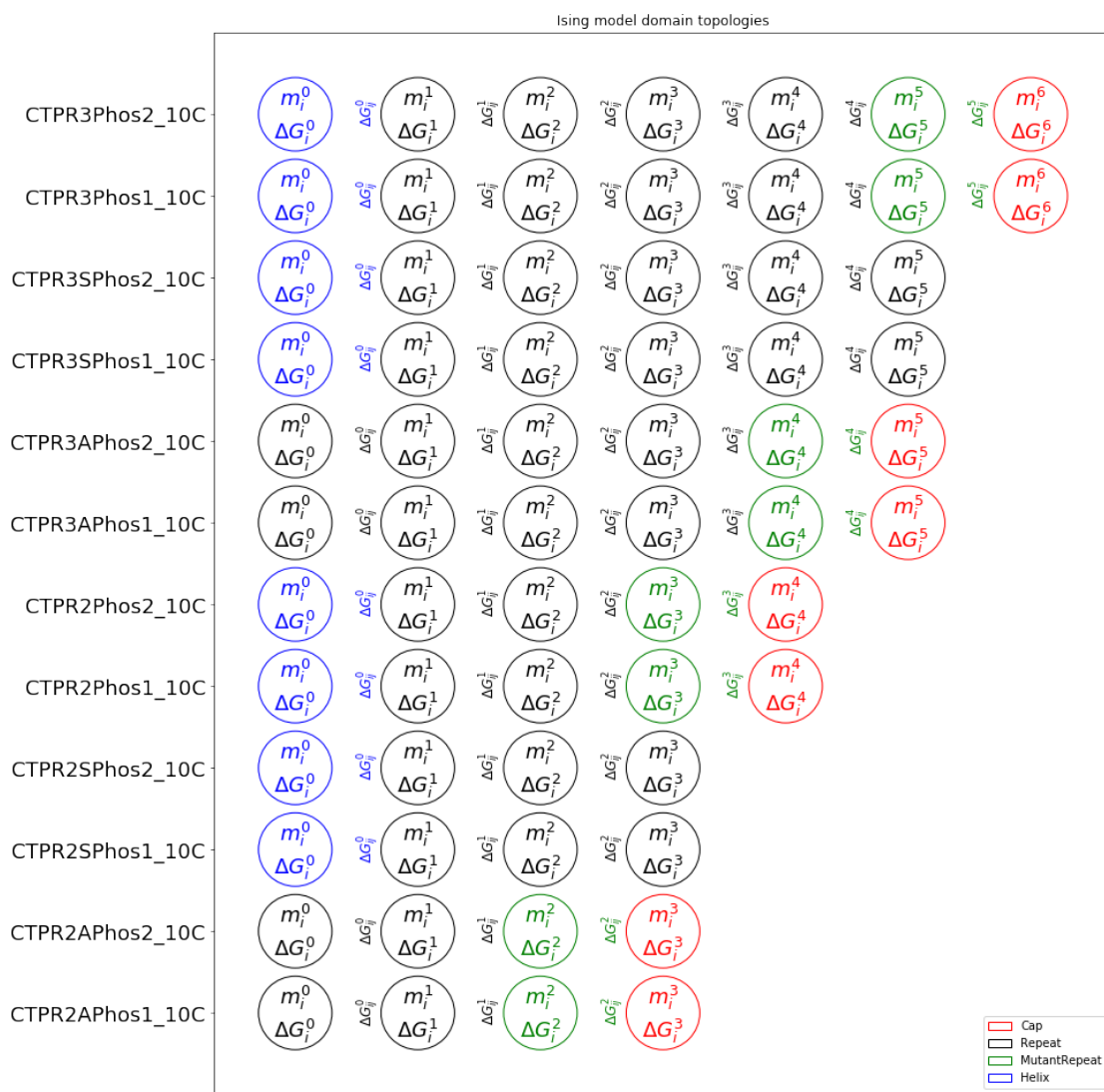
Optimiser: scipy.optimize.differential_evolution

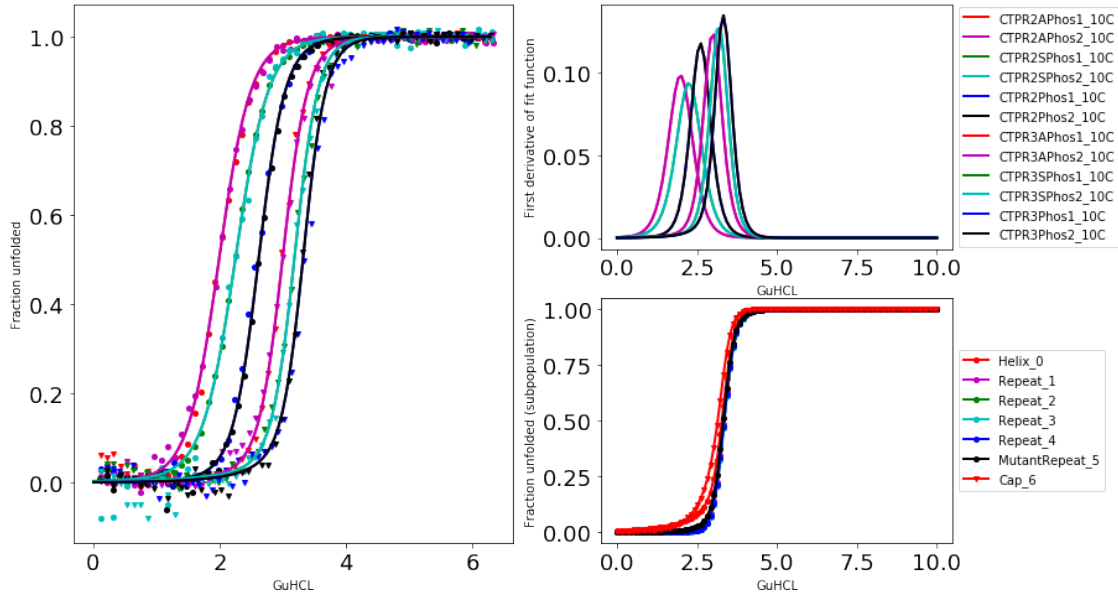
Temperature: 10.00°C

(s) Repeat DG_i	3.46641 ± 0.91527	95% CI[3.23405, 3.69877]
(s) Repeat DG_ij	-6.15007 ± 0.61596	95% CI[-6.30644, -5.99369]
(s) Repeat m_i	-0.53090 ± 0.21582	95% CI[-0.58569, -0.47611]
(s) MutantRepeat DG_i	5.17597 ± 0.16560	95% CI[5.13393, 5.21801]

(s) MutantRepeat DG_ij	-9.28958 ± 0.42047	95% CI[-9.39632, -9.18283]
(s) MutantRepeat m_i	-0.00790 ± 0.13650	95% CI[-0.04255, 0.02676]
(s) Cap DG_i	4.89001 ± 0.41848	95% CI[4.78377, 4.99625]
(s) Cap m_i	-1.25264 ± 0.93580	95% CI[-1.49021, -1.01507]
(s) Helix DG_i	6.87395 ± 1.29828	95% CI[6.54435, 7.20354]
(s) Helix DG_ij	-9.94555 ± 1.30027	95% CI[-10.27565, -9.61545]
(s) Helix m_i	-0.63753 ± 2.80242	95% CI[-1.34898, 0.07392]

R²: 0.99765





In [9]: *#Now do Topology 2*

```
topology2 = [[r,r,r,c],[r,r,r,c],
             [h,r,r,r],[h,r,r,r],
             [h,r,r,r,c],[h,r,r,r,c],
             [r,r,r,r,r,c],[r,r,r,r,r,c],
             [h,r,r,r,r,r],[h,r,r,r,r,r],
             [h,r,r,r,r,r,c],[h,r,r,r,r,r,c]]
```

