kinetics Documentation

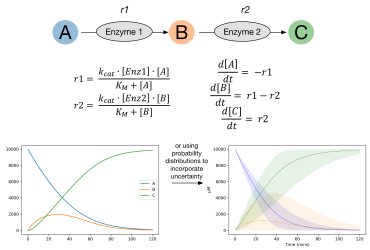
Release 1.4.1

William Finnigan

Dec 15, 2021

Contents

kinetics - a open-source python package for modelling multi-enzyme reactions with uncertainty



kinetics is a package for modelling reactions using ordinary differential equations. It's primarily aimed at modelling enzyme reactions, although can be used for other purposes.

kinetics uses scipy.integrate.odeint to solve ordinary differential equations, but extends upon this to allow the use of parameter distributions rather than single parameter values. This allows error to be incorporated into the modelling.

kinetics uses scipy's probability distributions, with a large number of distributions to choose from. Typically uniform , normal, log-uniform or log-normal distributions are used.

Documentation: ReadTheDocs

Github: kinetics

Requirements: NumPy, SciPy, matplotlib, tqdm, pandas, SALib, seaborn, and deap.

Installation: `pip install kinetics`

Citation: Finnigan, W., Cutlan, R., Snajdrova, R., Adams, J., Littlechild, J. and Harmer, N. (2019), Engineering a seven enzyme biotransformation using mathematical modelling and characterized enzyme parts. ChemCatChem.

CHAPTER 1

Features

- Construct systems of ODEs simply by selecting suitable rate equations and naming parameters and species.
- Use either simple parameter values or probability distributions.
- Run sensitivity analysis using SALib
- Easily plot model runs using predefined plotting functions
- Optimisation using genetic algorithm using DEAP (coming soon)

1.1 Installation

It is highly recommended to use a distribution such as anaconda, which already contains many of the required packages.

1.1.1 Installing kinetics

To install the latest stable version of kinetics using pip, together with all the dependencies, run the following command:

```
pip install kinetics
```

1.1.2 Running in google colab

An easy way to get started quickly is to use a google colab. notebook.

In the first cell of the notebook, run `!pip install kinetics` to install the kinetics package.

Try this block of code in a google colab. notebook to get started quickly..

```
!pip install kinetics
import kinetics

# Define reactions
enzyme_1 = kinetics.Uni(kcat='enz1_kcat', kma='enz1_km', enz='enz_1', a='A',
                        substrates=['A'], products=['B'])

enzyme_1.parameters = {'enz1_kcat' : 100,
                       'enz1_km' : 8000}

# Set up the model
model = kinetics.Model(logging=False)
model.append(enzyme_1)
model.set_time(0, 120, 1000) # 120 mins, 1000 timepoints.

# Set starting concentrations
model.species = {"A" : 10000,
                 "enz_1" : 4,}
model.setup_model()

# Run the model
model.run_model()
model.plot_substrate('A')
model.plot_substrate('B', plot=True)
```

1.1.3 Prerequisite Software

You shouldn't need to worry about this if using the **anaconda** python distribution, and `pip install kinetics`.

kinetics requires NumPy, SciPy, matplotlib, tqdm, pandas, SALib, seaborn, and deap,installed on your computer. Using pip, these libraries can be installed with the following command:

```
pip install numpy
pip install scipy
pip install matplotlib
pip install tqdm
pip install pandas
pip install salib
pip install deap
pip install seaborn
```

The packages are normally included with most Python bundles, such as Anaconda. In any case, they are installed automatically when using pip or setuptools to install kinetics.

1.2 Why, What and How to build models

1.2.1 What is a model?

In the cases of the deterministic kinetic models this package deals with, a model is a set of ordinary differential equations (ODEs), which describe changes in substrate concentrations over time. Essentially a model is answering the question: "how fast will my reaction go?"

To define the rate at which substrates concentrations change over time, we define rate laws.

