PyscesToolbox Documentation

Release 1.0.0

Carl Christensen and Johann Rohwer

Aug 19, 2020

Contents

Contents:

CHAPTER 1

Introduction

PySCeSToolbox is a set of extensions to the original Python Simulator for Cellular Systems (PySCeS) [1]. The goals of this software are **(1)** to provide metabolic model analysis tools that are beyond the scope of PySCeS and **(2)** to provide a streamlined framework for using these tools together. The reader is referred to the *Bioinformatics* paper [2] for further details.

Currently, PySCeSToolbox includes three main analysis tools:

1. SymCa for performing symbolic control analysis [3,4].
2. RateChar for performing generalised supply demand analysis [5,6].
3. ThermoKin for distinguishing between the thermodynamic and kinetic contributions towards reaction rates and enzyme elasticities [7,8].

In addition to these tools PySCeSToolbox provides functionality for displaying interactive plots, tables of results, and typeset mathematical expressions and symbols by making extensive use of the wonderful Jupyter (IPython) Notebook platform. Therefore, in order to make the best use of its features we recommend that users run PySCeSToolbox within the IPython Notebook environment. Regardless of being designed for interactive work through the Notebook, the core features are completely compatible with traditional python scripting.

We recommend that users unfamiliar with PySCeS refer to its documentation before continuing here.

CHAPTER 2

Installation

PySCeSToolbox is compatible with macOS, Linux, and Windows, and can be installed either with `conda` in an Anaconda environment, or with `pip` in an existing Python environment. We have made special effort to provide as detailed instructions as possible, assuming a clean installation of each operating system prior to installation of PySCeSToolbox, and relatively limited knowledge of Python. If further assistance is required, please contact the developers.

Below follow abbreviated requirements, installation instructions for `conda` and `pip`, as well as operating system-specific instructions for setting up Maxima.

2.1 Abbreviated requirements

PySCeSToolbox has a number of requirements that must be met before installation can take place. Fortunately most requirements, save for a few exceptions (as discussed in the operating system-specific sections), will be taken care of automatically during installation. An abbreviated list of requirements follows:

- A Python 3.x installation (Python 3.6 or higher is recommended)
- The full SciPy Stack (see http://scipy.org/install.html).
- PySCeS (see http://pysces.sourceforge.net)
- Maxima (see http://maxima.sourceforge.net)
- Jupyter Notebook (jupyter-core version in the 4.x.x series)

2.2 Installation on Anaconda

For most users (especially those unfamiliar with Python) we recommend using the Anaconda Python distribution (https://www.anaconda.com/products/individual#Downloads). This is a low fuss solution available for all three operating systems that will install Python on you system *together with many of the packages necessary for running PySCeSToolbox*. Download the appropriate **Python 3.7** package from the download page (most probably the 64bit edition) and follow the instructions of the installation wizard.

2.2.1 Virtual environments

Virtual environments are a great way to keep package dependencies separate from your system files. It is highly recommended to install PyscesToolbox into a separate environment, which first must be created (here we create an environment called `pysces`). It is recommended to use a Python version >=3.6 (here we use Python 3.7). After creation, activate the environment:

```
(base) $ conda create -n pysces python=3.7
(base) $ conda activate pysces
```

Then install PyscesToolbox:

```
(pysces) $ conda install -c pysces -c sbmlteam pyscestoolbox
```

Be sure to specify the *pysces* and *sbmlteam* channels in the command line as above, otherwise some of the packages won't be found. The required Python dependencies will be installed automatically. For Maxima, refer to the operating system-specific instructions below.

2.2.2 Enabling widgets

If you are running the Jupyter notebook for the first time, or if you have not yet enabled the notebook widgets you may need to run the following command:

```
(pysces) $ jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

We also recommend running the following two commands to enable the ModelGraph functionality of PySCeSToolbox. Rerunning these commands may be necessary when updating/reinstalling PySCeSToolbox.

```
(pysces) $ jupyter nbextension install --py --user d3networkx_psctb
(pysces) $ jupyter nbextension enable --py --user d3networkx_psctb
```

2.3 Alternative: direct `pip`-based install

First be sure to have Python 3 and `pip` installed. Pip is a useful Python package management system.

On Debian and Ubuntu-like Linux systems these can be installed with the following terminal commands:

```
$ sudo apt install python3
$ sudo apt install python3-pip
```

Other Linux distributions will also have Python 3 and `pip` available in their repositories.

On Windows, download Python from https://www.python.org/downloads/windows; be sure to install `pip` as well when prompted by the installer, and add the Python directories to the system PATH. You can verify that the Python paths are set up correctly by checking the `pip` version in a Windows Command Prompt:

```
> pip -V
```

On macOS you can install Python directly from https://www.python.org/downloads/mac-osx, or by installing Homebrew and then installing Python 3 with Homebrew. Both come with `pip` available.

Note: While most Linux distributions come pre-installed with a version of Python 3, the options for Windows and macOS detailed above are more advanced and for experienced users, who prefer fine-grained control. If you are

starting out, we strongly recommend using Anaconda!

2.3.1 Virtual environments

Again it is highly recommended to install PyscesToolbox into a separate virtual environment. There are several options for setting up your working environment. We will use virtualenvwrapper, which works out of the box on Linux and macOS. On Windows, virtualenvwrapper can be used under an MSYS environment in a native Windows Python installation. Alternatively, you can use virtualenvwrapper-win. This will take care of managing your virtual environments by maintaining a separate Python *site-directory* for you.

Install virtualenvwrapper using `pip`. On Linux and MacOS:

```
$ sudo -H pip install virtualenv
$ sudo -H pip install virtualenvwrapper
```

On Windows in a Python command prompt:

```
> pip install virtualenv
> pip install virtualenvwrapper-win
```

Make a new virtual environment for working with PyscesToolbox (e.g. `pysces`), and specify that it use Python 3 (we used Python 3.7):

```
$ mkvirtualenv -p /path/to/your/python3.7 pysces
```

The new virtual environment will be activated automatically, and this will be indicated in the shell prompt, e.g.:

```
(pysces) $
```

If you are not yet familiar with virtual environments we recommend you survey the basic commands (https://virtualenvwrapper.readthedocs.io/en/latest/) before continuing.

The PyscesToolbox code and its dependencies can now be installed directly from PyPI into your virtual environment using `pip`.

```
(pysces) $ pip install pyscestoolbox
```

As for the `conda`-based install, the required Python dependencies will be installed automatically. For Maxima, refer to the operating system-specific instructions below.

2.3.2 Enabling widgets

Refer to the *Anaconda-based install*.

2.4 Maxima

Maxima is necessary for generating control coefficient expressions using SymCA. Below we provide operating-specific instructions for setting up Maxima.

2.4.1 Windows

The latest version of Maxima can be downloaded and installed from the Windows download page at http://maxima.sourceforge.net/download.html.

Windows might also require the path to `maxima.bat` to be defined in the `psctb_config.ini` file, found at `%USERPROFILE%\Pysces\psctb_config.ini` by default, or in `C:\Pysces` for older PySCeS versions.

Note: As of PySCeS version 0.9.8 the default location of configuration and model files moved from `C:\Pysces` to `%USERPROFILE%\Pysces`, i.e. typically `C:\Users\<username>\Pysces`, to bring the Windows installation more in line with the macOS and Linux installations. Refer to the PySCeS 0.9.8 release notes for more information.

The default path included in `psctb_config.ini` is set as `C:\maxima?\bin\maxima.bat`, where the question marks are wildcards (since the specific path will depend on the version of Maxima). If Maxima has been installed to a user specified directory, the correct path to the `maxima.bat` file must be specified here.

2.4.2 macOS (Mac OS X)

The latest version of Maxima can be downloaded and installed from the MacOS download page at http://maxima.sourceforge.net/download.html. We recommend the VTK version of Maxima.

After downloading and installing the Maxima dmg, the following lines must be added to your `.bash_profile` or `.zshrc` file (depending on which shell you use):

```
export M_PREFIX=/Applications/Maxima.app/Contents/Resources/opt
export PYTHONPATH=${M_PREFIX}/Library/Frameworks/Python.framework/Versions/2.7/lib/
↪python2.7/site-packages/:$PYTHONPATH
export MANPATH=${M_PREFIX}/share/man:$MANPATH
export PATH=${M_PREFIX}/bin:$PATH
alias maxima=rmaxima
```

2.4.3 Linux

Maxima can be installed from your repositories, if available, otherwise the latest packages can be downloaded from the Linux link at http://maxima.sourceforge.net/download.html.

CHAPTER 3

Basic Usage

This section gives a quick overview of some features and conventions that are common to all the main analysis tools. While the main analysis tools will be briefly referenced here, later sections will cover them in full.

3.1 Starting a PySCeSToolbox session

To start a PySCeSToolbox session in a Jupyter notebook:

1. Open a terminal in the environment where you installed PyscesToolbox (i.e. Anaconda environment or other Python environment)
2. Start up the Jupyter Notebook using the `jupyter notebook` command in the terminal
3. Create a new notebook by clicking the `New` button on the top right of the window and selecting `Python 3`
4. Run the following three commands in the first cell:

```
import pysces
import psctb
%matplotlib inline
```

3.2 Downloading interactive Jupyter notebooks

To facilitate learning of this software, a set of interactive Jupyter notebooks are provided that mirror the pages for Basic Usage (this page), RateChar, Symca and Thermokin found in this documentation. They can be downloaded from Included Files. The models and associated files should be saved in the `~/Pysces/psc` folder, while the example notebooks can go anywhere.

3.3 Syntax

As PySCeSToolbox was designed to work on top of PySCeS, many of its conventions are employed in this project. The syntax (or naming scheme) for referring to model variables and parameters is the most obvious legacy. Syntax is briefly described in the table below and relates to the provided example model (for input file syntax refer to the PySCeS model descriptor language documentation):

Description	Syntax description	PySCeS example	Rendered LaTeX example
Parameters	As defined in model file	Keq2	$Keq2$
Species	As defined in model file	S1	$S1$
Reactions	As defined in model file	R1	$R1$
Steady state species	"_ss" appended to model definition	S1_ss	$S1_{ss}$
Steady state reaction rates (Flux)	"J_" prepended to model definition	J_R1	J_{R1}
Control coefficients	In the format "ccJreaction_reaction"	ccJR1_R2	C^{JR1}_{R2}
Elasticity coefficients	In the format "ecreaction_modifier"	ecR1_S1 or ecR2_Vf1	ε^{R1}_{S1} or ε^{R2}_{Vf2}
Response coefficients	In the format "rcJreaction_parameter"	rcJR3_Vf3	R^{JR3}_{Vf3}
Partial response coefficients	In the format "prcJreaction_parameter_reaction"	prcJR3_X2_R2	${}^{R2}R^{JR3}_{X\ 2}$
Control patterns	CPn where n is an number assigned to a specific control pattern	CP4	$CP4$
Flux contribution by specific term	In the format "J_reaction_term"	J_R1_binding	$J_{R1_{binding}}$
Elasticity contribution by specific term	In the format "pecreaction_modifier_term"	pecR1_S1_binding	$\varepsilon^{R1\ binding}_{S1}$

Note: Any underscores (_) in model defined variables or parameters will be removed when rendering to LaTeX to ensure consistency.

3.4 Saving and Default Directories

Whenever any analysis tool is used for the first time on a specific model, a directory is created within the PySCeS output directory that corresponds to the model name. A second directory which corresponds to the analysis tool name will be created within the first. These directories serve a dual purpose:

The fist, and most pertinent to the user, is for providing a default location for saving results. PySCeSToolbox allows users to save results to any arbitrary location on the file system, however when no location is provided, results will be saved to the default directory corresponding to the model name and analysis method as described above. We consider this a fairly intuitive and convenient system that is especially useful for outputting small sets of results. Result saving functionality is usually provided by a `save_results` method for each respective analysis tool. Exceptions are `RateChar` where multiple types of results may be saved, each with their own method, and `ScanFig` where figures are saved simply with a `save` method.

The second purpose is to provide a location for writing temporary files and internal data that is used to save "analysis sessions" for later loading. In this case specifying the output destination is not supported in most cases and these features depend on the default directory. Session saving functionality is provided only for tools that take significant amounts of time to generate results and will always be provided by a `save_session` method and a corresponding `load_session` method will read these results from disk.

Note: Depending on your OS the default PySCeS directory will be either `~/Pysces` or `C:\Pysces` (on Windows with PySCeS versions up to 0.9.7) or `C:\Users\<username>\Pysces` (on Windows with PySCeS version 0.9.8+). PySCeSToolbox will therefore create the following type of folder structure: `~/Pysces/model_name/analysis_method/` or `C:\Pysces\model_name\analysis_method\` or `C:\Users\<username>\Pysces\model_name\analysis_method\` depending on your configuration.

3.5 Plotting and Displaying Results

As already mentioned previously, PySCeSToolbox includes the functionality to plot results generated by its tools. Typically these plots will either contain results from a parameter scan where some metabolic variables are plotted against a change in parameter, or they will contain results from a time simulation where the evolution of metabolic variables over a certain time period are plotted.

3.5.1 Data2D

The `Data2D` class provides functionality for capturing raw parameter scan/simulation results and provides an interface to the actual plotting tool `ScanFig`. It is used internally by other tools in PySCeSToolbox and a `Data2D` object will be created and returned automatically after performing a parameter scan with any of the `do_par_scan` methods provided by these tools.

Features

- Access to scan/simulation results through its `scan_results` dictionary.
- The ability to save results in the form of a `csv` file using the `save_results` method.
- The ability to generate a `ScanFig` object via the `plot` method.

Usage example

Below is an usage example of Data2D, where results from a PySCeS parameter scan are saved to a object.

In [1]:

```
# PySCeS model instantiation using the `example_model.py` file
# with name `mod`
mod = pysces.model('example_model')
mod.SetQuiet()

# Parameter scan setup and execution
# Here we are changing the value of `Vf2` over logarithmic
# scale from `log10(1)` (or 0) to log10(100) (or 2) for a
# 100 points.
mod.scan_in = 'Vf2'
mod.scan_out = ['J_R1','J_R2','J_R3']
mod.Scan1(numpy.logspace(0,2,100))

# Instantiation of `Data2D` object with name `scan_data`
column_names = [mod.scan_in] + mod.scan_out
```

(continues on next page)

(continued from previous page)

```
scan_data = psctb.utils.plotting.Data2D(mod=mod,
                                        column_names=column_names,
                                        data_array=mod.scan_res)
```

Out[1]:

```
Assuming extension is .psc
Using model directory: /home/jr/Pysces/psc
/home/jr/Pysces/psc/example_model.psc loading .....
Parsing file: /home/jr/Pysces/psc/example_model.psc

Calculating L matrix . . . . . .  done.
Calculating K matrix . . . . . .  done.
```

Results that can be accessed via `scan_results`:

In [2]:

```
# Each key represents a field through which results can be accessed
list(scan_data.scan_results.keys())
```

Out[2]:

```
['scan_in', 'scan_out', 'scan_range', 'scan_results', 'scan_points']
```

e.g. The first 10 data points for the scan results:

In [3]:

```
scan_data.scan_results.scan_results[:10,:]
```

Out[3]:

```
array([[10.92333359,  0.97249011,  9.95084348],
       [10.96942935,  1.01871933,  9.95071002],
       [11.01771234,  1.06714226,  9.95057008],
       [11.06828593,  1.1178626 ,  9.95042334],
       [11.12125839,  1.17098892,  9.95026946],
       [11.176743  ,  1.2266349 ,  9.9501081 ],
       [11.23485838,  1.28491951,  9.94993887],
       [11.29572869,  1.34596731,  9.94976138],
       [11.35948389,  1.40990867,  9.94957522],
       [11.42626002,  1.47688006,  9.94937996]])
```

Results can be saved using the default path as discussed in *Saving and default directories* with the `save_results` method:

In [4]:

```
scan_data.save_results()
```

Or they can be saved to a specified location:

In [5]:

```
# This path leads to the Pysces root folder
data_file_name = '~/Pysces/example_mod_Vf2_scan.csv'
```

(continues on next page)

(continued from previous page)

```
# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    data_file_name = psctb.utils.misc.unix_to_windows_path(data_file_name)
else:
    data_file_name = path.expanduser(data_file_name)

scan_data.save_results(file_name=data_file_name)
```

Finally, a `ScanFig` object can be created using the `plot` method:

In [6]:

```
# Instantiation of `ScanFig` object with name `scan_figure`
scan_figure = scan_data.plot()
```

3.5.2 ScanFig

The `ScanFig` class provides the actual plotting object. This tool allows users to display figures with results directly in the Notebook and to control which data is displayed on the figure by use of an interactive widget based interface. As mentioned and shown above they are created by the `plot` method of a Data2D object, which means that a user never has the need to instantiate ScanFig directly.

Features

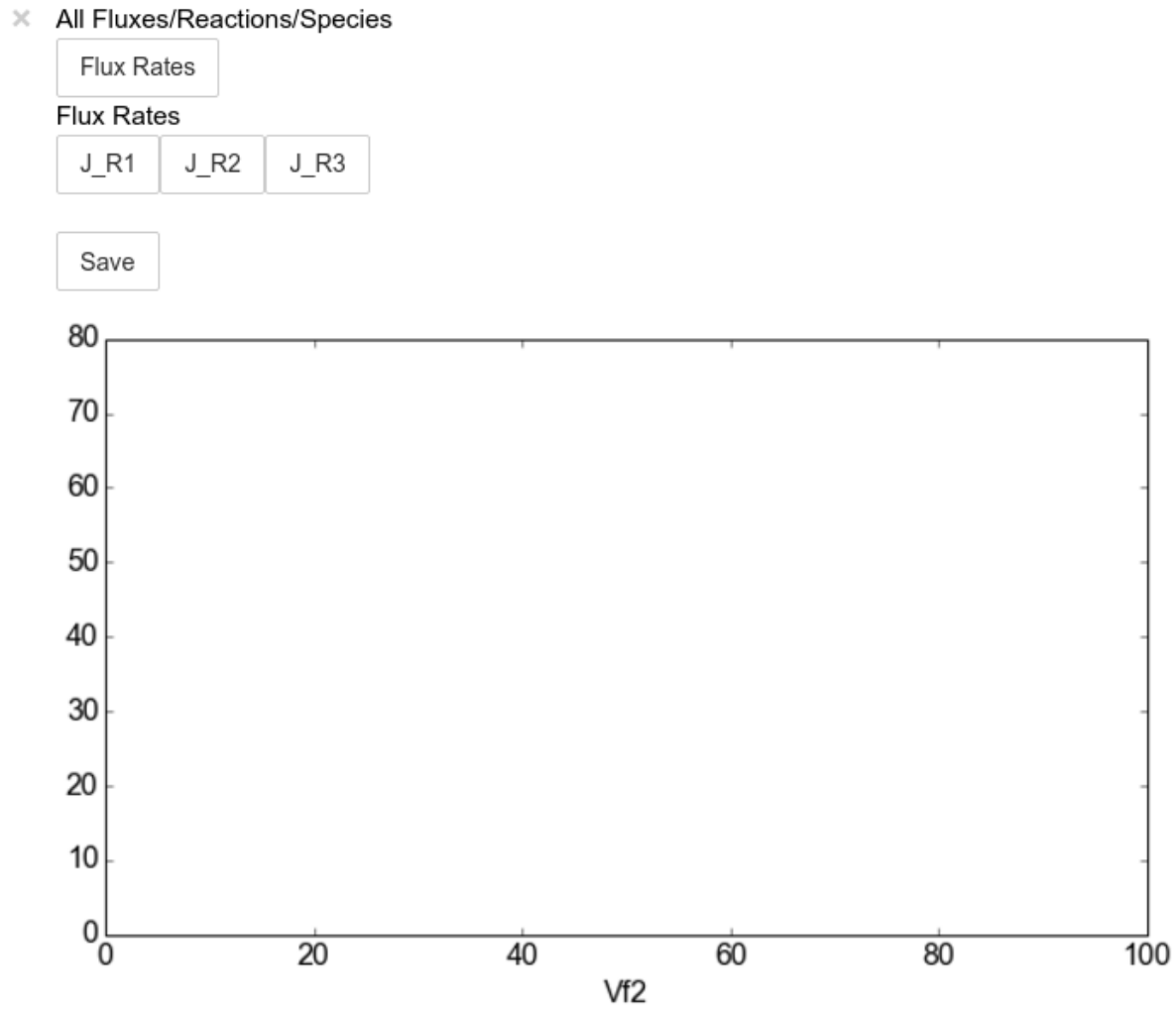
- Interactive plotting via the `interact` method.
- Script based plot generation where certain lines, or categories of lines (based on the type of information they represent), can be enabled and disabled via `toggle_line` or `toggle_category` methods.
- Saving of plots with the `save` method.
- Customisation of figures using standard `matplotlib` functionality.

Usage Example

Below is an usage example of `ScanFig` using the `scan_figure` instance created in the previous section. Here results from the parameter scan of `Vf2` as generated by `Scan1` is shown.

In [7]:

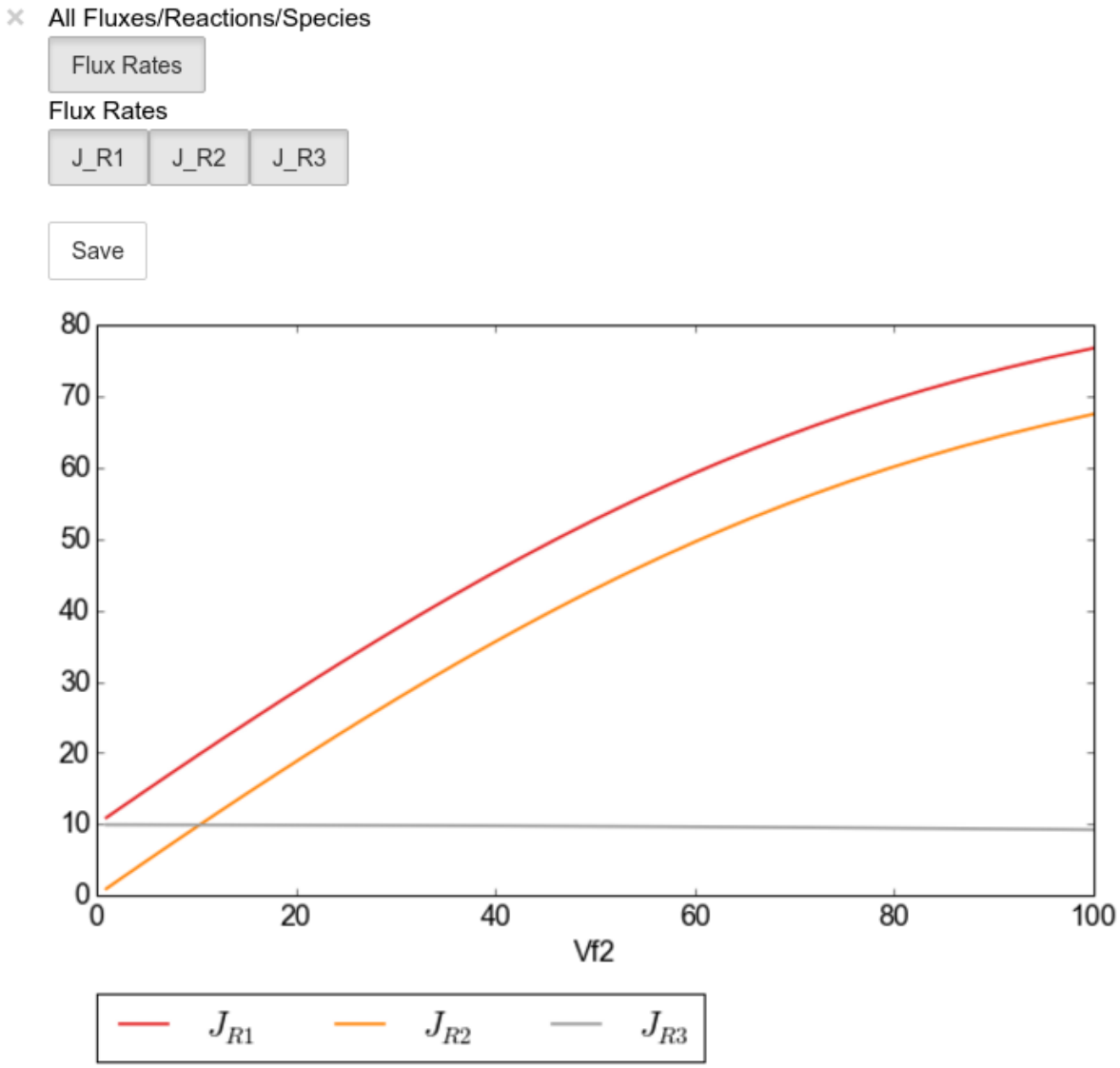
```
scan_figure.interact()
```



The Figure shown above is empty - to show lines we need to click on the buttons. First we will click on the `Flux Rates` button which will allow any of the lines that fall into the category `Flux Rates` to be enabled. Then we click the other buttons:

In [8]:

```
# The four method calls below are equivalent to clicking the category buttons
# scan_figure.toggle_category('Flux Rates',True)
# scan_figure.toggle_category('J_R1',True)
# scan_figure.toggle_category('J_R2',True)
# scan_figure.toggle_category('J_R3',True)

scan_figure.interact()
```



Note: Certain buttons act as filters for results that fall into their category. In the case above the `Flux Rates` button determines the visibility of the lines that fall into the `Flux Rates` category. In essence it overwrites the state of the buttons for the individual line categories. This feature is useful when multiple categories of results (species concentrations, elasticities, control patterns etc.) appear on the same plot by allowing to toggle the visibility of all the lines in a category.

We can also toggle the visibility with the `toggle_line` and `toggle_category` methods. Here `toggle_category` has the exact same effect as the buttons in the above example, while `toggle_line` bypasses any category filtering. The line and category names can be accessed via `line_names` and `category_names`:

In [9]:

```
print('Line names     : ', scan_figure.line_names)
print('Category names : ', scan_figure.category_names)
```

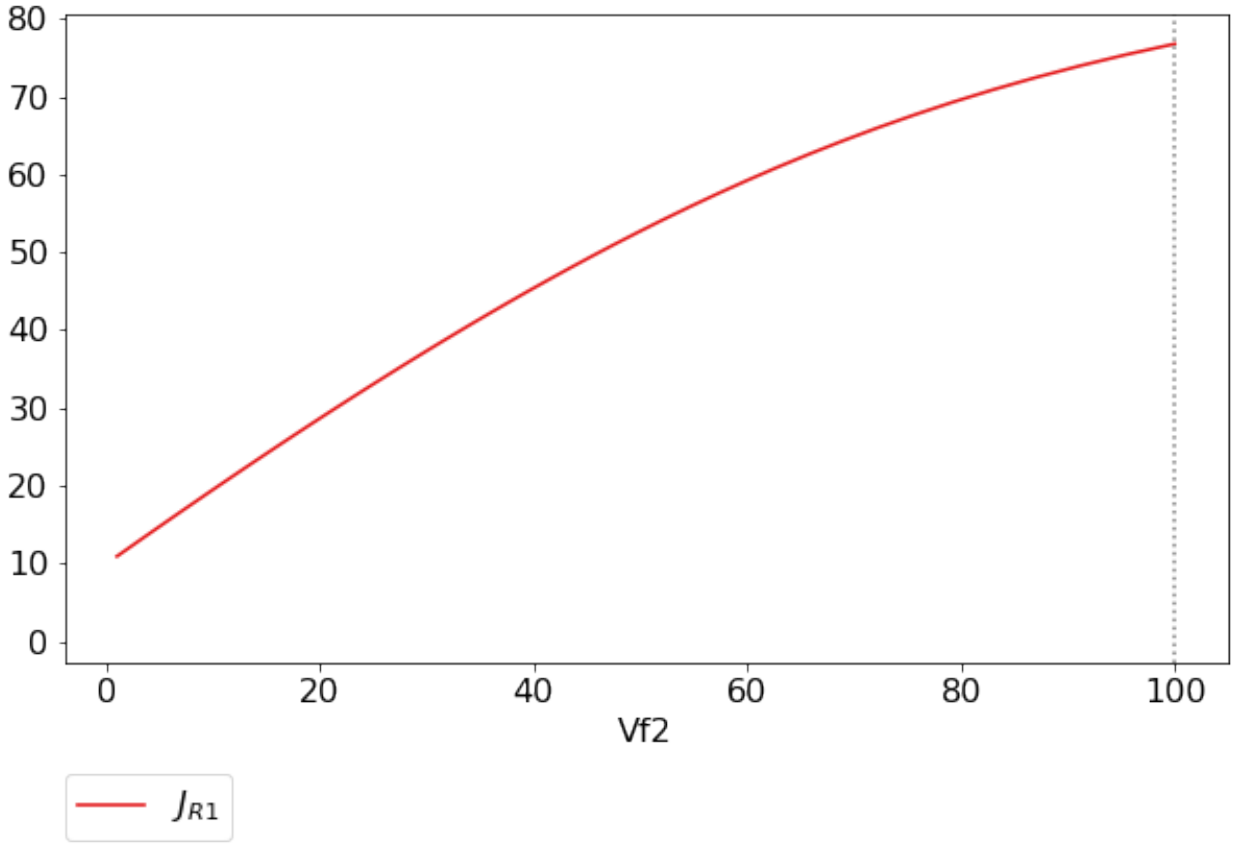
Out[9]:

```
Line names      :  ['J_R1', 'J_R2', 'J_R3']
Category names :  ['J_R2', 'Flux Rates', 'J_R1', 'J_R3']
```

In the example below we set the `Flux Rates` visibility to `False`, but we set the `J_R1` line visibility to `True`. Finally we use the `show` method instead of `interact` to display the figure.

In [10]:

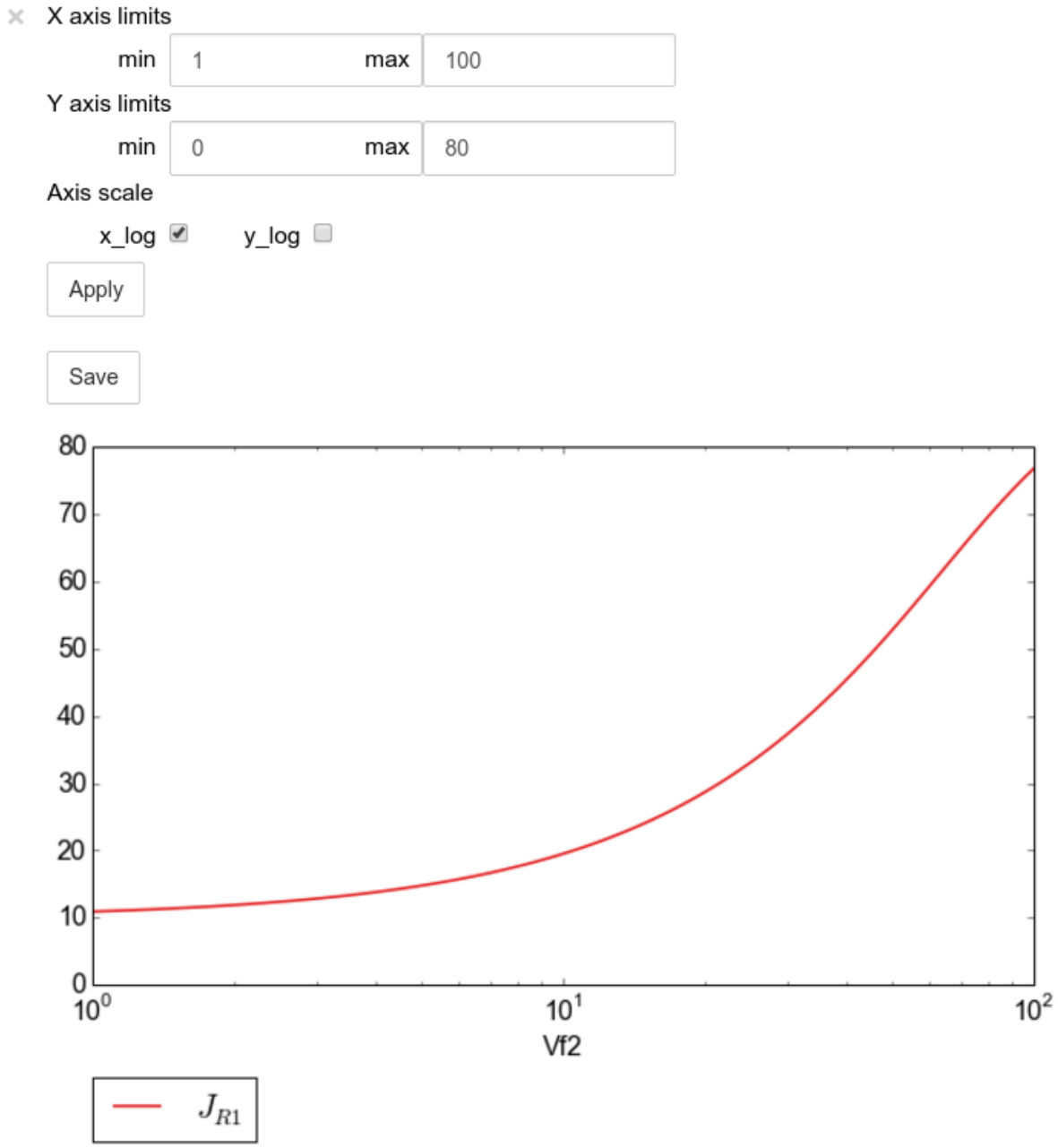
```
scan_figure.toggle_category('Flux Rates',False)
scan_figure.toggle_line('J_R1',True)
scan_figure.show()
```



The figure axes can also be adjusted via the `adjust_figure` method. Recall that the `Vf2` scan was performed for a logarithmic scale rather than a linear scale. We will therefore set the x axis to log and its minimum value to `1`. These settings are applied by clicking the `Apply` button.

In [11]:

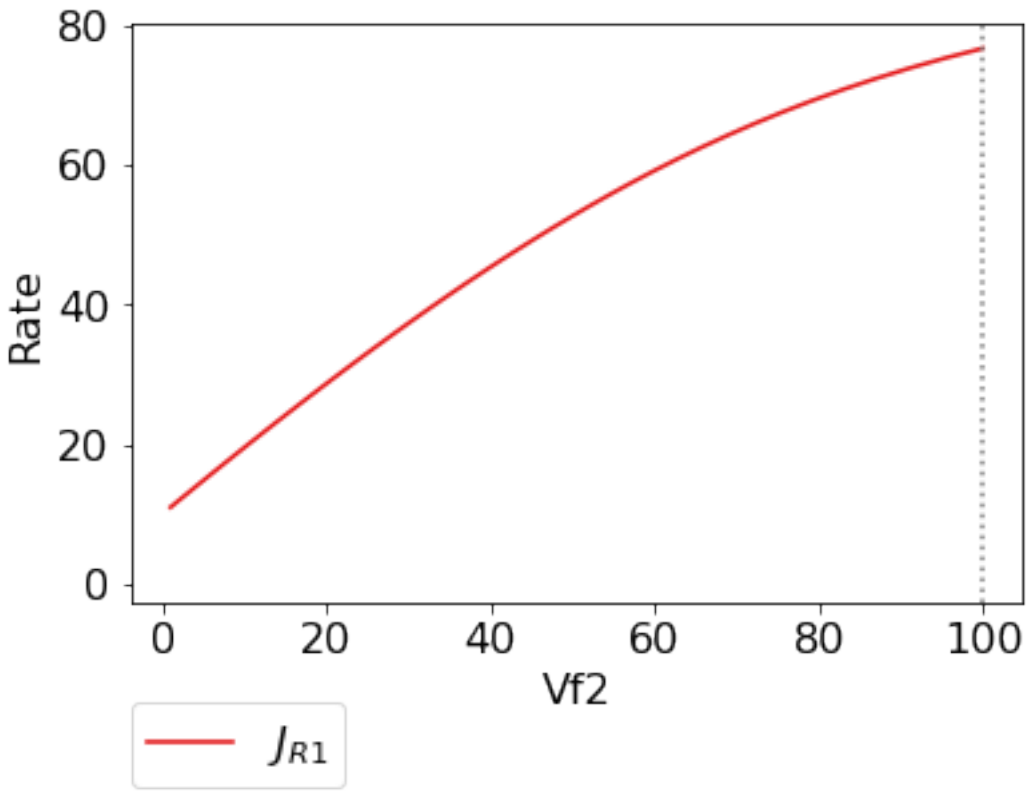
```
scan_figure.adjust_figure()
```



The underlying `matplotlib` objects can be accessed through the `fig` and `ax` fields for the figure and axes, respectively. This allows for manipulation of the figures using `matplotlib's` functionality.

In [12]:

```
scan_figure.fig.set_size_inches((6,4))
scan_figure.ax.set_ylabel('Rate')
scan_figure.line_names
scan_figure.show()
```



Finally the plot can be saved using the `save` method (or equivalently by pressing the `save` button) without specifying a path where the file will be saved as an `svg` vector image to the default directory as discussed under *Saving and default directories*:

In [13]:

```
scan_figure.save()
```

A file name together with desired extension (and image format) can also be specified:

In [14]:

```
# This path leads to the Pysces root folder
fig_file_name = '~/Pysces/example_mod_Vf2_scan.png'

# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    fig_file_name = psctb.utils.misc.unix_to_windows_path(fig_file_name)
else:
    fig_file_name = path.expanduser(fig_file_name)

scan_figure.save(file_name=fig_file_name)
```

3.5.3 Tables

In PySCeSToolbox, results are frequently stored in an dictionary-like structure belonging to an analysis object. In most cases the dictionary will be named with `_results` appended to the type of results (e.g. Control coefficient results in `SymCa` are saved as `cc_results` while the parametrised internal metabolite scan results of `RateChar` are saved as `scan_results`).

In most cases the results stored are structured so that a single dictionary key is mapped to a single result (or result object). In these cases simply inspecting the variable in the IPython/Jupyter Notebook displays these results in an html style table where the variable name is displayed together with it's value e.g. for `cc_results` each control coefficient will be displayed next to its value at steady-state.

Finally, any 2D data-structure commonly used in together with PyCSeS and PySCeSToolbox can be displayed as an html table (e.g. list of lists, NumPy arrays, SymPy matrices).

Usage Example

Below we will construct a list of lists and display it as an html table.Captions can be either plain text or contain html tags.

In [15]:

```
list_of_lists = [['a','b','c'],[1.2345,0.6789,0.0001011],[12,13,14]]
```

In [16]:

```
psctb.utils.misc.html_table(list_of_lists,
                            caption='Example')
```

a	b	c
1.23	0.68	0.00
12.00	13.00	14.00

Table: Example

By default floats are all formatted according to the argument `float_fmt` which defaults to `%.2f` (using the standard Python formatter string syntax). A formatter function can be passed to as the `formatter` argument which allows for more customisation.

Below we instantiate such a formatter using the `formatter_factory` function. Here all float values falling within the range set up by `min_val` and `max_val` (which includes the minimum, but excludes the maximum) will be formatted according to `default_fmt`, while outliers will be formatted according to `outlier_fmt`.

In [17]:

```
formatter = psctb.utils.misc.formatter_factory(min_val=0.1,
                                               max_val=10,
                                               default_fmt='%.1f',
                                               outlier_fmt='%.2e')
```

The constructed `formatter` takes a number (e.g. float, int, etc.) as argument and returns a formatter string according to the previously setup parameters.

In [18]:

```
print(formatter(0.09)) # outlier
print(formatter(0.1))  # min for default
print(formatter(2))    # within range for default
print(formatter(9))    # max int for default
print(formatter(10))   # outlier
```

Out[18]:

```
9.00e-02
0.1
2.0
9.0
1.00e+01
```

Using this `formatter` with the previously constructed `list_of_lists` lead to a differently formatted html representation of the data:

In [19]:

```
psctb.utils.misc.html_table(list_of_lists,
                            caption='Example',
                            formatter=formatter,    # Previously constructed formatter
                            first_row_headers=True) # The first row can be set as the␣
↪header
```

a	b	c
1.2	0.7	1.01e-04
1.20e+01	1.30e+01	1.40e+01

Table: Example

3.6 Graphic Representation of Metabolic Networks

PySCeSToolbox includes functionality for displaying interactive graph representations of metabolic networks through the `ModelGraph` tool. The main purpose of this feature is to allow for the visualisation of control patterns in `SymCa`. Currently, this tool is fairly limited in terms of its capabilities and therefore does not represent a replacement for more fully featured tools such as e.g. CellDesigner. One such limitation is that no automatic layout capabilities are included, and nodes representing species and concentrations have to be laid out by hand. Nonetheless it is useful for quickly visualising the structure of pathway and, as previously mentioned, for visualising the importance of various control patterns in `SymCa`.

3.6.1 Features

- Displays interactive (d3.js based) reaction networks in the notebook.
- Layouts can be saved and applied to other similar networks.

3.6.2 Usage Example

The main use case is for visualising control patterns. However, `ModelGraph` can be used in this capacity, the graph layout has to be defined. Below we will set up the layout for the `example_model`.

First we load the model and instantiate a `ModelGraph` object using the model. The show method displays the graph.

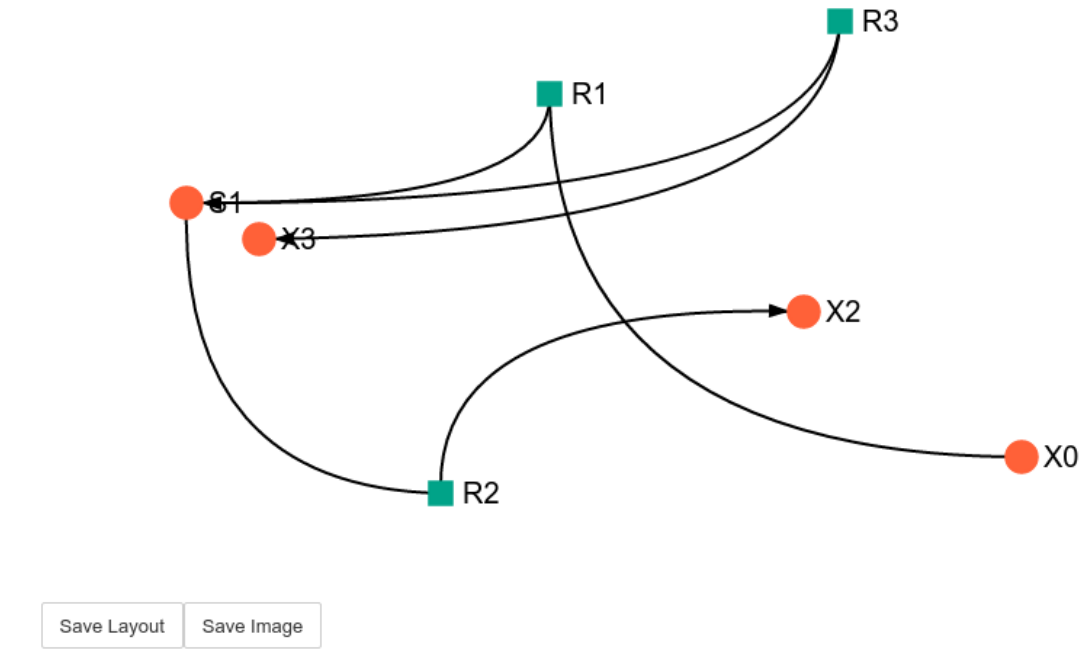
In [20]:

```
model_graph = psctb.ModelGraph(mod)
```

Unless a layout has been previously defined, the species and reaction nodes will be placed randomly. Nodes are snap to an invisible grid.

In [21]:

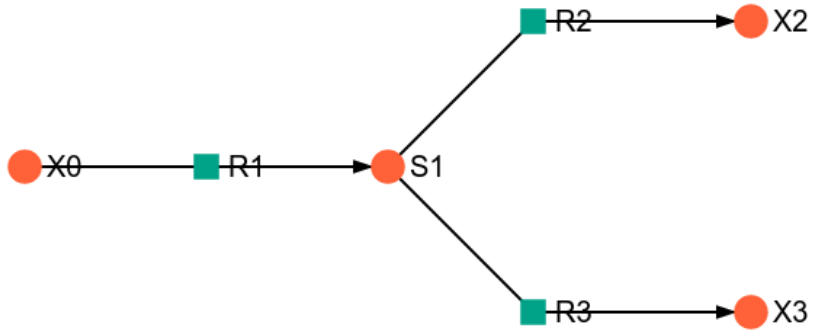
```
model_graph.show()
```



A layout file for the `example_model` is included (see link for details) and can be loaded by specifying the location of the layout file on the disk during `ModelGraph` instantiation.

In [22]:

```
# This path leads to the provided layout file
path_to_layout = '~/Pysces/psc/example_model_layout.dict'

# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    path_to_layout = psctb.utils.misc.unix_to_windows_path(path_to_layout)
else:
    path_to_layout = path.expanduser(path_to_layout)


model_graph = psctb.ModelGraph(mod, pos_dic=path_to_layout)
model_graph.show()
```

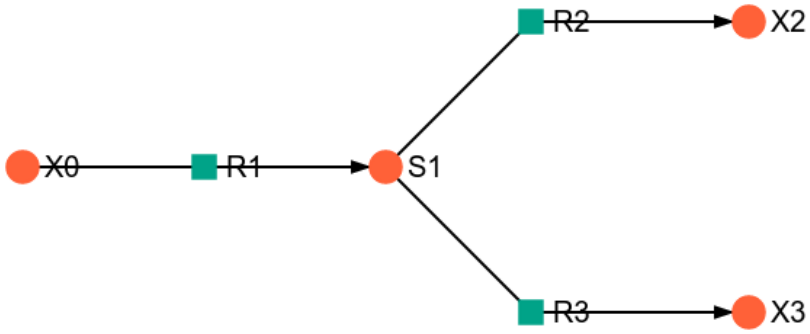


Clicking the `Save Layout` button saves this layout to the `~/Pysces/example_model/model_graph` or `C:\Pysces\example_model\model_graph` directory for later use. The `Save Image` Button wil save an svg image of the graph to the same location.

Now any future instantiation of a `ModelGraph` object for `example_model` will use the saved layout automatically.

In [23]:

```
model_graph = psctb.ModelGraph(mod)
model_graph.show()
```



Save Layout Save Image

CHAPTER 4

RateChar

RateChar is a tool for performing generalised supply-demand analysis (GSDA) [5,6]. This entails the generation data needed to draw rate characteristic plots for all the variable species of metabolic model through parameter scans and the subsequent visualisation of these data in the form of `ScanFig` objects.

4.1 Features

- Performs parameter scans for any variable species of a metabolic model
- Stores results in a structure similar to `Data2D`.
- Saving of raw parameter scan data, together with metabolic control analysis results to disk.
- Saving of `RateChar` sessions to disk for later use.
- Generates rate characteristic plots from parameter scans (using `ScanFig`).
- Can perform parameter scans of any variable species with outputs for relevant response, partial response, elasticity and control coefficients (with data stores as `Data2D` objects).

4.2 Usage and Feature Walkthrough

4.2.1 Workflow

Performing GSDA with `RateChar` usually requires taking the following steps:

1. Instantiation of `RateChar` object (optionally specifying default settings).
2. Performing a configurable parameter scan of any combination of variable species (or loading previously saved results).
3. Accessing scan results through `RateCharData` objects corresponding to the names of the scanned species that can be found as attributes of the instantiated `RateChar` object.

4. Plotting results of a particular species using the `plot` method of the `RateCharData` object corresponding to that species.
5. Further analysis using the `do_mca_scan` method.
6. Session/Result saving if required.
7. Further Analysis

Note: Parameter scans are performed for a range of concentrations values between two set values. By default the minimum and maximum scan range values are calculated relative to the steady state concentration the species for which a scan is performed respectively using a division and multiplication factor. Minimum and maximum values may also be explicitly specified. Furthermore the number of points for which a scan is performed may also be specified. Details of how to access these options will be discussed below.

4.2.2 Object Instantiation

Like most tools provided in PySCeSToolbox, instantiation of a `RateChar` object requires a pysces model object (`PysMod`) as an argument. A `RateChar` session will typically be initiated as follows (here we will use the included lin4_fb.psc model):

`In [1]:`

```
mod = pysces.model('lin4_fb.psc')
rc = psctb.RateChar(mod)
```

`Out[1]:`

```
Using model directory: /home/jr/Pysces/psc
/home/jr/Pysces/psc/lin4_fb.psc loading .....
Parsing file: /home/jr/Pysces/psc/lin4_fb.psc
Info: "X4" has been initialised but does not occur in a rate equation

Calculating L matrix . . . . . .  done.
Calculating K matrix . . . . . .  done.
```

Default parameter scan settings relating to a specific `RateChar` session can also be specified during instantiation:

`In [2]:`

```
rc = psctb.RateChar(mod,min_concrange_factor=100,
                    max_concrange_factor=100,
                    scan_points=255,
                    auto_load=False)
```

- `min_concrange_factor` : The steady state division factor for calculating scan range minimums *(default: 100)*.
- `max_concrange_factor` : The steady state multiplication factor for calculating scan range maximums *(default: 100)*.
- `scan_points` : The number of concentration sample points that will be taken during parameter scans *(default: 256)*.
- `auto_load` : If `True` `RateChar` will try to load saved data from a previous session during instantiation. Saved data is unaffected by the above options and are only subject to the settings specified during the session where they were generated. *(default:* `False`*)*.

The settings specified with these optional arguments take effect when the corresponding arguments are not specified during a parameter scan.

4.2.3 Parameter Scan

After object instantiation, parameter scans may be performed for any of the variable species using the `do_ratechar` method. By default `do_ratechar` will perform parameter scans for all variable metabolites using the settings specified during instantiation. For saving/loading see Saving/Loading Sessions below.

In [3]:

```
mod.species
```

Out[3]:

```
('S1', 'S2', 'S3')
```

In [4]:

```
rc.do_ratechar()
```

Various optional arguments, similar to those used during object instantiation, can be used to override the default settings and customise any parameter scan:

- `fixed` : A string or list of strings specifying the species for which to perform a parameter scan. The string `'all'` specifies that all variable species should be scanned. *(default: ''all'')*
- `scan_min` : The minimum value of the scan range, overrides `min_concrange_factor` *(default: None)*.
- `scan_max` : The maximum value of the scan range, overrides `max_concrange_factor` *(default: None)*.
- `min_concrange_factor` : The steady state division factor for calculating scan range minimums *(default: None)*
- `max_concrange_factor` : The steady state multiplication factor for calculating scan range maximums *(default: None)*.
- `scan_points` : The number of concentration sample points that will be taken during parameter scans *(default: None)*.
- `solver` : An integer value that specifies which solver to use (0:Hybrd,1:NLEQ,2:FINTSLV). *(default: 0)*.

Note: For details on different solvers see the PySCeS documentation:

For example in a scenario where we only wanted to perform parameter scans of 200 points for the metabolites `S1` and `S3` starting at a value of 0.02 and ending at a value 110 times their respective steady-state values the method would be called as follows:

In [5]:

```
rc.do_ratechar(fixed=['S1','S3'], scan_min=0.02, max_concrange_factor=110, scan_
↪points=200)
```

4.2.4 Accessing Results

Parameter Scan Results

Parameter scan results for any particular species are saved as an attribute of the `RateChar` object under the name of that species. These `RateCharData` objects are similar to `Data2D` objects with parameter scan results being accessible through a `scan_results` DotDict:

In [6]:

```
# Each key represents a field through which results can be accessed
sorted(rc.S3.scan_results.keys())
```

Out[6]:

```
['J_R3',
 'J_R4',
 'ecR3_S3',
 'ecR4_S3',
 'ec_data',
 'ec_names',
 'fixed',
 'fixed_ss',
 'flux_data',
 'flux_max',
 'flux_min',
 'flux_names',
 'prcJR3_S3_R1',
 'prcJR3_S3_R3',
 'prcJR3_S3_R4',
 'prcJR4_S3_R1',
 'prcJR4_S3_R3',
 'prcJR4_S3_R4',
 'prc_data',
 'prc_names',
 'rcJR3_S3',
 'rcJR4_S3',
 'rc_data',
 'rc_names',
 'scan_max',
 'scan_min',
 'scan_points',
 'scan_range',
 'total_demand',
 'total_supply']
```

Note: The `DotDict` data structure is essentially a dictionary with additional functionality for displaying results in table form (when appropriate) and for accessing data using dot notation in addition the normal dictionary bracket notation.

In the above dictionary-like structure each field can represent different types of data, the most simple of which is a single value, e.g., `scan_min` and `fixed`, or a 1-dimensional numpy ndarray which represent input (`scan_range`) or output (`J_R3`, `J_R4`, `total_supply`):

In [7]:

```
# Single value results
```

(continues on next page)

(continued from previous page)

```
# scan_min value
rc.S3.scan_results.scan_min
```

Out[7]:

```
0.020000000000000004
```

In [8]:

```
# fixed metabolite name
rc.S3.scan_results.fixed
```

Out[8]:

```
'S3'
```

In [9]:

```
# 1-dimensional ndarray results (only every 10th value of 200 value arrays)

# scan_range values
rc.S3.scan_results.scan_range[::10]
```

Out[9]:

```
array([2.00000000e-02, 3.42884038e-02, 5.87847316e-02, 1.00781731e-01,
       1.72782234e-01, 2.96221349e-01, 5.07847861e-01, 8.70664626e-01,
       1.49268501e+00, 2.55908932e+00, 4.38735439e+00, 7.52176893e+00,
       1.28954725e+01, 2.21082584e+01, 3.79028445e+01, 6.49814018e+01,
       1.11405427e+02, 1.90995713e+02, 3.27446907e+02, 5.61381587e+02])
```

In [10]:

```
# J_R3 values for scan_range
rc.S3.scan_results.J_R3[::10]
```

Out[10]:

```
array([199.95837618, 199.95793443, 199.95717575, 199.95586349,
       199.95351373, 199.94862132, 199.93277067, 199.84116362,
       199.13023486, 193.32039795, 154.71345957,  58.57037566,
        12.34220931,   4.95993525,   4.0627301 ,   3.94870431,
         3.91873852,   3.88648387,   3.83336626,   3.74248032])
```

In [11]:

```
# total_supply values for scan_range
rc.S3.scan_results.total_supply[::10]

# Note that J_R3 and total_supply are equal in this case, because S3
# only has a single supply reaction
```

Out[11]:

```
array([199.95837618, 199.95793443, 199.95717575, 199.95586349,
       199.95351373, 199.94862132, 199.93277067, 199.84116362,
```

(continues on next page)

(continued from previous page)

```
       199.13023486, 193.32039795, 154.71345957,  58.57037566,
        12.34220931,   4.95993525,   4.0627301 ,   3.94870431,
         3.91873852,   3.88648387,   3.83336626,   3.74248032])
```

Finally data needed to draw lines relating to metabolic control analysis coefficients are also included in `scan_results`. Data is supplied in 3 different forms: Lists names of the coefficients (under `ec_names`, `prc_names`, etc.), 2-dimensional arrays with exactly 4 values (representing 2 sets of x,y coordinates) that will be used to plot coefficient lines, and 2-dimensional array that collects coefficient line data for each coefficient type into single arrays (under `ec_data`, `prc_names`, etc.).

In [12]:

```
# Metabolic Control Analysis coefficient line data

# Names of elasticity coefficients related to the 'S3' parameter scan
rc.S3.scan_results.ec_names
```

Out[12]:

```
['ecR4_S3', 'ecR3_S3']
```

In [13]:

```
# The x, y coordinates for two points that will be used to plot a
# visual representation of ecR3_S3
rc.S3.scan_results.ecR3_S3
```

Out[13]:

```
array([[  7.74368133, 166.89714925],
       [  8.87553568,  11.92812753]])
```

In [14]:

```
# The x,y coordinates for two points that will be used to plot a
# visual representation of ecR4_S3
rc.S3.scan_results.ecR4_S3
```

Out[14]:

```
array([[ 2.77554202, 39.66048804],
       [24.76248588, 50.19530973]])
```

In [15]:

```
# The ecR3_S3 and ecR4_S3 data collected into a single array
# (horizontally stacked).
rc.S3.scan_results.ec_data
```

Out[15]:

```
array([[  2.77554202,  39.66048804,   7.74368133, 166.89714925],
       [ 24.76248588,  50.19530973,   8.87553568,  11.92812753]])
```

Metabolic Control Analysis Results

The in addition to being able to access the data that will be used to draw rate characteristic plots, the user also has access to the values of the metabolic control analysis coefficient values at the steady state of any particular species via the `mca_results` field. This field represents a `DotDict` dictionary-like object (like `scan_results`), however as each key maps to exactly one result, the data can be displayed as a table (see Basic Usage):

In [16]:

```
# Metabolic control analysis coefficient results
rc.S3.mca_results
```

C_{R1}^{JR3}	1.000
C_{R3}^{JR3}	4.612e-05
C_{R4}^{JR3}	0.000
C_{R1}^{JR4}	0.000
C_{R3}^{JR4}	0.000
C_{R4}^{JR4}	1.000
ε_{S3}^{R1}	-2.888
ε_{S3}^{R3}	-19.341
ε_{S3}^{R4}	0.108
${}^{R1}R_{S3}^{JR3}$	-2.888

${}^{R3}R_{S3}^{JR3}$	-8.920e-04
${}^{R4}R_{S3}^{JR3}$	0.000
${}^{R1}R_{S3}^{JR4}$	-0.000
${}^{R3}R_{S3}^{JR4}$	-0.000
${}^{R4}R_{S3}^{JR4}$	0.108
R_{S3}^{JR3}	-2.889
R_{S3}^{JR4}	0.108

Naturally, coefficients can also be accessed individually:

In [17]:

```
# Control coefficient ccJR3_R1 value
rc.S3.mca_results.ccJR3_R1
```

Out[17]:

```
0.999867853018012
```

4.2.5 Plotting Results

One of the strengths of generalised supply-demand analysis is that it provides an intuitive visual framework for inspecting results through the used of rate characteristic plots. Naturally this is therefore the main focus of RateChar. Parameter scan results for any particular species can be visualised as a `ScanFig` object through the `plot` method:

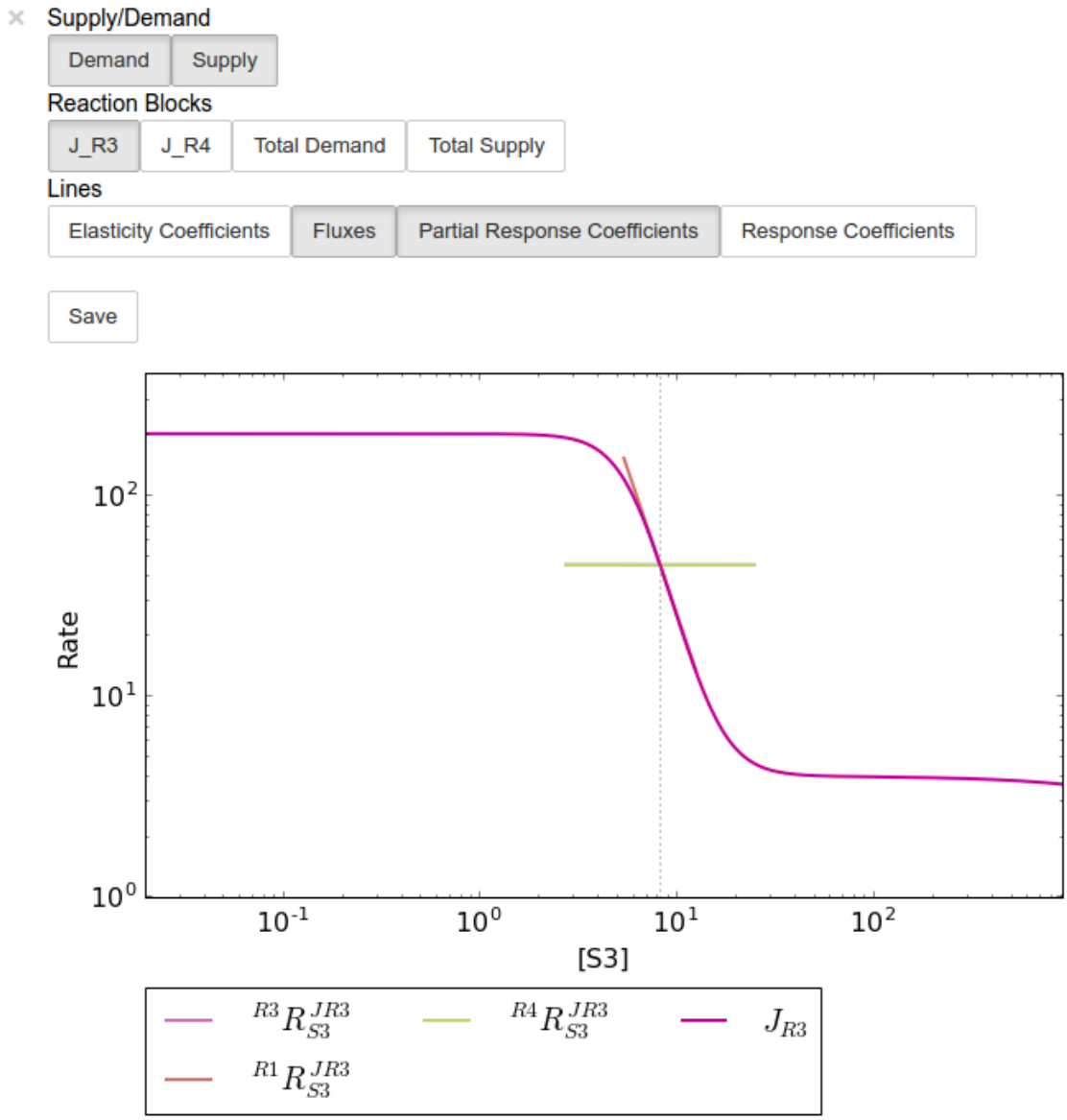
In [18]:

```
# Rate characteristic plot for 'S3'.
S3_rate_char_plot = rc.S3.plot()
```

Plots generated by `RateChar` do not have widgets for each individual line; lines are enabled or disabled in batches according to the category they belong to. By default the `Fluxes`, `Demand` and `Supply` categories are enabled when plotting. To display the partial response coefficient lines together with the flux lines for `J_R3`, for instance, we would click the `J_R3` and the `Partial Response Coefficients` buttons (in addition to those that are enabled by default).

In [19]:

```
# Display plot via `interact` and enable certain lines by clicking category buttons.

# The two method calls below are equivalent to clicking the 'J_R3'
# and 'Partial Response Coefficients' buttons:
# S3_rate_char_plot.toggle_category('J_R3',True)
# S3_rate_char_plot.toggle_category('Partial Response Coefficients',True)

S3_rate_char_plot.interact()
```

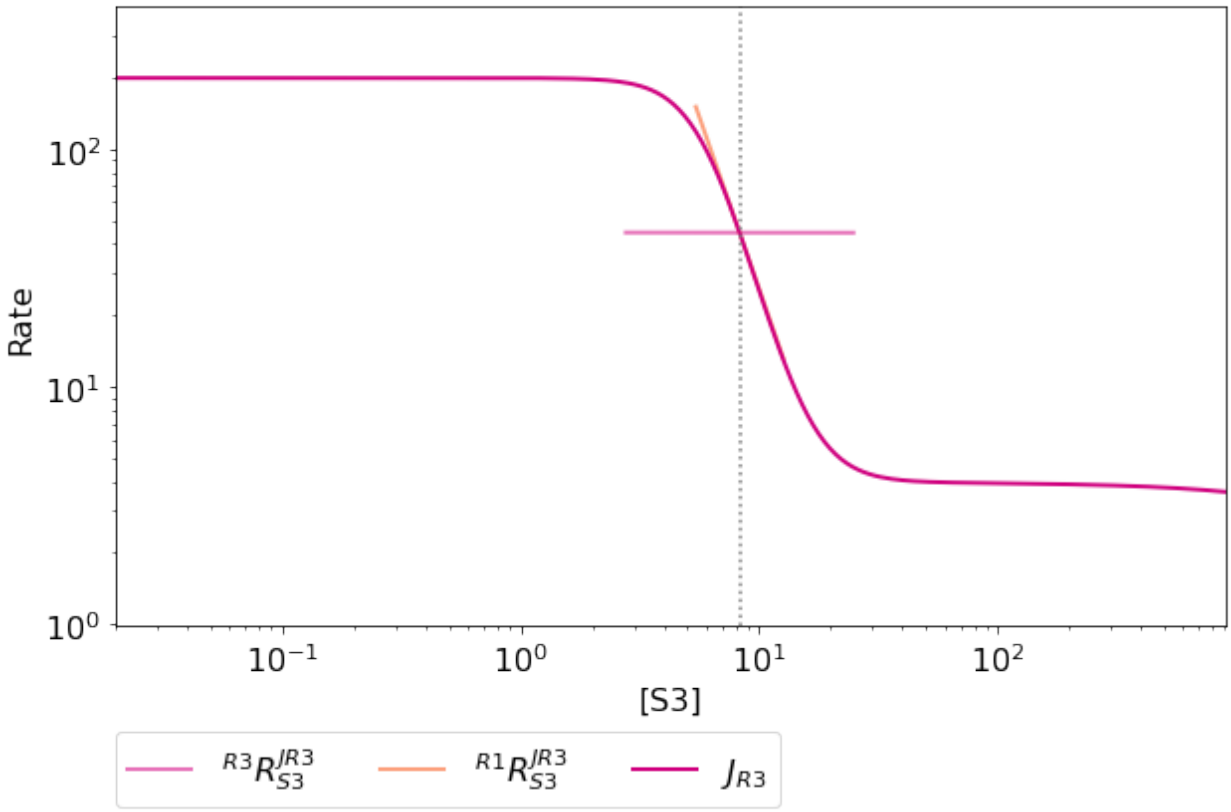


Modifying the status of individual lines is still supported, but has to take place via the `toggle_line` method. As an

example `prcJR3_C_R4` can be disabled as follows:

In [20]:

```
S3_rate_char_plot.toggle_line('prcJR3_S3_R4', False)
S3_rate_char_plot.show()
```



Note: For more details on saving see the sections Saving and Default Directories and ScanFig under Basic Usage.

4.2.6 Saving

Saving/Loading Sessions

RateChar sessions can be saved for later use. This is especially useful when working with large data sets that take some time to generate. Data sets can be saved to any arbitrary location by supplying a path:

In [21]:

```
# This points to a file under the Pysces directory
save_file = '~/Pysces/rc_doc_example.npz'

# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    save_file = psctb.utils.misc.unix_to_windows_path(save_file)
else:
    save_file = path.expanduser(save_file)
```

(continues on next page)

(continued from previous page)

```
rc.save_session(file_name = save_file)
```

When no path is supplied the dataset will be saved to the default directory. (Which should be "~/Pysces/lin4_fb/ratechar/save_data.npz" in this case.

In [22]:

```
rc.save_session() # to "~/Pysces/lin4_fb/ratechar/save_data.npz"
```

Similarly results may be loaded using the `load_session` method, either with or without a specified path:

In [23]:

```
rc.load_session(save_file)
# OR
rc.load_session() # from "~/Pysces/lin4_fb/ratechar/save_data.npz"
```

Saving Results

Results may also be exported in csv format either to a specified location or to the default directory. Unlike saving of sessions results are spread over multiple files, so here an existing folder must be specified:

In [24]:

```
# This points to a subdirectory under the Pysces directory
save_folder = '~/Pysces/lin4_fb/'

# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    save_folder = psctb.utils.misc.unix_to_windows_path(save_folder)
else:
    save_folder = path.expanduser(save_folder)

rc.save_results(save_folder)
```

A subdirectory will be created for each metabolite with the files `ec_results_N`, `rc_results_N`, `prc_results_N`, `flux_results_N` and `mca_summary_N` (where `N` is a number starting at "0" which increments after each save operation to prevent overwriting files).

In [25]:

```
# Otherwise results will be saved to the default directory
rc.save_results(save_folder) # to sub folders in "~/Pysces/lin4_fb/ratechar/
```

Alternatively the methods `save_coefficient_results`, `save_flux_results`, `save_summary` and `save_all_results` belonging to individual `RateCharData` objects can be used to save the individual result sets.

CHAPTER 5

Symca

Symca is used to perform symbolic metabolic control analysis [3,4] on metabolic pathway models in order to dissect the control properties of these pathways in terms of the different chains of local effects (or control patterns) that make up the total control coefficient values. Symbolic/algebraic expressions are generated for each control coefficient in a pathway which can be subjected to further analysis.

5.1 Features

- Generates symbolic expressions for each control coefficient of a metabolic pathway model.
- Splits control coefficients into control patterns that indicate the contribution of different chains of local effects.
- Control coefficient and control pattern expressions can be manipulated using standard `SymPy` functionality.
- Values of control coefficient and control pattern values are determined automatically and updated automatically following the calculation of standard (non-symbolic) control coefficient values subsequent to a parameter alteration.
- Analysis sessions (raw expression data) can be saved to disk for later use.
- The effect of parameter scans on control coefficient and control patters can be generated and displayed using `ScanFig`.
- Visualisation of control patterns by using `ModelGraph` functionality.
- Saving/loading of `Symca` sessions.
- Saving of control pattern results.

5.2 Usage and feature walkthrough

5.2.1 Workflow

Performing symbolic control analysis with `Symca` usually requires the following steps:

1. Instantiation of a Symca object using a PySCeS model object.
2. Generation of symbolic control coefficient expressions.
3. Access generated control coefficient expression results via cc_results and the corresponding control coefficient name (see Basic Usage)
4. Inspection of control coefficient values.
5. Inspection of control pattern values and their contributions towards the total control coefficient values.
6. Inspection of the effect of parameter changes (parameter scans) on the values of control coefficients and control patterns and the contribution of control patterns towards control coefficients.
7. Session/result saving if required
8. Further analysis.

5.2.2 Object instantiation

Instantiation of a Symca analysis object requires PySCeS model object (PysMod) as an argument. Using the included lin4_fb.psc model a Symca session is instantiated as follows:

In [1]:

```
mod = pysces.model('lin4_fb')
sc = psctb.Symca(mod)
```

Out[1]:

```
Assuming extension is .psc
Using model directory: /home/jr/Pysces/psc
/home/jr/Pysces/psc/lin4_fb.psc loading .....
Parsing file: /home/jr/Pysces/psc/lin4_fb.psc
Info: "X4" has been initialised but does not occur in a rate equation

Calculating L matrix . . . . . done.
Calculating K matrix . . . . . done.

(hybrd) The solution converged.
```

Additionally Symca has the following arguments:

- internal_fixed: This must be set to True in the case where an internal metabolite has a fixed concentration *(default: ''False'')*
- auto_load: If True Symca will try to load a previously saved session. Saved data is unaffected by the internal_fixed argument above *(default: ''False'')*.

Note: For the case where an internal metabolite is fixed see Fixed internal metabolites below.

5.2.3 Generating symbolic control coefficient expressions

Control coefficient expressions can be generated as soon as a Symca object has been instantiated using the do_symca method. This process can potentially take quite some time to complete, therefore we recommend saving the generated expressions for later loading (see Saving/Loading Sessions below). In the case of lin4_fb.psc expressions should be generated within a few seconds.

In [2]:

```
sc.do_symca()
```

Out[2]:

```
Simplifying matrix with 28 elements
**********
```

do_symca has the following arguments:

- internal_fixed: This must be set to True in the case where an internal metabolite has a fixed concentration *(default: ''False'')*
- auto_save_load: If set to True Symca will attempt to load a previously saved session and only generate new expressions in case of a failure. After generation of new results, these results will be saved instead. Setting internal_fixed to True does not affect previously saved results that were generated with this argument set to False *(default: ''False'')*.

5.2.4 Accessing control coefficient expressions

Generated results may be accessed via a dictionary-like cc_results object (see Basic Usage - Tables). Inspecting this cc_results object in a IPython/Jupyter notebook yields a table of control coefficient values:

In [3]:

```
sc.cc_results
```

C_{R1}^{JR1}	0.036
C_{R2}^{JR1}	3.090e-06
C_{R3}^{JR1}	1.657e-06
C_{R4}^{JR1}	0.964
C_{R1}^{JR2}	0.036
C_{R2}^{JR2}	3.090e-06
C_{R3}^{JR2}	1.657e-06
C_{R4}^{JR2}	0.964
C_{R1}^{JR3}	0.036
C_{R2}^{JR3}	3.090e-06

C_{R3}^{JR3}	1.657e-06
C_{R4}^{JR3}	0.964
C_{R1}^{JR4}	0.036
C_{R2}^{JR4}	3.090e-06
C_{R3}^{JR4}	1.657e-06
C_{R4}^{JR4}	0.964
C_{R1}^{S1}	0.323
C_{R2}^{S1}	-0.092
C_{R3}^{S1}	-0.049
C_{R4}^{S1}	-0.182

C_{R1}^{S2}	0.335
C_{R2}^{S2}	2.885e-05
C_{R3}^{S2}	-0.052
C_{R4}^{S2}	-0.284
C_{R1}^{S3}	0.334
C_{R2}^{S3}	2.871e-05
C_{R3}^{S3}	1.539e-05
C_{R4}^{S3}	-0.334
Σ	631.138

Inspecting an individual control coefficient yields a symbolic expression together with a value:

In [4]:

```
sc.cc_results.ccJR1_R4
```

$$C_{R4}^{JR1} = (-\varepsilon_{S1}^{R1}\varepsilon_{S2}^{R2}\varepsilon_{S3}^{R3} - \varepsilon_{S3}^{R1}\varepsilon_{S1}^{R2}\varepsilon_{S2}^{R3})/\ \Sigma = 0.964$$

In the above example, the expression of the control coefficient consists of two numerator terms and a common denominator shared by all the control coefficient expression signified by Σ.

Various properties of this control coefficient can be accessed such as the: * Expression (as a `SymPy` expression)

In [5]:

```
sc.cc_results.ccJR1_R4.expression
```

$$\frac{-ecR_{1S1}ecR_{2S2}ecR_{3S3} - ecR_{1S3}ecR_{2S1}ecR_{3S2}}{-ecR_{1S1}ecR_{2S2}ecR_{3S3} + ecR_{1S1}ecR_{2S2}ecR_{4S3} - ecR_{1S1}ecR_{3S2}ecR_{4S3} - ecR_{1S3}ecR_{2S1}ecR_{3S2} + ecR_{2S1}ecR_{3S2}ecR_{4S3}}$$

- Numerator expression (as a `SymPy` expression)

In [6]:

```
sc.cc_results.ccJR1_R4.numerator
```

$$-ecR_{1S1}ecR_{2S2}ecR_{3S3} - ecR_{1S3}ecR_{2S1}ecR_{3S2}$$

- Denominator expression (as a `SymPy` expression)

In [7]:

```
sc.cc_results.ccJR1_R4.denominator
```

$$-ecR_{1S1}ecR_{2S2}ecR_{3S3} + ecR_{1S1}ecR_{2S2}ecR_{4S3} - ecR_{1S1}ecR_{3S2}ecR_{4S3} - ecR_{1S3}ecR_{2S1}ecR_{3S2} + ecR_{2S1}ecR_{3S2}ecR_{4S3}$$

- Value (as a `float64`)

In [8]:

```
sc.cc_results.ccJR1_R4.value
```

Out[8]:

```
0.9640799846074221
```

Additional, less pertinent, attributes are `abs_value`, `latex_expression`, `latex_expression_full`, `latex_numerator`, `latex_name`, `name` and `denominator_object`.

The individual control coefficient numerator terms, otherwise known as control patterns, may also be accessed as follows:

In [9]:

```
sc.cc_results.ccJR1_R4.CP001
```

$$CP001 = -\varepsilon_{S1}^{R1}\varepsilon_{S2}^{R2}\varepsilon_{S3}^{R3} / \Sigma = 0.000$$

In [10]:

```
sc.cc_results.ccJR1_R4.CP002
```

$$CP002 = -\varepsilon_{S3}^{R1}\varepsilon_{S1}^{R2}\varepsilon_{S2}^{R3} / \Sigma = 0.964$$

Each control pattern is numbered arbitrarily starting from 001 and has similar properties as the control coefficient object (i.e., their expression, numerator, value etc. can also be accessed).

Control pattern percentage contribution

Additionally control patterns have a `percentage` field which indicates the degree to which a particular control pattern contributes towards the overall control coefficient value:

In [11]:

```
sc.cc_results.ccJR1_R4.CP001.percentage
```

Out[11]:

```
0.03087580996475991
```

In [12]:

```
sc.cc_results.ccJR1_R4.CP002.percentage
```

Out[12]:

```
99.96912419003525
```

Unlike conventional percentages, however, these values are calculated as percentage contribution towards the sum of the absolute values of all the control coefficients (rather than as the percentage of the total control coefficient value). This is done to account for situations where control pattern values have different signs.

A particularly problematic example of where the above method is necessary, is a hypothetical control coefficient with a value of zero, but with two control patterns with equal value but opposite signs. In this case a conventional percentage calculation would lead to an undefined (`NaN`) result, whereas our methodology would indicate that each control pattern is equally (50%) responsible for the observed control coefficient value.

5.2.5 Dynamic value updating

The values of the control coefficients and their control patterns are automatically updated when new steady-state elasticity coefficients are calculated for the model. Thus changing a parameter of `lin4_hill`, such as the V_f value of reaction 4, will lead to new control coefficient and control pattern values:

In [13]:

```
mod.reLoad()
# mod.Vf_4 has a default value of 50
mod.Vf_4 = 0.1
# calculating new steady state
mod.doMca()
```

Out[13]:

```
Parsing file: /home/jr/Pysces/psc/lin4_fb.psc
Info: "X4" has been initialised but does not occur in a rate equation

Calculating L matrix . . . . . done.
Calculating K matrix . . . . . done.

(hybrd) The solution converged.
```

In [14]:

```
# now ccJR1_R4 and its two control patterns should have new values
sc.cc_results.ccJR1_R4
```

$$C_{R4}^{JR1} = (-\varepsilon_{S1}^{R1}\varepsilon_{S2}^{R2}\varepsilon_{S3}^{R3} - \varepsilon_{S3}^{R1}\varepsilon_{S1}^{R2}\varepsilon_{S2}^{R3})/\ \Sigma = 1.000$$

In [15]:

```
# original value was 0.000
sc.cc_results.ccJR1_R4.CP001
```

$$CP001 = -\varepsilon_{S1}^{R1}\varepsilon_{S2}^{R2}\varepsilon_{S3}^{R3}/\ \Sigma = 1.000$$

In [16]:

```
# original value was 0.964
sc.cc_results.ccJR1_R4.CP002
```

$$CP002 = -\varepsilon_{S3}^{R1}\varepsilon_{S1}^{R2}\varepsilon_{S2}^{R3}/\ \Sigma = 0.000$$

In [17]:

```
# resetting to default Vf_4 value and recalculating
mod.reLoad()
mod.doMca()
```

Out[17]:

```
Parsing file: /home/jr/Pysces/psc/lin4_fb.psc
Info: "X4" has been initialised but does not occur in a rate equation

Calculating L matrix . . . . . done.
Calculating K matrix . . . . . done.