#Fit

```
ising.fit_heteropolymer(curves[0:12], topology2[0:12], popsize=10, maxiter= 10000)
```

Appending 12 curves to GlobalFitIsing...

```
+ added CTPR2APhos1_10C with topology ['Repeat', 'Repeat', 'Repeat', 'Cap']
+ added CTPR2APhos2_10C with topology ['Repeat', 'Repeat', 'Repeat', 'Cap']
+ added CTPR2SPhos1_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR2SPhos2_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR2Phos1_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Cap']
+ added CTPR2Phos2_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Cap']
+ added CTPR3APhos1_10C with topology ['Repeat', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'Cap']
+ added CTPR3APhos2_10C with topology ['Repeat', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'Cap']
+ added CTPR3SPhos1_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR3SPhos2_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR3Phos1_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'Repeat']
+ added CTPR3Phos2_10C with topology ['Helix', 'Repeat', 'Repeat', 'Repeat', 'Repeat', 'Repeat']
```

Performing global optimisation of Ising model (12 curves, Population size: 10, Tolerance: 1.00E-
- Fitting in progress (Iteration: 100, Convergence: 2.66817E-08, Timing: 1.68s per iteration)

- Fitting in progress (Iteration: 200, Convergence: 1.06884E-07, Timing: 1.66s per iteration)
- Fitting in progress (Iteration: 300, Convergence: 1.65748E-06, Timing: 1.70s per iteration)
- Fitting in progress (Iteration: 400, Convergence: 6.98981E-05, Timing: 1.67s per iteration)
- Fitting in progress (Iteration: 500, Convergence: 1.74531E-03, Timing: 1.60s per iteration)
- Fitting in progress (Iteration: 600, Convergence: 6.79443E-02, Timing: 1.68s per iteration)
- Fitting in progress (Iteration: 700, Convergence: 6.93502E-01, Timing: 1.79s per iteration)

Fitting results (NOTE: Careful with the errors here):

Fitting results

ID: CTPR2APhos1_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R²: 0.99874

Fitting results

ID: CTPR2APhos2_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R²: 0.99901

=====

Fitting results

=====

ID: CTPR2SPhos1_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R²: 0.99952

=====

=====

Fitting results

=====

ID: CTPR2SPhos2_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R²: 0.99491

=====

=====

Fitting results

=====

ID: CTPR2Phos1_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

 R^2: 0.99823
 =====

=====

Fitting results

=====

ID: CTPR2Phos2_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

 R^2: 0.99888
 =====

=====

Fitting results

=====

ID: CTPR3APhos1_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
-----------------	-------------------	---------	----------	----------

(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R²: 0.99838
=====

=====

Fitting results

=====

ID: CTPR3APhos2_10C
Model: Heteropolymer Ising Model
Optimiser: scipy.optimize.differential_evolution
Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R²: 0.99824
=====

=====

Fitting results

=====

ID: CTPR3SPhos1_10C
Model: Heteropolymer Ising Model
Optimiser: scipy.optimize.differential_evolution
Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]

(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]
---------------	--------------------	---------	-----------	----------

R^2: 0.99878
=====

=====

Fitting results

=====

ID: CTPR3SPhos2_10C
Model: Heteropolymer Ising Model
Optimiser: scipy.optimize.differential_evolution
Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R^2: 0.99734
=====

=====

Fitting results

=====

ID: CTPR3Phos1_10C
Model: Heteropolymer Ising Model
Optimiser: scipy.optimize.differential_evolution
Temperature: 10.00°C