The most well known of these is the Michaelis-Menton equation. A–enz–>B Our rate law would be:

$$rate = \frac{c_{enz} \cdot k_{cat} \cdot c_A}{c_A + K_M^A}$$

And our Ordinary Differential Equations (ODEs) are:

$$\frac{dA}{dt} = -rate$$

$$\frac{dB}{dt} = +rate$$

Keep in mind that using Michaelis-Menton kinetics we are making the steady-state assumption, among others. For more on this see: Reaction Chemical Kinetics in Biology, N. J. Harmer, M. Vivoli Vega, in Biomol. Bioanal. Tech. (Ed.: V. Ramesh), 2019, pp. 179–217

1.2.2 Why build a model?

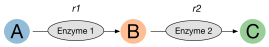
A model is simply a mathematical description for what an experimenter 'thinks' will happen in a reaction. For single enzymes, relatively simple mental or back-of-the-envelope models are often sufficient. For example, it is fairly obvious in the majority of cases that more enzyme results in a faster reaction. However, as we begin to build more complicated multi-enzyme reactions, the increasing complexity of these systems requires a more methodical approach. Constructing a kinetic model allows the dynamics of such a system to be investigated in silico, and to ask questions such as, more of which enzyme gives a faster reaction?

1.2.3 How to build a model?

The simple and advanced tutorials in this documentation deal with how to use the kinetics package to build models of reactions.

However, before writing the code we need to understand which reactions we need to model, and the mechanisms behind them.

A good first step is to draw a reaction scheme such as:



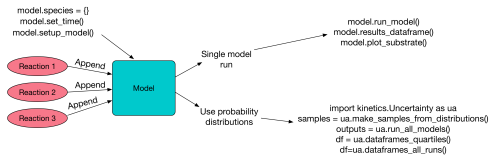
From this (albeit simple) example, we can see we need rate equations for enzyme 1 and enzyme 2.

The next step is to identify the mechanism of these enzymes. In this case both enymes take a single substrate, so can be modelled using either the uni-irreversible or the uniuni-reversible rate equations (see section on reactions).

Where multiple substrates are present (as is common), more options are available to pick from. Literature is a good place to find the appropriate mechanism for you enzyme, from which you can identify which rate equation to use.

Having found the appropriate rate equations, follow either the simple or advanced tutorials to model your system using this package.

1.2.4 Conceptual diagram for how this package works



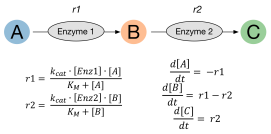
1.3 Simple Tutorial

I recommend running this using a jupyter notebook.

Alternatively use a google colab. notebook to get started quickly. (Use `!pip install kinetics` in the first cell)

Or here is a direct link to this workbook in google colab. https://colab.research.google.com/github/willfinnigan/kinetics/blob/master/examples/Simple_example.ipynb

This example shows the code required to model the relatively simple system shown on the main page. The complete example as a single block of code is available at the end.



1.3.1 Complete code example

Below is the complete code for this example. Copy it into a jupyter notebook or google colab notebook to get started. This example is then explained step by step in the rest of this tutorial.

```
# Uncomment and run this if using google colab
# !pip install kinetics

import kinetics
import matplotlib.pyplot as plt
%config InlineBackend.figure_format ='retina'

# Define reactions
enzyme_1 = kinetics.Uni(kcat='enz1_kcat', kma='enz1_km', enz='enz_1', a='A',
                        substrates=['A'], products=['B'])

enzyme_1.parameters = {'enz1_kcat' : 100,
                       'enz1_km' : 8000}

enzyme_2 = kinetics.Uni(kcat='enz2_kcat', kma='enz2_km', enz='enz_2', a='B',
                        substrates=['B'], products=['C'])

enzyme_2.parameters = {'enz2_kcat' : 30,
                       'enz2_km' : 2000}

# Set up the model
model = kinetics.Model(logging=False)
```

(continues on next page)

(continued from previous page)

```
model.append(enzyme_1)
model.append(enzyme_2)
model.set_time(0, 120, 1000) # 120 mins, 1000 timepoints.

# Set starting concentrations
model.species = {"A" : 10000,
                 "enz_1" : 4,
                 "enz_2" : 10}
model.setup_model()

# Run the model
model.run_model()
model.plot_substrate('A')
model.plot_substrate('B')
model.plot_substrate('C', plot=True)

# Now try altering the enzyme concentration, km or kcat, and re-running the model to␣
↪see the effects this has....
```

1.3.2 Define reactions

The first step in building a kinetic model using this package is to define rate equations which describe the rates of the reactions in a system. Rate equations will typically contain rate constants, referred to as parameters here, and species concentrations. Each rate equation which needs to be included in the model should be set up as a reaction object, as shown below.