(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R^2: 0.99653
=====

=====

Fitting results

=====

ID: CTPR3Phos2_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

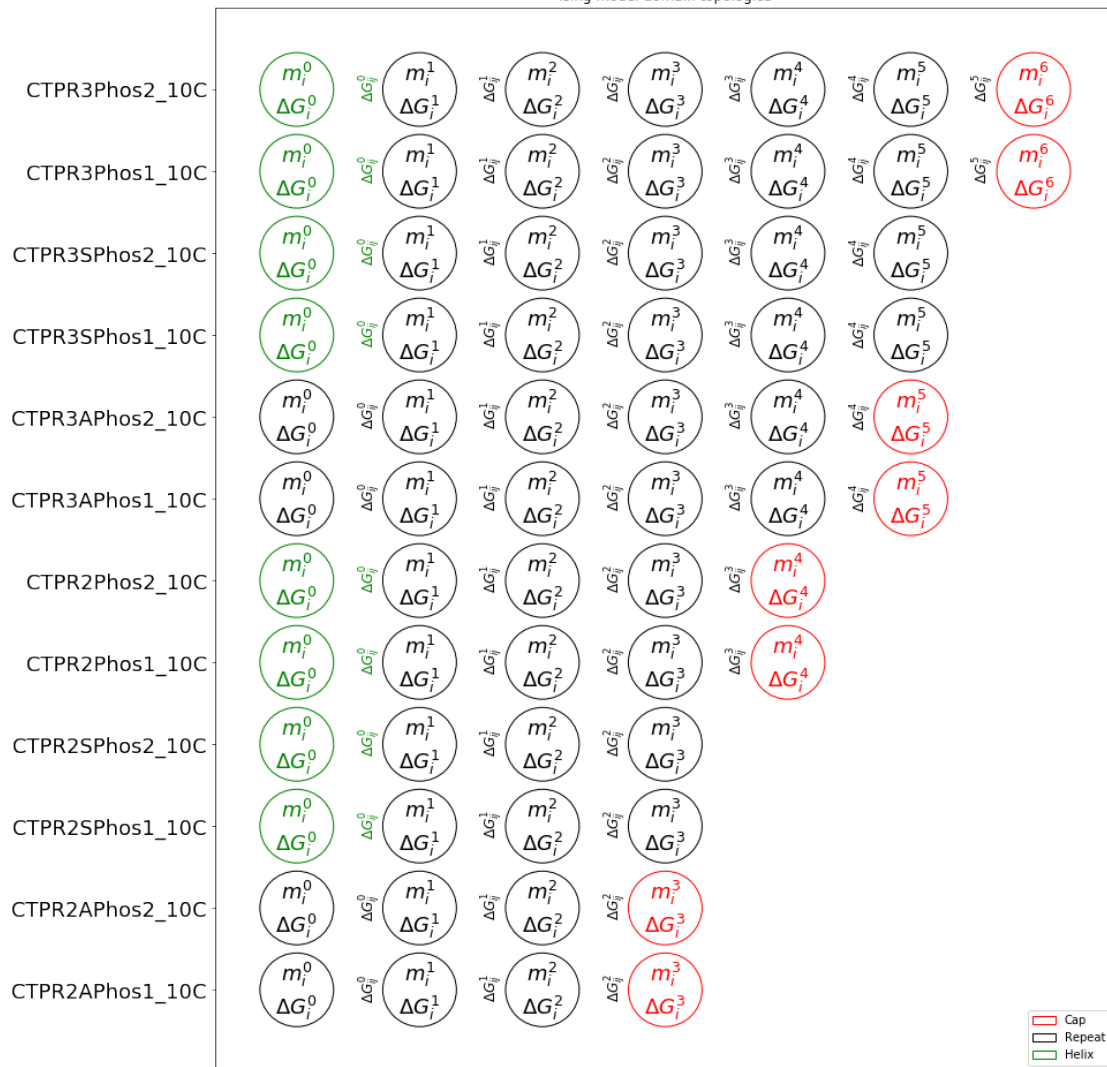
Temperature: 10.00°C

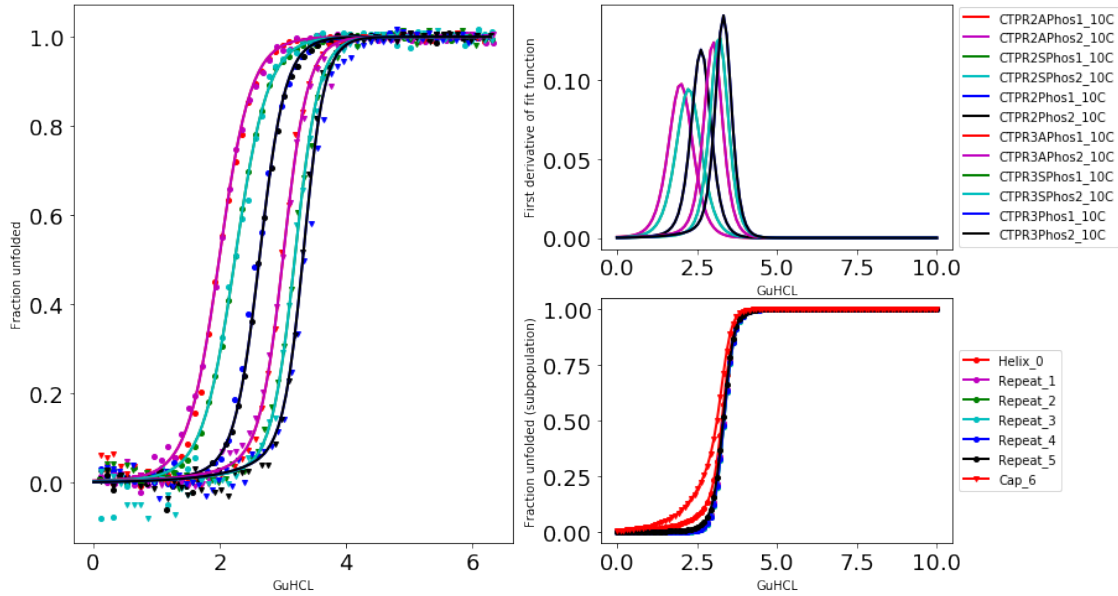
(s) Repeat DG_i	3.58860 ± 1.13612	95% CI[3.30017,	3.87702]
(s) Repeat DG_ij	-6.23306 ± 1.48678	95% CI[-6.61051,	-5.85561]
(s) Repeat m_i	-0.51206 ± 0.78378	95% CI[-0.71104,	-0.31308]
(s) Cap DG_i	3.29234 ± 0.94705	95% CI[3.05191,	3.53277]
(s) Cap m_i	-0.81869 ± 3.29559	95% CI[-1.65535,	0.01796]
(s) Helix DG_i	6.72118 ± 1.47219	95% CI[6.34744,	7.09493]
(s) Helix DG_ij	-9.99497 ± 1.47612	95% CI[-10.36971,	-9.62022]
(s) Helix m_i	-0.70077 ± 4.45640	95% CI[-1.83212,	0.43058]

R²: 0.99794

=====

Ising model domain topologies





In [10]: *#Now do Topology 3*

```
topology3 = [[h,r,h,c], [h,r,h,c],
             [r,h,r,h], [r,h,r,h],
             [r,h,r,h,c], [r,h,r,h,c],
             [h,r,h,r,h,c], [h,r,h,r,h,c],
             [r,h,r,h,r,h], [r,h,r,h,r,h],
             [r,h,r,h,r,h,c], [r,h,r,h,r,h,c]]
```

#fit

```
ising.fit_heteropolymer(curves[0:12], topology3[0:12], popsize=10, maxiter= 10000)
```

Appending 12 curves to GlobalFitIsing...

```
+ added CTPR2APhos1_10C with topology ['Helix', 'Repeat', 'Helix', 'Cap']
+ added CTPR2APhos2_10C with topology ['Helix', 'Repeat', 'Helix', 'Cap']
+ added CTPR2SPhos1_10C with topology ['Repeat', 'Helix', 'Repeat', 'Helix']
+ added CTPR2SPhos2_10C with topology ['Repeat', 'Helix', 'Repeat', 'Helix']
+ added CTPR2Phos1_10C with topology ['Repeat', 'Helix', 'Repeat', 'Helix', 'Cap']
+ added CTPR2Phos2_10C with topology ['Repeat', 'Helix', 'Repeat', 'Helix', 'Cap']
+ added CTPR3APhos1_10C with topology ['Helix', 'Repeat', 'Helix', 'Repeat', 'Helix', 'Cap']
+ added CTPR3APhos2_10C with topology ['Helix', 'Repeat', 'Helix', 'Repeat', 'Helix', 'Cap']
+ added CTPR3SPhos1_10C with topology ['Repeat', 'Helix', 'Repeat', 'Helix', 'Repeat', 'Helix']
+ added CTPR3SPhos2_10C with topology ['Repeat', 'Helix', 'Repeat', 'Helix', 'Repeat', 'Helix']
+ added CTPR3Phos1_10C with topology ['Repeat', 'Helix', 'Repeat', 'Helix', 'Repeat', 'Helix']
+ added CTPR3Phos2_10C with topology ['Repeat', 'Helix', 'Repeat', 'Helix', 'Repeat', 'Helix']
```

Performing global optimisation of Ising model (12 curves, Population size: 10, Tolerance: 1.00E-
- Fitting in progress (Iteration: 100, Convergence: 1.74023E-08, Timing: 1.66s per iteration)

- Fitting in progress (Iteration: 200, Convergence: 5.09258E-08, Timing: 1.73s per iteration)
- Fitting in progress (Iteration: 300, Convergence: 6.14388E-07, Timing: 1.91s per iteration)
- Fitting in progress (Iteration: 400, Convergence: 8.90962E-06, Timing: 1.91s per iteration)
- Fitting in progress (Iteration: 500, Convergence: 1.73486E-04, Timing: 1.88s per iteration)
- Fitting in progress (Iteration: 600, Convergence: 1.09176E-03, Timing: 2.24s per iteration)
- Fitting in progress (Iteration: 700, Convergence: 3.72576E-03, Timing: 2.30s per iteration)
- Fitting in progress (Iteration: 800, Convergence: 2.86758E-02, Timing: 1.76s per iteration)
- Fitting in progress (Iteration: 900, Convergence: 1.94056E-01, Timing: 1.87s per iteration)
- Fitting in progress (Iteration: 1000, Convergence: 4.83044E-01, Timing: 1.87s per iteration)

Fitting results (NOTE: Careful with the errors here):

Fitting results

ID: CTPR2APhos1_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R²: 0.99867

Fitting results

ID: CTPR2APhos2_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]

(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]
-------------	--------------------	---------	-----------	-----------

R^2: 0.99903
=====

=====

Fitting results

=====

ID: CTPR2SPhos1_10C
Model: Heteropolymer Ising Model
Optimiser: scipy.optimize.differential_evolution
Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R^2: 0.99946
=====

=====

Fitting results

=====

ID: CTPR2SPhos2_10C
Model: Heteropolymer Ising Model
Optimiser: scipy.optimize.differential_evolution
Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R^2: 0.99496
=====

=====

Fitting results

=====

ID: CTPR2Phos1_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R^2: 0.99844

=====

=====

Fitting results

=====

ID: CTPR2Phos2_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution
 Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R^2: 0.99881

=====

=====

Fitting results

=====

ID: CTPR3APhos1_10C
 Model: Heteropolymer Ising Model
 Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R^2: 0.99844
=====

=====

Fitting results

=====

ID: CTPR3APhos2_10C
Model: Heteropolymer Ising Model
Optimiser: scipy.optimize.differential_evolution
Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R^2: 0.99816
=====

=====

Fitting results

=====

ID: CTPR3SPhos1_10C
Model: Heteropolymer Ising Model
Optimiser: scipy.optimize.differential_evolution
Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]