In this example, a pair of single substrate irreversible enzyme reactions which follow Michaelis-Menton kinetics are shown:

```
import kinetics

enzyme_1 = kinetics.Uni(kcat='enz1_kcat', kma='enz1_km',
                        enz='enz_1', a='A',
                        substrates=['A'], products=['B'])

enzyme_1.parameters = {'enz1_kcat' : 100,
                       'enz1_km' : 8000}

enzyme_2 = kinetics.Uni(kcat='enz2_kcat', kma='enz2_km',
                        enz='enz_2', a='B',
                        substrates=['B'], products=['C'])

enzyme_2.parameters = {'enz2_kcat' : 30,
                       'enz2_km' : 2000}
```

When setting up reactions, parameter names (`kcat='enz1_kcat', kma='enz1_km'`) and species names (`enz='enz_1', a='A',`) are first given. These are used by the reaction when calculating the rate. A list of substrates which are used up in the reaction, and products that are made in the reaction is the specified (`substrates=['A'], products=['B']`).

Most of the common enzyme mechanisms are pre-defined in the package. For more information see Reactions (link).

Next we need to add these reactions to a model.

1.3.3 Define the model

The model class is central to the kinetics package. It is essentially a list to which we append our reactions, with some extra variables and functions to run the model.

```
import kinetics

# Initiate a new model
model = kinetics.Model()
```

Add reactions to the model

The reactions we defined above are appended to the model.

```
# Append our reactions.
model.append(enzyme_1)
model.append(enzyme_2)
```

Set how long the model will simulate

The time that a model will simulate can be set by using:

```
# Set the model to run from 0 to 120 minutes, over 1000 steps
model.set_time(0, 120, 1000)
```

Set starting species concentrations

The model defaults all starting species to 0, for anything defined in the reactions. For the model to predict anything useful, we need to give it starting concentrations (including the enzymes). Only species which are greater than 0 need to be defined here.

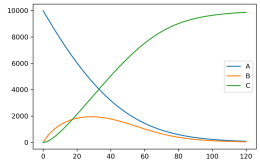
```
# Set starting concentrations
model.species = {"A" : 10000,
                 "enz_1" : 4,
                 "enz_2" : 10}
```

Run the model

Once everything is set, run `model_one.setup_model()` followed by `model_one.run_model()`. A dataframe containing the simulation results is then available using `model_one.results_dataframe()`. Alternatively, results can plotted directly using an in-built plot function `model_one.plot_substrate('A')`.

```
# Setup and run the model
model.setup_model()
model.run_model()

# Plot the results
model.plot_substrate('A')
model.plot_substrate('B')
model.plot_substrate('C', plot=True)
```

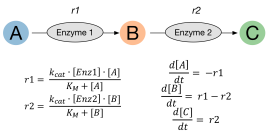


1.4 Advanced Tutorial

This tutorial uses the same example as the simple tutorial, but demonstrates the use of probability distributions rather than single parameter values.

Again I recommend running this using a jupyter notebook or google colab.

Here is a direct link to this example in google colab. Or here is a direct link to this workbook in google colab. https://colab.research.google.com/github/willfinnigan/kinetics/blob/master/examples/Advanced_example.ipynb



1.4.1 Define reactions with uncertain parameters

We can describe the uncertainty we have for a parameter using a probability distribution.

Where parameters have been characterised, resulting in a standard error, we can use this to describe a normal distribution.

Where we roughly know where a parameter is, but don't want to make any suggestion as to more or less likley values, we can use a uniform distribution.

Where we have absolutely no idea what value a parameter takes, we can use a log-uniform distribution (reciprocal) to describe it was being equally as likely to be within a number of orders of magnitude.

Log-normal distributions can also be used. A recent paper describes how log-normal distributions can be created taking into account numerous literature values.

To use these probability distributions in our modelling, we can employ the probability distributions available through scipy.