(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R²: 0.99844

Fitting results

ID: CTPR3SPhos2_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R²: 0.99747

Fitting results

ID: CTPR3Phos1_10C

Model: Heteropolymer Ising Model

Optimiser: scipy.optimize.differential_evolution

Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R²: 0.99655

Fitting results

ID: CTPR3Phos2_10C

Model: Heteropolymer Ising Model

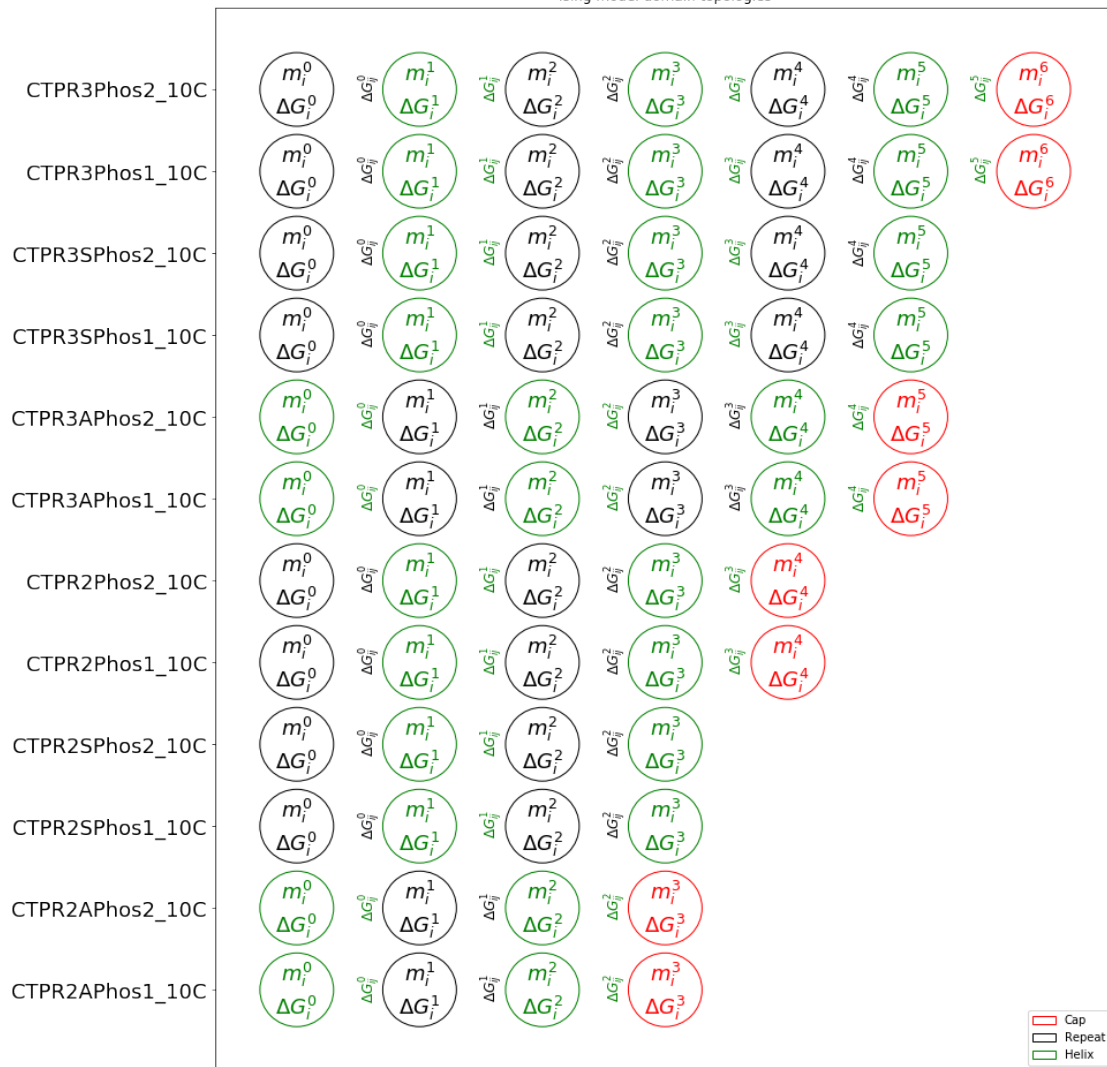
Optimiser: scipy.optimize.differential_evolution

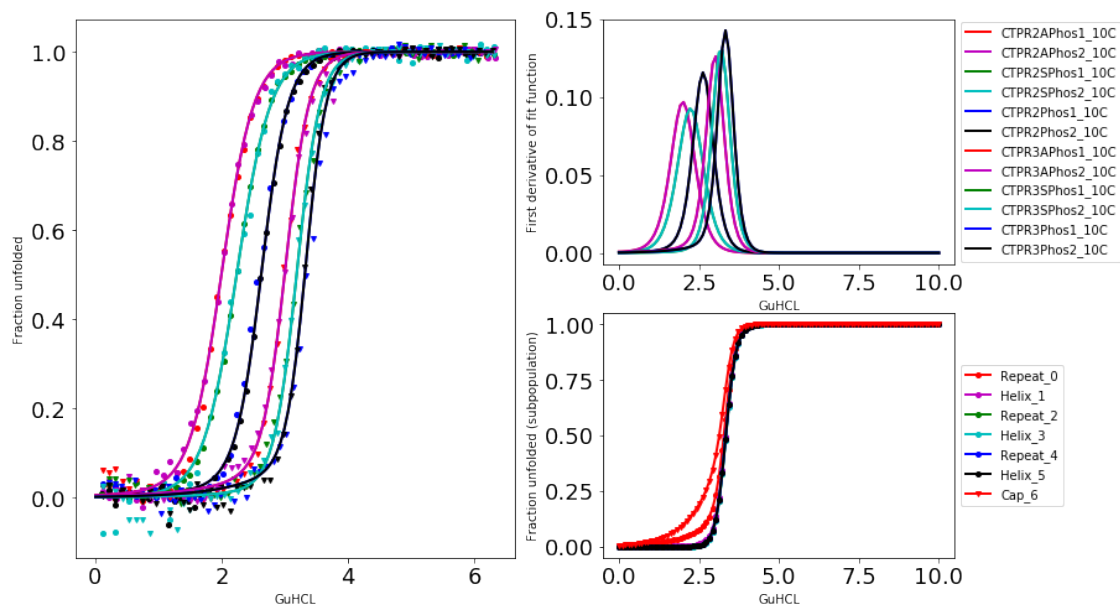
Temperature: 10.00°C

(s) Helix DG_i	3.63211 ± 4.48305	95% CI[2.49399,	4.77023]
(s) Helix DG_ij	-6.01364 ± 0.72695	95% CI[-6.19819,	-5.82909]
(s) Helix m_i	-0.46349 ± 0.61955	95% CI[-0.62077,	-0.30620]
(s) Repeat DG_i	6.99113 ± 2.24650	95% CI[6.42081,	7.56145]
(s) Repeat DG_ij	-9.99998 ± 1.59313	95% CI[-10.40444,	-9.59553]
(s) Repeat m_i	-0.60650 ± 5.86237	95% CI[-2.09479,	0.88179]
(s) Cap DG_i	3.14314 ± 1.33059	95% CI[2.80534,	3.48094]
(s) Cap m_i	-0.81343 ± 2.57567	95% CI[-1.46732,	-0.15954]

R²: 0.99795

Ising model domain topologies





End of this Notebook.

9 SI Notebook 7 - Simulating Ising Models

[Author] ARL & ERGM

Here we are going to generate an Ising model representation of a protein and simulate the unfolding curve. There are several steps:

1. Initialise domains with their appropriate values (ΔG , m etc..)
2. Build a topology with these domains
3. Create the partition function
4. Simulate!

Let's start with loading PyFolding:

```
In [1]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

import pyfolding
from pyfolding import ising, models
```

<IPython.core.display.Javascript object>

PyFolding: Jupyter autoscrolling has been disabled

```
In [2]: # Lets see what modules we have by listing them
ising.list_models()
```

```
Out[2]: ['CapDomain',
         'DecoupleCapDomain',
         'HelixDomain',
         'LoopDomain',
         'MutantCapDomain',
         'MutantLoopDomain',
         'MutantRepeatDomain',
         'RepeatDomain']
```

9.1 Initialise domains and give them their appropriate values

1st, we need some domains to build our protein. Let's use a cap domain and a repeat domain:

```
In [3]: # NOTE: we need to instantiate these domains, hence the () after their names
```

```
cap = ising.CapDomain()
repeat = ising.RepeatDomain()
```

2nd, we need to set the domain properties:

```
In [4]: cap.DG_i = -5.
        cap.m_i = -1.5

        repeat.DG_i = -1.
        repeat.m_i = -2.5
        repeat.DG_ij = -1.
```

9.2 Build a topology

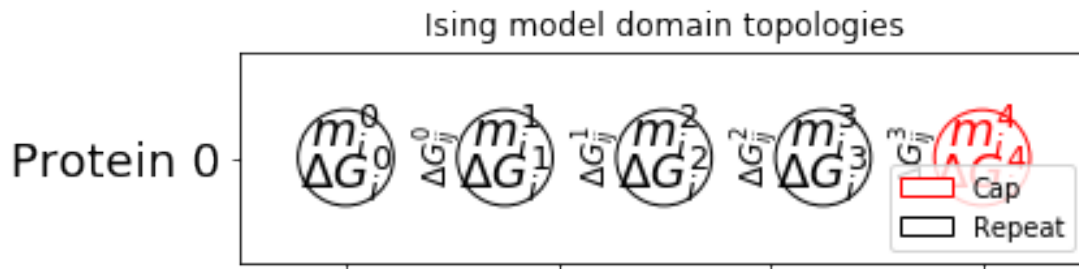
Having defined the properties of the domains, we need to design a protein topology to represent the protein we're simulating. Let's use a cap and three repeats to define our simulated protein:

```
In [5]: topology = [repeat, repeat, repeat, repeat, cap]
```

Let's plot our topology to make sure we've done everything correctly:

```
In [6]: # NOTE: You can now save the domain image by using the save keyword:
        # ising.plot_domains([topology], save="/Users/ubcg83a/Desktop/domains.pdf")

        ising.plot_domains([topology], collapse=False) #, save="/Users/ERGM/Desktop/domains.pdf"
```



9.3 Generate partition function

Having defined our protein topology, we can generate the partition function to go along with this:

```
In [7]: partition = ising.IsingPartitionFunction( topology )
        pyfolding.set_temperature(10.)
```

Set temperature to 10.00°C

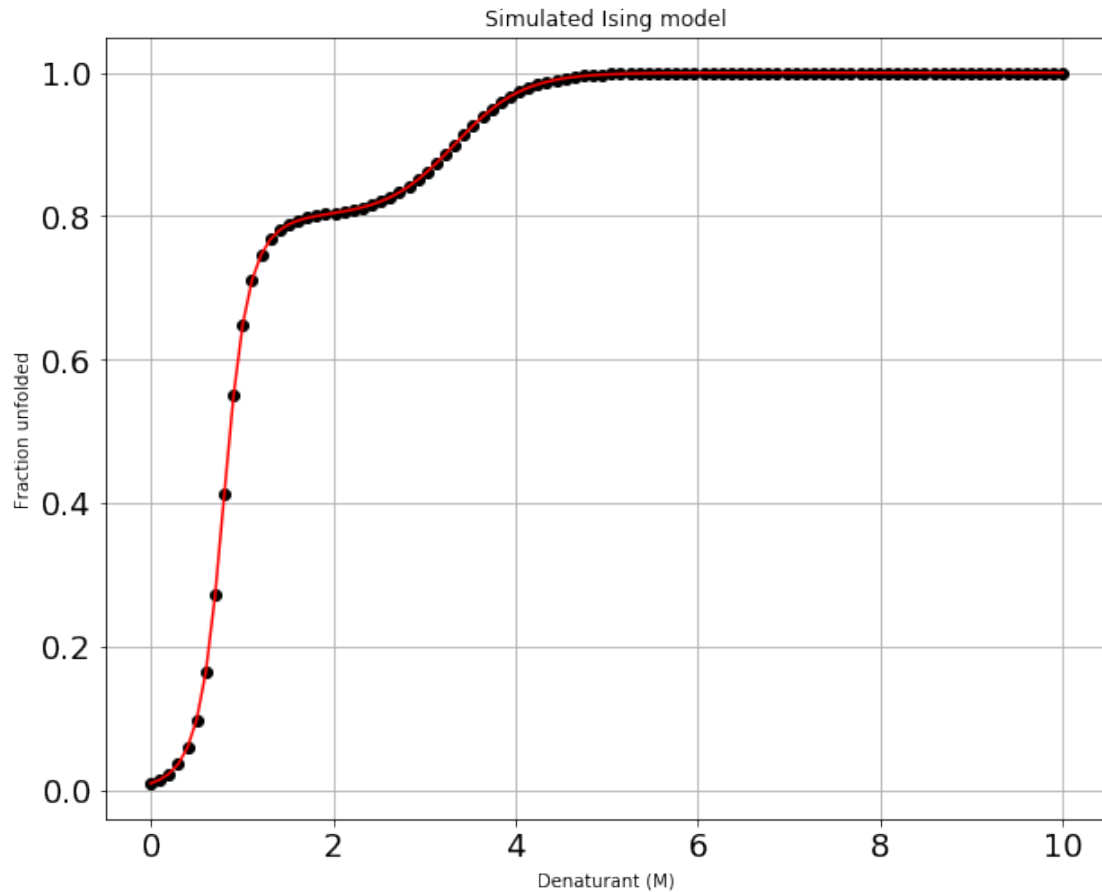
(NOTE: Careful, this sets the temperature for all subsequent calculations)