Define our reactions as before, but this time we specify reaction.parameter_distributions. Make sure to import these from scipy.

```
import kinetics
from scipy.stats import reciprocal, uniform, norm

# Define reactions
enzyme_1 = kinetics.Uni(kcat='enz1_kcat', kma='enz1_km', enz='enz_1', a='A',
                        substrates=['A'], products=['B'])

enzyme_1.parameter_distributions = {'enz1_kcat' : norm(100,12),
                                    'enz1_km' : uniform(2000, 6000)}

enzyme_2 = kinetics.Uni(kcat='enz2_kcat', kma='enz2_km', enz='enz_2', a='B',
```

(continues on next page)

(continued from previous page)

```
                          substrates=['B'], products=['C'])

enzyme_2.parameter_distributions = {'enz2_kcat' : norm(30, 5),
                                    'enz2_km' : reciprocal(1,10000)}
```

Next we define our model as before.

```
# Set up the model
model = kinetics.Model()
model.append(enzyme_1)
model.append(enzyme_2)
model.set_time(0, 120, 1000)
```

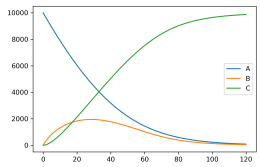
We can include uncertainty in some or all of the starting species concentrations. Here we have specified uncertainty in the enzyme concentraions using a normal distribution with a standard deviation of 5% of the starting value. We have specified no uncertainty in the starting concentration of A.

```
# Set starting concentrations
model.species = {"A" : 10000}
model.species_distributions = {"enz_1" : norm(4, 4*0.05),
                               "enz_2" : norm(10, 10*0.05)}
model.setup_model()
```

1.4.2 Running the model with a single set of parameter values

We can run the model exactly as in the simple example, and we will get a single prediction for each substrate. Running the model this way will use the mean of each probability distribution specified, unless a different value is specified.

```
model.run_model()
model.plot_substrate('A')
model.plot_substrate('B')
model.plot_substrate('C', plot=True)
```



1.4.3 Running the model by sampling within the probability distributions

However we would like to run lots of models, sampling within our probability distributions.

To generate samples from within the distributions we have defined, run `kinetics.sample_distributions(model, num_samples=1000)`. This returns a set of samples which can be used by `kinetics.run_all_models(model, samples)`.

`kinetics.run_all_models(model, samples)` will return a list of outputs. Each entry in this list is equivalent to `model.y` after running `model.run_model()`.

```
# Run the model 1000 times, sampling from distributions
samples = kinetics.sample_distributions(model, num_samples=1000)
outputs = kinetics.run_all_models(model, samples, logging=True)
```

1.4.4 Plotting the data

To deal with the large amount of data this generates, two functions are available to generate a dictionary containing dataframes for each species in the model.

`dataframes_all_runs(model, output)` will return dataframes containing every single run.

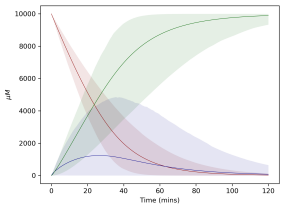
`dataframes_quartiles(model, output, quartile=95)` will return dataframes containing a High, Low and Mean value, based on whatever quartile is specified (default=95%).

These dataframes can then be exported for further use, or can be used to generate plots.

Plotting graphs with confidence intervals

Plotting the 95% confidence intervals can look neater, but we lose some information on the outliers by doing this.

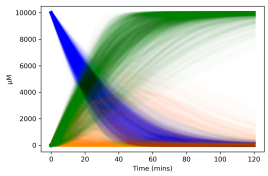
```
# Plot model runs at 95% CI
ci_dataframes = kinetics.dataframes_quartiles(model, outputs)
kinetics.plot_ci_intervals(['A', 'B', 'C'], ci_dataframes, colours=['blue',
↪'darkorange', 'green'], plot=True)
```



Plotting graphs showing all runs (spagetti plots)

Alternatively we can plot every single run. With 1000 runs this can look a bit chaotic, and it may be clearer to plot each substrate on its own graph. Also, altering the alpha and linewidth values allows the graphs to be tweaked to preference.

```
# Plot all model runs
all_runs_dataframes = kinetics.dataframes_all_runs(model, outputs)
kinetics.plot_substrate('A', all_runs_dataframes, colour='blue', alpha=0.01,␣
↪linewidth=5)
kinetics.plot_substrate('B', all_runs_dataframes, colour='darkorange', alpha=0.01,␣
↪linewidth=5)
kinetics.plot_substrate('C', all_runs_dataframes, colour='green', alpha=0.01,␣
↪linewidth=5, plot=True)
```



Of course the dataframes are also available to be used as the output, possibly to create your own graphs or for other analysis.

1.4.5 Complete code

```
import kinetics
import matplotlib.pyplot as plt
from scipy.stats import reciprocal, uniform, norm
%config InlineBackend.figure_format ='retina'

# Define reactions
enzyme_1 = kinetics.Uni(kcat='enz1_kcat', kma='enz1_km', enz='enz_1', a='A',
                        substrates=['A'], products=['B'])

enzyme_1.parameter_distributions = {'enz1_kcat' : norm(100,12),
                                    'enz1_km' : uniform(2000, 6000)}

enzyme_2 = kinetics.Uni(kcat='enz2_kcat', kma='enz2_km', enz='enz_2', a='B',
                        substrates=['B'], products=['C'])

enzyme_2.parameter_distributions = {'enz2_kcat' : norm(30, 5),
                                    'enz2_km' : reciprocal(1,10000)}

# Set up the model
model = kinetics.Model(logging=False)
model.append(enzyme_1)
model.append(enzyme_2)
model.set_time(0, 120, 1000)

# Set starting concentrations
model.species = {"A" : 10000}
model.species_distributions = {"enz_1" : norm(4, 4*0.05),
                               "enz_2" : norm(10, 10*0.05)}
model.setup_model()

# Run the model 1000 times, sampling from distributions
samples = kinetics.sample_distributions(model, num_samples=1000)
outputs = kinetics.run_all_models(model, samples, logging=True)

model.run_model()
model.plot_substrate('A')
model.plot_substrate('B')
model.plot_substrate('C', plot=True)

# Plot model runs at 95% CI
ci_dataframes = kinetics.dataframes_quartiles(model, outputs)
kinetics.plot_ci_intervals(['A', 'B', 'C'], ci_dataframes, colours=['blue',
↪'darkorange', 'green'])
plt.show()

# Plot all model runs
all_runs_dataframes = kinetics.dataframes_all_runs(model, outputs)
kinetics.plot_substrate('A', all_runs_dataframes, colour='blue', alpha=0.01,␣
↪linewidth=5)
kinetics.plot_substrate('B', all_runs_dataframes, colour='darkorange', alpha=0.01,␣
↪linewidth=5)
kinetics.plot_substrate('C', all_runs_dataframes, colour='green', alpha=0.01,␣
↪linewidth=5)
plt.show()
```

1.5 Reactions

kinetics works by specifying a `Model()` object to which reactions are added.

Reactions are first defined, selected from one of the reaction objects described here. Parameters are set and the reaction added to the model.

For example:

```
import kinetics

enzyme_1 = kinetics.Uni(kcat='kcat1', kma='kma1', a='a', enz='enz1',
                        substrates=['a'], productions=['b'])

# To specify single parameter values
enzyme_1.parameters = {'kcat1': 100,
                       'kma1': 500}

# To specify parameter distributions
enzyme_1.parameter_distributions = {'kcat1': norm(100,10),
                                    'kma1': uniform(25,50)}

model = kinetics.Model()
model.append(enzyme1)
```

1.5.1 Michaelis-Menten kinetics, irreversible.

Uni

class kinetics.**Uni**(*kcat=None*, *kma=None*, *a=None*, *enz=None*, *substrates=[]*, *products=[]*)

The classic Miachelis-Menton equation for a single substrate.

$$rate = \frac{c_{enz} \cdot k_{cat} \cdot c_A}{c_A + K_M^A}$$

Bi

class kinetics.**Bi**(*kcat=None*, *kma=None*, *kmb=None*, *a=None*, *b=None*, *enz=None*, *substrates=[]*, *products=[]*)

Not strictly a true Miachaelis-Menton equation. Use with caution. Will give a reasonable prediction if one substrate is saturating, otherwise is likely wrong.