9.4 Simulate

Now we can simulate our protein unfolding over a certain concentration range:

```
In [8]: x = np.linspace(0.,10.,100) # range of denaturants, 0-10M with 100 points (i.e. every 0.1M)
        y = partition.theta(x)

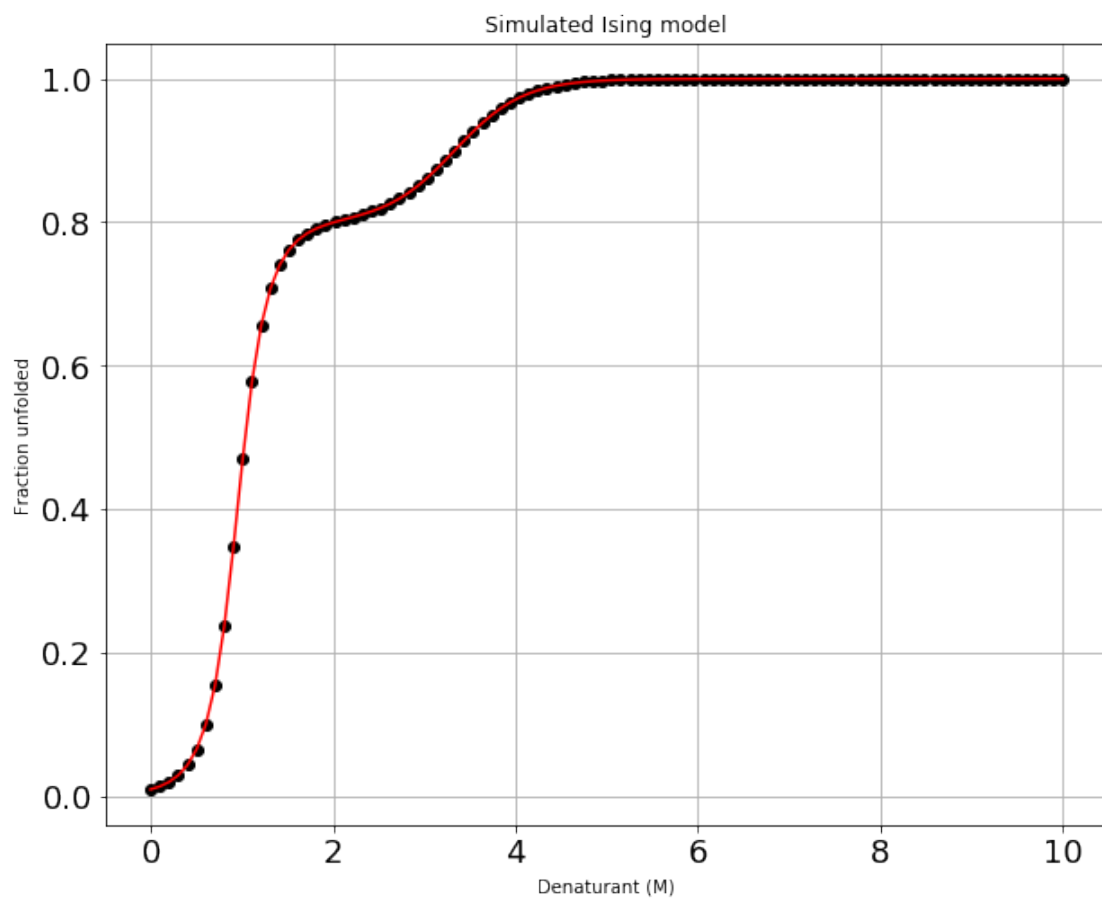
        plt.figure(figsize=(10,8))
        plt.plot(x,y,'ko', x,y,'r-')
        plt.title("Simulated Ising model")
        plt.xlabel('Denaturant (M)')
        plt.ylabel('Fraction unfolded')
        plt.grid(True)
        plt.show()
```



Try changing a value of the model! Let's set m_i for the repeat to -2.1:

```
In [9]: repeat.m_i = -2.1
        y = partition.theta(x)

        plt.figure(figsize=(10,8))
        plt.plot(x,y,'ko', x,y,'r-')
        plt.title("Simulated Ising model")
        plt.xlabel('Denaturant (M)')
        plt.ylabel('Fraction unfolded')
        plt.grid(True)
        plt.show()
```



End of this Notebook.

10 SI Notebook 8 - Global fit Equilibrium Curves from a dimeric protein to dimeric equilibrium models

[Authors] ERGM & ARL

In this notebook we will show how equilibrium folding data from dimeric proteins can be imported into a notebook and fitted to dimeric Equilibrium denaturation models using global fits over multiple datasets.

If you are less script/computer orientated, you can simply change the data paths and other parameters for your proteins and re-run the jupyter notebook (“Kernal/Restart & Run all” from the menu above).

10.1 Data Format

Please see PyFolding SI Notebooks 1 and 2 for the format your data has to be in to enable this type of analysis. Remember for Ising Model Analysis here each protein dataset (equilibrium denaturation curve) must have its own .csv

First off lets load pyfolding & pyplot into this ipython notebook (pyplot allows us to plot more complex figures of our results):

```
In [1]: # use this command to tell Jupyter to plot figures inline with the text
        %matplotlib inline

        # import pyfolding, the pyfolding models and ising models
        import pyfolding
        from pyfolding import *

        # import the package for plotting, call it plt
        import matplotlib.pyplot as plt

        # import numpy as well
        import numpy as np
```

<IPython.core.display.Javascript object>

PyFolding: Jupyter autoscrolling has been disabled

Now, we need to load some data to analyse. I will import the equilibrium denaturations of dimeric tr-LcrH at a series of protein concentrations from:

Singh S.K., Boyle A.L. & Main E.R. (2013) "LcrH, a class II chaperone from the type three secretion system, has a highly flexible native structure." J Biol Chem., 288, (6) 4048-55.

I will load 4 denaturation curves that correspond tr-LcrH at the following protein concentrations:

Protein	Filename	Total Protein Concentration
tr-LcrH	50uM_1.csv	50 uM
tr-LcrH	50uM_2.csv	50 uM
tr-LcrH	80uM_1.csv	80 uM
tr-LcrH	80uM_2.csv	80 uM

10.2 Import Data, assign names and put into a list

In [2]: # we will load all of the data together, as follows:

```
# arguments are "path", "filename"
pth = "../examples/LcrH"

# this is a set of commands to automate loading the data
# for each denaturation
fn = ["50uM_1.csv", "50uM_2.csv", "80uM_1.csv", "80uM_2.csv"]

# Here we are loading all the curves in a list called "proteins"
# and assigning them names
proteins = [pyfolding.read_equilibrium_data(pth,f) for f in fn]

# also store the total protein concentration for each denaturation
Pt = [50e-6, 50e-6, 80e-6, 80e-6]
```

10.3 Set Temperature

```
In [3]: # Set temperature to 25.00°C
# (NOTE: Careful, this sets the temperature for all subsequent calculations)
pyfolding.set_temperature(25.)
```

Set temperature to 25.00°C

(NOTE: Careful, this sets the temperature for all subsequent calculations)

10.4 Plot Data

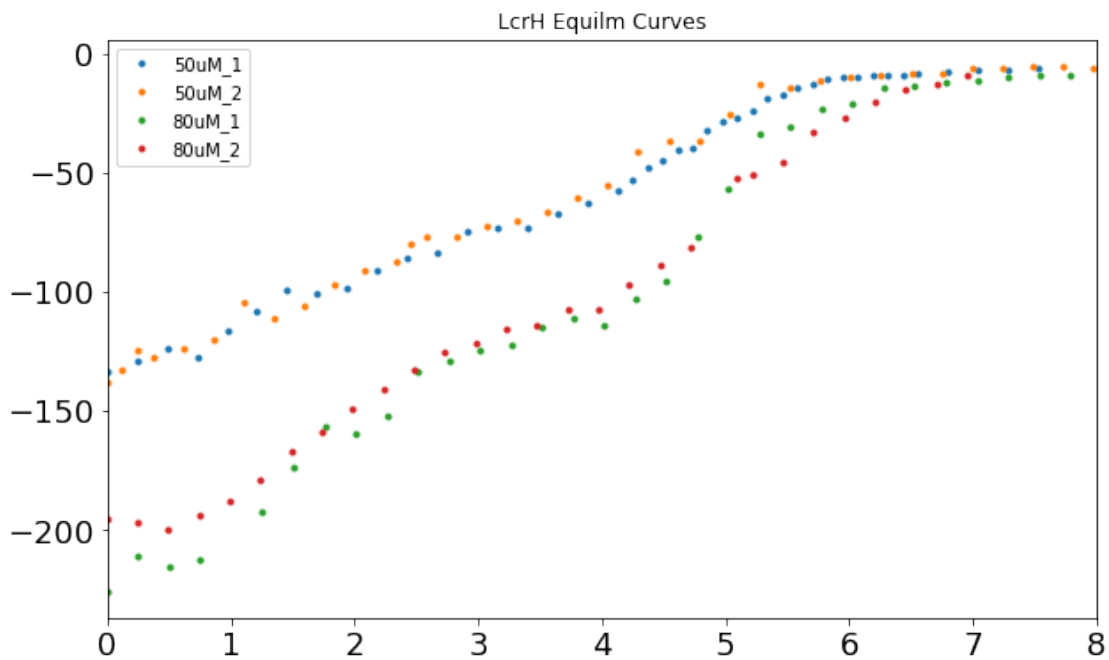
In [4]: # the following commands plot all the data curves on one plot

```

plt.figure(figsize=(10,6))
for c in proteins:
    plt.plot(c.x, c.y, '.')

# the following commands plot all the data curves on one plot,
# where "loc" command determines where the legend goes.
plt.legend([c.ID for c in proteins], loc='best')
plt.title("LcrH Equilm Curves")
plt.xlim([0, 8])          # x axis from 0 to 8
plt.show()

```



```

In [5]: # Command imports pyfolding models
from pyfolding.models import *

# command lists models
list_models()

# After the model name:
# 'Verified: True' model has been rigourously and it functions as expected.
# 'Verified:False' signifies that the model has not been rigourously tested.

```

```

Out [5]: [('ChevronPolynomialFit', 'Verified: True'),
          ('HeteropolymerIsingEquilibrium', 'Verified: False'),
          ('HomozipperIsingEquilibrium', 'Verified: True'),
          ('ParallelTwoStateChevron', 'Verified: False'),

```

```
( 'ParallelTwoStateUnfoldingChevron', 'Verified: False'),
( 'TemplateModel', 'Verified: False'),
( 'ThreeStateChevron', 'Verified: True'),
( 'ThreeStateDimericIEquilibrium', 'Verified: True'),
( 'ThreeStateEquilibrium', 'Verified: True'),
( 'ThreeStateFastPhaseChevron', 'Verified: True'),
( 'ThreeStateMonoIEquilibrium', 'Verified: True'),
( 'ThreeStateSequentialChevron', 'Verified: True'),
( 'TwoStateChevron', 'Verified: True'),
( 'TwoStateChevronMovingTransition', 'Verified: True'),
( 'TwoStateDimerEquilibrium', 'Verified: True'),
( 'TwoStateEquilibrium', 'Verified: True'),
( 'TwoStateEquilibriumSloping', 'Verified: True']
```

10.5 Fit Data to 3 state denaturation that unfolds via dimeric I

In [6]: *# We are going to fit this data to 3 state denaturation that unfolds via
a dimeric intermediate (as per the J.B.C. paper)
Lets print the equation & Info*

```
models.ThreeStateDimericIEquilibrium().info()
```

$$Y_{rel} = Y_N F_N + Y_I F_I + Y_D F_D$$

expanded:

$$Y_{rel} = Y_N \cdot \frac{2PtF_D^2}{K_1 K_2} + Y_I \frac{2PtF_D^2}{K_2} + Y_D * (F_D)$$

where:

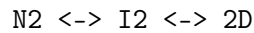
$$F_D = \frac{-K_1 K_2 + \sqrt{((K_1 K_2)^2 + 8(1 + K_1)(K_1 K_2)Pt)}}{4Pt(1 + K_1)} \quad (5)$$

$$K_1 = \exp \frac{-\Delta G_{H_2O}^1 + m_1 x}{RT}$$

$$K_2 = \exp \frac{-\Delta G_{H_2O}^2 + m_2 x}{RT}$$

Three State model for a dimer denaturation Equilibrium - Dimeric Intermediate.

Folding Scheme:



Params:

```
Y_rel = spectroscopic signal at a given concentration of urea
Y_N = spectroscopic signal for native state
Y_D = spectroscopic signal for denatured state
Y_I = spectroscopic signal for intermediate state
F_D = fraction denatured monomers
```



```

F_N = fraction native dimers
F_I = fraction intermediate dimers
Pt = total protein concentration. This variable needs to be set per denaturation
K1 = equilibrium constant of unfolding native to intermediate state
K2 = equilibrium constant of unfolding intermediate to denatured state
DG1 = stability of native state relative to intermediate state
m1 = m-value of native to intermediate transition
DG2 = stability of intermediate state relative to denatured state
m2 = m-value of intermediate to denatured transition
x = denaturant concentration (M)
R = Universal Gas Constant (kcal.mol-1.K-1)
T = Temperature (Kelvin)

```

Reference:

Mallam and Jackson. Folding studies on a knotted protein.
 Journal of Molecular Biology (2005) vol. 346 (5) pp. 1409-1421

10.6 Automatic global fitting to the Three State Dimeric Intermediate Equilibrium model

We could fit individual denaturations to this model. However, one aspect of the model is that it should be able to fit globally to a number of denaturations. We will try to fit this with the Three State Dimeric Intermediate Equilibrium model

```

In [7]: global_fit = core.GlobalFit()
        global_fit.fit_funcs = [models.ThreeStateDimericIEquilibrium
                                for i in xrange(len(proteins))]
        global_fit.constants = [((Pt,c),) for c in Pt]
        global_fit.shared = ['DG1','DG2','m1','m2'] #
        global_fit.x = [p.x for p in proteins]
        global_fit.y = [p.y for p in proteins]
        global_fit.ID = [p.ID for p in proteins]

        global_fit.initialise()

In [8]: print global_fit.params.keys(), len(global_fit.params.keys())

['DG2', 'DG1', 'm1', 'm2', 'Y_N_{50uM_1}', 'Y_I_{50uM_1}', 'Y_D_{50uM_1}', 'Y_N_{50uM_2}', 'Y_I_

In [9]: # this commands gives our initial parameters for fitting
        p0 = [5.,1.7,1.,1.7,
              -200,-50,-10,

```

```

-200,-50,-10,
-200,-50,-10,
-200,-100,-10]

# this command gives fitting constraints
b = bounds=((0.,0.,-3.,-3.,
             -300,-300,-300,
             -300,-300,-300,
             -300,-300,-300,
             -300,-300,-300),
            (20.,20.,3.,3.,
             0,0,0,
             0,0,0,
             0,0,0,
             0,0,0))

# this commands checks that our inputs for the initial parameters and
# constraints (bounds) above
# match the number of our variables
print len(b[0]), len(b[1]), len(p0)

#this commands tell pyfolding to fit our data
out, covar = global_fit.fit( p0=p0, bounds=b )

# print out the results of the fit
for result in global_fit.results:
    result.display()

```

16 16 16

```

=====
Fitting results
=====
ID: 50uM_1
Model: ThreeStateDimericIEquilibrium
Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit
Temperature: 25.00°C

(s) DG1      1.66962 ± 0.72748          95% CI[   1.48549,    1.85376]
(s) m1       1.07181 ± 0.44714          95% CI[   0.95863,    1.18498]
(s) DG2     13.91139 ± 2.41006          95% CI[  13.30137,   14.52140]
(s) m2       1.71687 ± 0.45270          95% CI[   1.60228,    1.83145]
(f) Y_N    -133.91534 ± 10.10650        95% CI[-136.47341, -131.35727]
(f) Y_I     -75.49283 ± 11.70917        95% CI[ -78.45656,  -72.52911]
(f) Y_D     -6.57639 ± 3.97871          95% CI[  -7.58345,   -5.56934]
(c) Pt       0.00005

-----
R^2: 0.99661
=====

```

=====

Fitting results

=====

ID: 50uM_2

Model: ThreeStateDimericIEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(s) DG1	1.66962 ± 0.72748	95% CI[1.48549,	1.85376]
(s) m1	1.07181 ± 0.44714	95% CI[0.95863,	1.18498]
(s) DG2	13.91139 ± 2.41006	95% CI[13.30137,	14.52140]
(s) m2	1.71687 ± 0.45270	95% CI[1.60228,	1.83145]
(f) Y_N	-135.06375 ± 9.56663	95% CI[-137.48517,	-132.64233]	
(f) Y_I	-72.07005 ± 11.55312	95% CI[-74.99427,	-69.14582]	
(f) Y_D	-5.47739 ± 4.07509	95% CI[-6.50884,	-4.44594]	
(c) Pt	0.00005			