$$rate = c_{enz} \cdot k_{cat} \cdot \frac{c_A}{c_A + K_M^A} \cdot \frac{c_B}{c_B + K_M^B}$$

Bi Ternary Complex

class kinetics.**Bi_ternary_complex**(*kcat=None*, *kma=None*, *kmb=None*, *kia=None*, *a=None*, *b=None*, *enz=None*, *substrates=[]*, *products=[]*)

For reactions with two substrates which have an sequential mechanism (either ordered or random).

$$rate = \frac{c_{enz} \cdot k_{cat} \cdot c_A \cdot c_B}{(K_I^A \cdot K_M^B) + (K_M^B \cdot c_A) + (K_M^A \cdot c_B) + (c_A \cdot c_B)}$$

Bi Ping Pong

class kinetics.**Bi_ping_pong**(*kcat=None*, *kma=None*, *kmb=None*, *a=None*, *b=None*, *enz=None*, *substrates=[]*, *products=[]*)

For reactions with two substrates which have a ping-pong mechanism

$$rate = \frac{c_{enz} \cdot k_{cat} \cdot c_A \cdot c_B}{(K_M^B \cdot c_A) + (K_M^A \cdot c_B) + (c_A \cdot c_B)}$$

Ter seq redam

class kinetics.**Ter_seq_redam**(*kcat=None*, *kma=None*, *kmb=None*, *kmc=None*, *kia=None*, *kib=None*, *enz=None*, *a=None*, *b=None*, *c=None*, *substrates=[]*, *products=[]*)

A three substrate rate equation which can be used for Reductive Aminase enzymes.

$$rate = \frac{c_{enz} \cdot k_{cat} \cdot c_A \cdot c_B \cdot c_C}{(K_I^A \cdot K_I^B \cdot K_M^C) + (K_I^B \cdot K_M^C \cdot c_A) + (K_I^A \cdot K_M^B \cdot c_C) + (K_M^C \cdot c_A \cdot c_B) + (K_M^B \cdot c_A \cdot c_C) + (K_M^A \cdot c_B \cdot c_C) + (c_A \cdot c_B \cdot c_C)}$$

Ter seq car

class kinetics.**Ter_seq_car**(*kcat=None*, *kma=None*, *kmb=None*, *kmc=None*, *kia=None*, *enz=None*, *a=None*, *b=None*, *c=None*, *substrates=[]*, *products=[]*)

A three substrate rate equation which can be used for Carboxylic Acid Reductase enzymes.

$$rate = \frac{c_{enz} \cdot k_{cat} \cdot c_A \cdot c_B \cdot c_C}{(K_I^A \cdot c_C) + (K_M^C \cdot c_A \cdot c_B) + (K_M^B \cdot c_A \cdot c_C) + (K_M^A \cdot c_B \cdot c_C) + (c_A \cdot c_B \cdot c_C)}$$

Bi ternary complex small kma

class kinetics.**Bi_ternary_complex_small_kma**(*kcat=None*, *kmb=None*, *kia=None*, *a=None*, *b=None*, *enz=None*, *substrates=[]*, *products=[]*)

A special case of Bi Ternary Complex where kma << kia.

$$rate = \frac{c_{enz} \cdot k_{cat} \cdot c_A \cdot c_B}{(K_I^A \cdot K_M^B) + (K_M^B \cdot c_A) + (c_A \cdot c_B)}$$

1.5.2 Michaelis-Menten kinetics, reversible.

UniUni Reversible

class kinetics.**UniUni_rev**(*kcatf=None*, *kcatr=None*, *kma=None*, *kmp=None*, *a=None*, *p=None*, *enz=None*, *substrates=[]*, *products=[]*)

$$rate = \frac{(c_{enz} \cdot k_{cat}^{fwd} \cdot c_A) - (c_{enz} \cdot k_{cat}^{rev} \cdot c_P)}{1 + \frac{c_A}{K_M^A} + \frac{c_P}{K_M^P}}$$

BiBi Ordered Rev

class kinetics.**BiBi_Ordered_rev**(*kcatf=None*, *kcatr=None*, *kmb=None*, *kia=None*, *kib=None*, *kmp=None*, *kip=None*, *kiq=None*, *enz=None*, *a=None*, *b=None*, *p=None*, *q=None*, *substrates=[]*, *products=[]*)