R^2: 0.99375

=====

=====

Fitting results

=====

ID: 80uM_1

Model: ThreeStateDimericIEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

Temperature: 25.00°C

(s) DG1	1.66962 ± 0.72748	95% CI[1.48549,	1.85376]
(s) m1	1.07181 ± 0.44714	95% CI[0.95863,	1.18498]
(s) DG2	13.91139 ± 2.41006	95% CI[13.30137,	14.52140]
(s) m2	1.71687 ± 0.45270	95% CI[1.60228,	1.83145]
(f) Y_N	-226.85519 ± 13.79841	95% CI[-230.34773,	-223.36265]	
(f) Y_I	-126.94931 ± 16.17772	95% CI[-131.04407,	-122.85454]	
(f) Y_D	-10.23378 ± 4.60544	95% CI[-11.39946,	-9.06809]	
(c) Pt	0.00008			

R^2: 0.99589

=====

=====

Fitting results

=====

ID: 80uM_2

Model: ThreeStateDimericIEquilibrium

Optimiser: pyfolding.GlobalFit and scipy.optimize.curve_fit

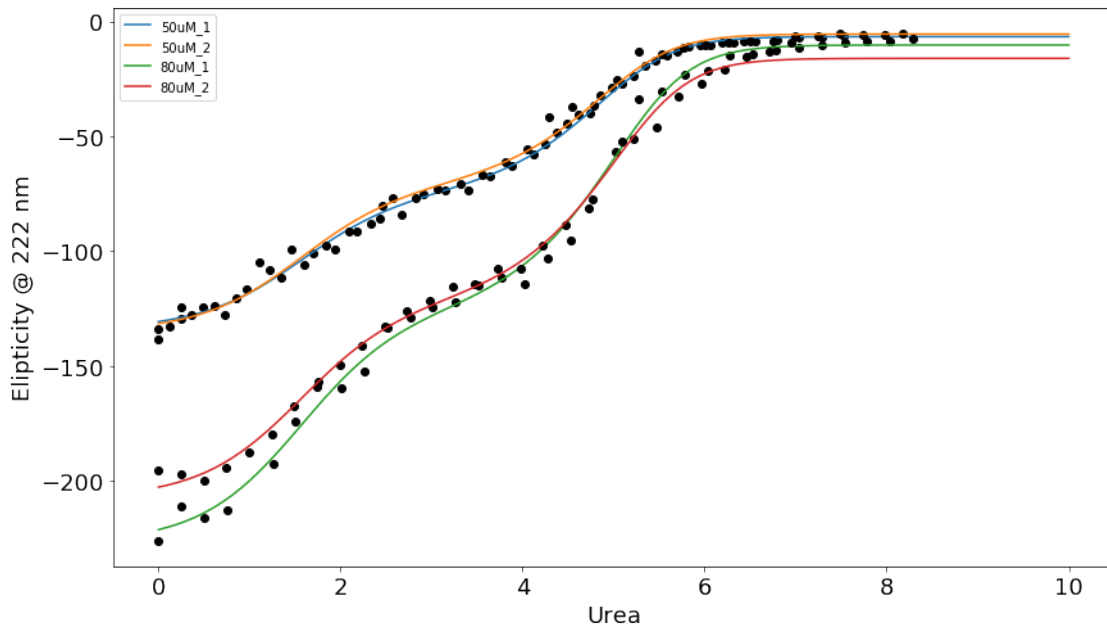
Temperature: 25.00°C

(s) DG1	1.66962 ± 0.72748	95% CI[1.48549,	1.85376]
(s) m1	1.07181 ± 0.44714	95% CI[0.95863,	1.18498]
(s) DG2	13.91139 ± 2.41006	95% CI[13.30137,	14.52140]
(s) m2	1.71687 ± 0.45270	95% CI[1.60228,	1.83145]
(f) Y_N	-207.47639 ± 12.61417	95% CI[-210.66918,	-204.28360]	
(f) Y_I	-122.90586 ± 14.63107	95% CI[-126.60915,	-119.20257]	
(f) Y_D	-16.04299 ± 6.40061	95% CI[-17.66305,	-14.42292]	
(c) Pt	0.00008			

R^2: 0.99668
=====

10.7 Plot the fitted data

```
In [10]: results = global_fit.results
plt.figure(figsize=(14,8))
for i, p in enumerate(proteins):
    plt.plot(p.x, p.y, 'ko')
    plt.plot(results[i].x_fit, results[i].y_fit, '-', label=p.ID)
plt.legend()
plt.xlabel(p.denaturant_label, fontsize=constants.FONT_SIZE)
plt.ylabel('Elipcticity @ 222 nm', fontsize=constants.FONT_SIZE)
plt.show()
```



10.8 Simulate Data

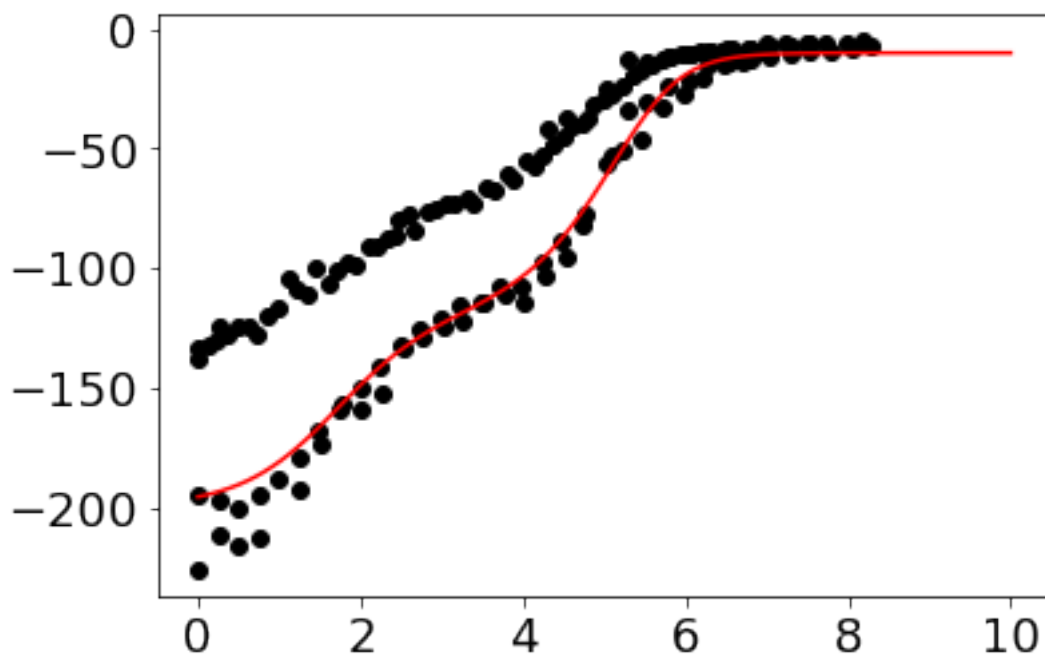
This is also a good way to work out what initial parameters might work well

```
In [12]: #Lets first choose some values for simulation:
DG1 = 1.7
DG2 = 14.
m1 = 1.
m2 = 1.7
Y_N = -200
Y_I = -120
Y_D = -10
Pt = 80e-6

# Then lets simulate what curve we would obtain with these values:
x = np.linspace(0.,10.,100)
m = models.ThreeStateDimericIEquilibrium()
y = m(x, DG1, m1, DG2, m2, Y_N, Y_I, Y_D, Pt)

#Then lets plot the simulation against the data we have:
plt.figure()
for i, p in enumerate(proteins):
    plt.plot(p.x, p.y, 'ko')
plt.plot(x,y,'r-')
```

```
plt.show()  
print y[0]
```



-195.679350161

End of this Notebook.