BiBi Random Rev

class kinetics.**BiBi_Random_rev**(*kcatf=None*, *kcatr=None*, *kmb=None*, *kia=None*, *kib=None*, *kmp=None*, *kip=None*, *kiq=None*, *a=None*, *b=None*, *p=None*, *q=None*, *enz=None*, *substrates=[]*, *products=[]*)

BiBi Pingpong Rev

class kinetics.**BiBi_Pingpong_rev**(*kcatf=None*, *kma=None*, *kmb=None*, *kia=None*, *kcatr=None*, *kmp=None*, *kmq=None*, *kip=None*, *kiq=None*, *enz=None*, *a=None*, *b=None*, *p=None*, *q=None*, *substrates=[]*, *products=[]*)

1.5.3 Equilibrium based reversible Michaelis-Menten kinetics

BiBi Ordered rev eq

class kinetics.**BiBi_Ordered_rev_eq**(*keq=None*, *kcatf=None*, *kcatr=None*, *kma=None*, *kmb=None*, *kmp=None*, *kmq=None*, *kib=None*, *kip=None*, *kia=None*, *a=”*, *b=”*, *p=”*, *q=”*, *enz=”*, *substrates=[]*, *products=[]*)

UniUni Ordered rev eq

class kinetics.**UniUni_rev_eq**(*keq=None*, *kcatf=None*, *kma=None*, *kmp=None*, *a=”*, *p=”*, *enz=”*, *substrates=[]*, *products=[]*)

1.5.4 Equilibrium based mass action

UniUni Ordered rev eq

class kinetics.**UniUni_rev_eq**(*keq=None*, *kcatf=None*, *kma=None*, *kmp=None*, *a=”*, *p=”*, *enz=”*, *substrates=[]*, *products=[]*)

1.5.5 Modifiers of Michaelis-Menten kinetics eg for Inhibition

Modifications to rate equations for things like competitive inhibition can applied as follows:

(Remember to add new parameters to the reaction parameters)

Modifications are applied at each timestep of the model, for example calculating the apparent Km resulting from competitive inhibtion.

This feature allows the easy modification of the pre-defined rate equations.

```
enzyme_1.add_modifier(kinetics.CompetitiveInhibition(km='kma1', ki='ki1', i='I'))
enzyme_1.parameters.update({'ki1': 25})
```

class kinetics.**SubstrateInhibition**(*ki=None*, *a=None*)

class kinetics.**CompetitiveInhibition**(*km=None*, *ki=None*, *i=None*)

class kinetics.**MixedInhibition**(*kcat=None*, *km=None*, *ki=None*, *alpha=None*, *i=None*)

class kinetics.**FirstOrder_Modifier**(*kcat=None*, *k=None*, *s=None*)

1.5.6 Generic Reaction Class

This reaction class could in theory be the only one you ever need. It allows you to specify your own rate equation.

class kinetics.**Generic**(*params=[]*, *species=[]*, *rate_equation=''*, *substrates=[]*, *products=[]*)
This Reaction class allows you to specify your own rate equation. Enter the parameter names in params, and the substrate names used in the reaction in species. Type the rate equation as a string in rate_equation, using these same names. Enter the substrates used up, and the products made in the reaction as normal.

1.6 Custom Reactions

It's not possible to pre-define every rate equation anyone would ever need. However many of the most common rate equations are already set up (see section on Reactions).

There are two options for custom rate equations.

1.6.1 1. Use the Generic Reaction Class

This is a reaction class which lets you specify your own rate equation.

class kinetics.**Generic**(*params=[]*, *species=[]*, *rate_equation=''*, *substrates=[]*, *products=[]*)
This Reaction class allows you to specify your own rate equation. Enter the parameter names in params, and the substrate names used in the reaction in species. Type the rate equation as a string in rate_equation, using these same names. Enter the substrates used up, and the products made in the reaction as normal.

An example which models a UniUni enzyme

```
import kinetics
import kinetics.Uncertainty as ua
import matplotlib.pyplot as plt
from scipy.stats import norm

step1 = kinetics.Generic(params=['k1', 'k_1'], species=['a','e','ea'],
                         rate_equation='(k1*a*e)-(k_1*ea)',
                         substrates=['a', 'e'], products=['ea'])

step2 = kinetics.Generic(params=['k2','k_2'], species=['ea','e','p'],
                         rate_equation='(k2*ea)-(k_2*ea*p)',
                         substrates=['ea'], products=['e', 'p'])

step1.parameters = {'k1' : 0.1,
                    'k_1' : 0.001}
```

(continues on next page)

(continued from previous page)

```
step2.parameters = {'k2' : 100,
                    'k_2': 0.1}

step1.parameter_distributions = {'k1' : norm(0.1, 0.01),
                                 'k_1' : norm(0.001, 0.0001)}

step2.parameter_distributions = {'k2' : norm(100, 10),
                                 'k_2' : norm(0.1, 0.01)}

model = kinetics.Model()

model.append(step1)
model.append(step2)
model.species = {'e' : 1,
                 'a' : 100}

model.setup_model()

model.run_model()
model.plot_substrate('a')
model.plot_substrate('e')
model.plot_substrate('ea')
model.plot_substrate('p')
plt.show()

samples = ua.make_samples_from_distributions(model, num_samples=1000)
outputs = ua.run_all_models(model, samples, logging=True)
all_runs_dataframes = ua.dataframes_all_runs(model, outputs)
ua.plot_substrate('a', all_runs_dataframes, colour='blue', alpha=0.01, linewidth=5)
ua.plot_substrate('e', all_runs_dataframes, colour='darkorange', alpha=0.01,␣
↪linewidth=5)
ua.plot_substrate('ea', all_runs_dataframes, colour='green', alpha=0.01, linewidth=5)
ua.plot_substrate('p', all_runs_dataframes, colour='purple', alpha=0.01, linewidth=5)
plt.show()
```

1.6.2 2. Make your own reaction class.

This might be useful if its going to be re-used alot

To make a reaction class for a custom rate equation we need to define a new class which inherits from `kinetics.Reaction`

The new class needs two funcions. an __init__() function and a calculate_rate() function.

The following code example provides an example for doing this:

```
class My_New_Reaction(kinetics.Reaction):

    def __init__(self,
                 param1='', param2='', species1='', species2='',
                 substrates=[], products=[]):

    # This is required to inherit from kinetics.Reaction
    super().__init__()

    # Set parameter and substrates names from the arguments passed in.  The order is␣
↪important here.
```

(continues on next page)

(continued from previous page)

```
    self.parameter_names=[param1, param2]
    self.reaction_substrate_names = [species1, species2]

    # Set the substrates and products from the arguments passed in.
    # Substrates are used up in the reaction, while produces are generated.
    self.substrates = substrates
    self.products = products

    def calculate_rate(self, substrates, parameters):

        # This function is used to calculate the rate at each time step in the model
        # It takes substrates and parameters as arguments, which are lists with the␣
↪same order as we defined in __init__.

        # Substrates
        species1 = substrates[0]
        species2 = substrates[1]

        # Parameters
        param1 = parameters[0]
        param2 = parameters[1]

        # This is where the rate equation goes.  An example is shown.
        rate = param1*species1 + param2*species2

        return rate
```

1.7 API

1.8 Authors

Will Finnigan - william.finnigan@manchester.ac.uk

1.9 Change Log

1.3.6 - Refactored Uncertainty.make_samples_from_distributions(..) to Uncertainty.sample_distributions(..) - Added Uncertainty.sample_uniforms - Added test_simple_model - Added MixedInhibition2, which takes kic and kiu - Refactored Senstivity module into Uncertainty - Added check to model, if no parameter is set in either reactions or model, take mean of model.parameter_distribution

1.3.7 - Reorganised code, Uncertainty and Senstivity modules are now imported directly into kinetics. - Changed to 'from setuptools import setup' in setup.py - Changed docs to reflect changine in Uncertainty module import

CHAPTER 2

Support

wjafinnigan@gmail.com or william.finnigan@manchester.ac.uk

CHAPTER 3

License

The project is licensed under the MIT license.

Index