(hybrd) The solution converged.
```

5.2.6 Control pattern graphs

As described under Basic Usage, `Symca` has the functionality to display the chains of local effects represented by control patterns on a scheme of a metabolic model. This functionality can be accessed via the `highlight_patterns`

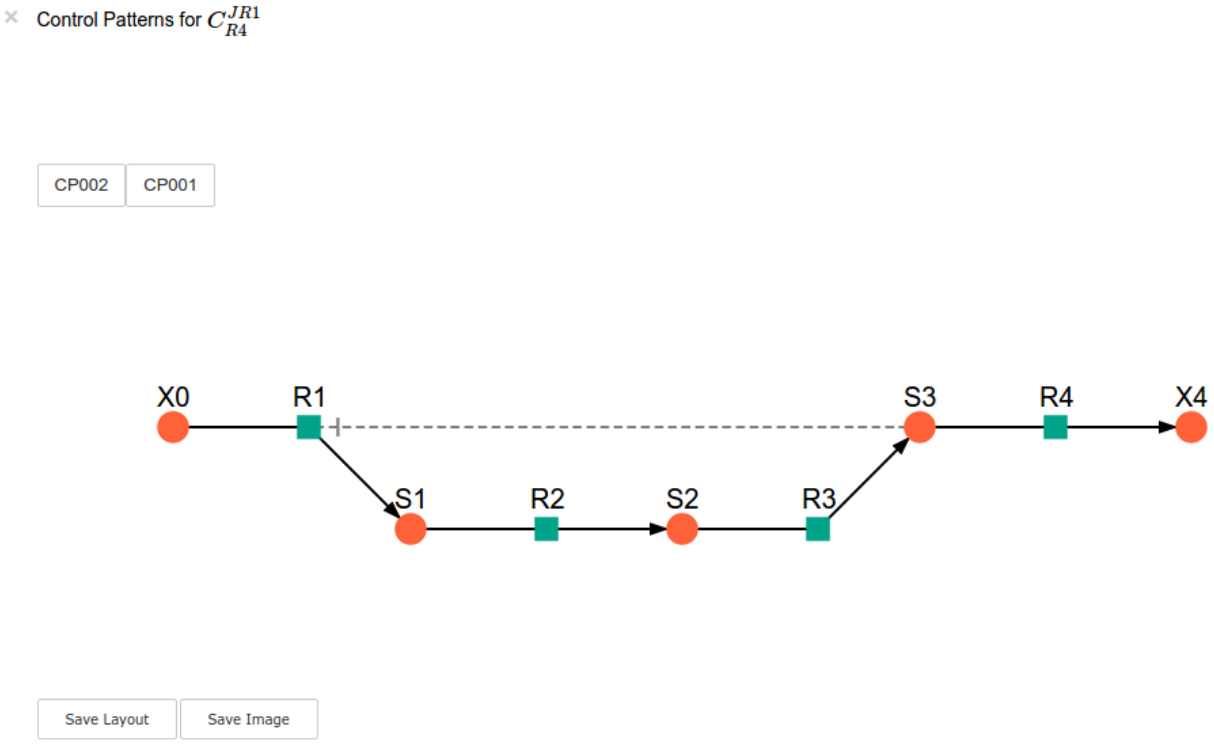
method:

In [18]:

```
# This path leads to the provided layout file
path_to_layout = '~/Pysces/psc/lin4_fb.dict'

# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    path_to_layout = psctb.utils.misc.unix_to_windows_path(path_to_layout)
else:
    path_to_layout = path.expanduser(path_to_layout)
```

In [19]:

```
sc.cc_results.ccJR1_R4.highlight_patterns(height = 350, pos_dic=path_to_layout)
```



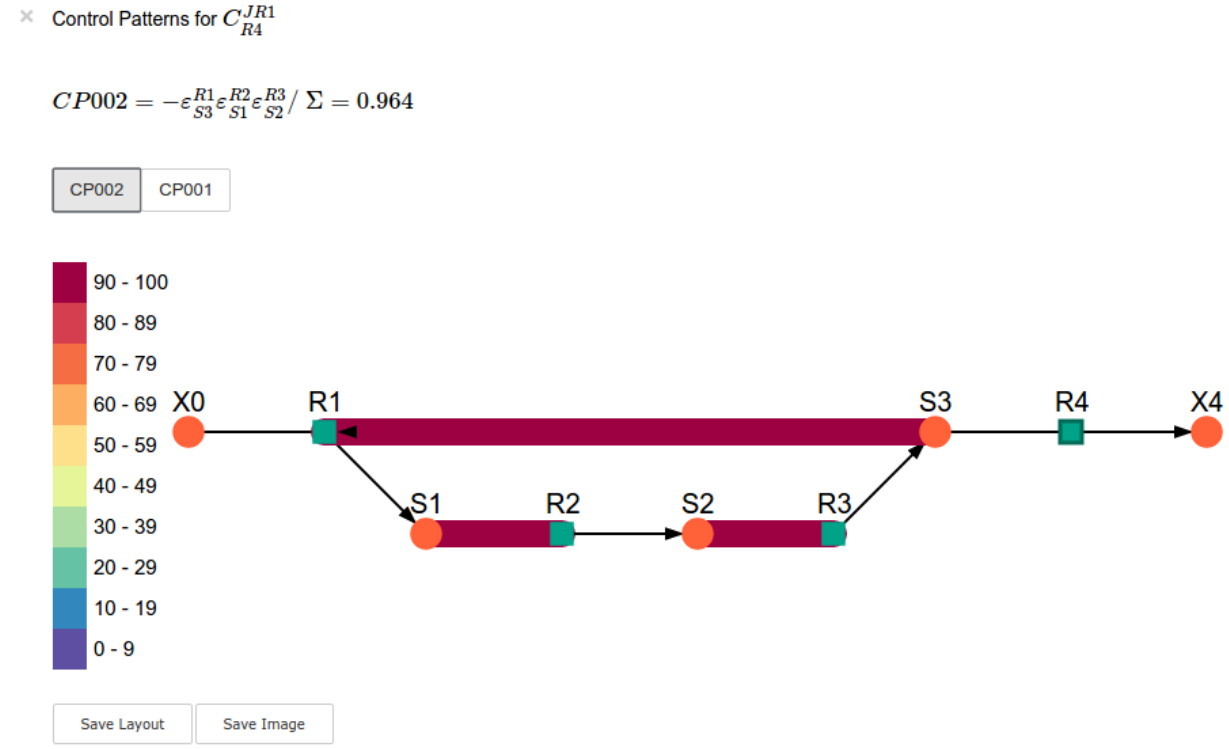
`highlight_patterns` has the following optional arguments:

- `width`: Sets the width of the graph (*default*: 900).
- `height`:Sets the height of the graph (*default*: 500).
- `show_dummy_sinks`: If `True` reactants with the "dummy" or "sink" will not be displayed (*default*: `False`).
- `show_external_modifier_links`: If `True` edges representing the interaction of external effectors with reactions will be shown (*default*: `False`).

Clicking either of the two buttons representing the control patterns highlights these patterns according according to their percentage contribution (as discussed above) towards the total control coefficient.

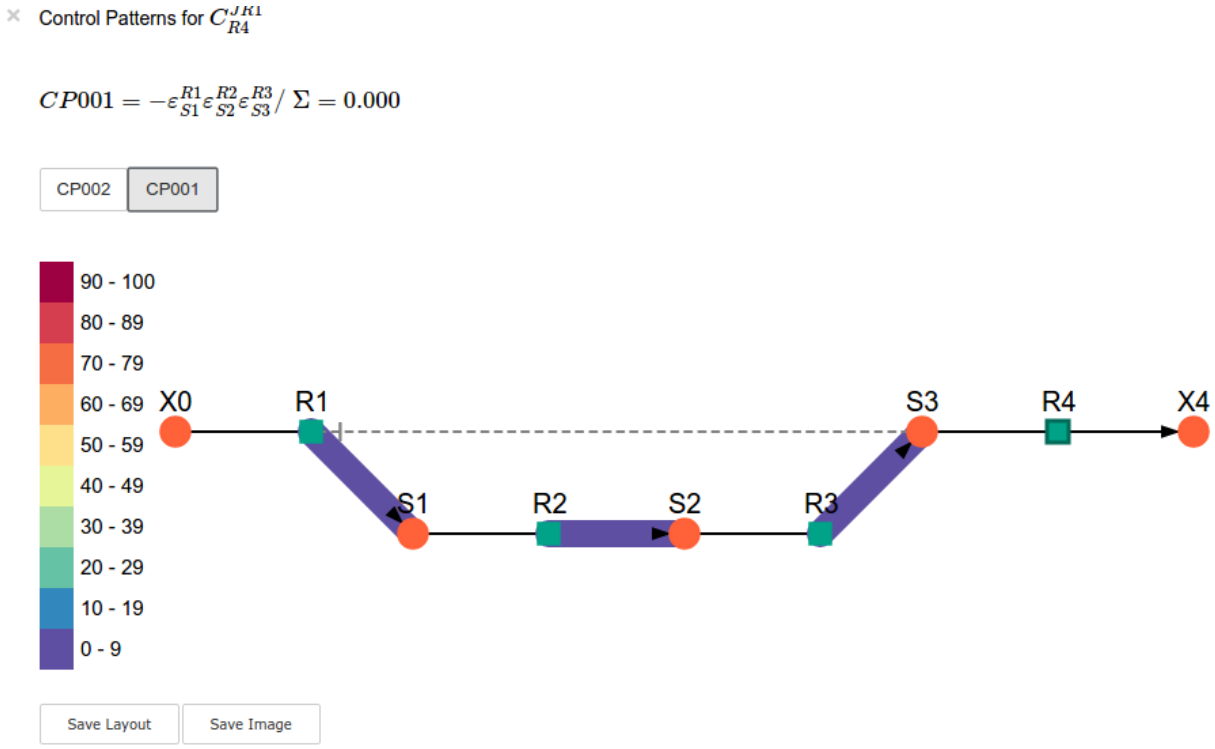
In [20]:

```
# clicking on CP002 shows that this control pattern representing
# the chain of effects passing through the feedback loop
# is totally responsible for the observed control coefficient value.
sc.cc_results.ccJR1_R4.highlight_patterns(height = 350, pos_dic=path_to_layout)
```



In [21]:

```
# clicking on CP001 shows that this control pattern representing
# the chain of effects of the main pathway does not contribute
# at all to the control coefficient value.
sc.cc_results.ccJR1_R4.highlight_patterns(height = 350, pos_dic=path_to_layout)
```



5.2.7 Parameter scans

Parameter scans can be performed in order to determine the effect of a parameter change on either the control coefficient and control pattern values or of the effect of a parameter change on the contribution of the control patterns towards the control coefficient (as discussed above). The procedures for both the "value" and "percentage" scans are very much the same and rely on the same principles as described in the Basic Usage and RateChar sections.

To perform a parameter scan the `do_par_scan` method is called. This method has the following arguments:

- `parameter`: A String representing the parameter which should be varied.
- `scan_range`: Any iterable representing the range of values over which to vary the parameter (typically a NumPy `ndarray` generated by `numpy.linspace` or `numpy.logspace`).
- `scan_type`: Either `"percentage"` or `"value"` as described above (*default*: `"percentage"`).
- `init_return`: If `True` the parameter value will be reset to its initial value after performing the parameter scan (*default*: `True`).
- `par_scan`: If `True`, the parameter scan will be performed by multiple parallel processes rather than a single process, thus speeding performance (*default*: `False`).
- `par_engine`: Specifies the engine to be used for the parallel scanning processes. Can either be `"multiproc"` or `"ipcluster"`. A discussion of the differences between these methods are beyond the scope of this document, see here for a brief overview of Multiprocessing in Python. (*default*: `"multiproc"`).
- `force_legacy`: If `True` `do_par_scan` will use a older and slower algorithm for performing the parameter scan. This is mostly used for debugging purposes. (*default*: `False`)

Below we will perform a percentage scan of V_{f4} for 200 points between 0.01 and 1000 in log space:

```
In [22]:
```

```
percentage_scan_data = sc.cc_results.ccJR1_R4.do_par_scan(parameter='Vf_4',
                                                          scan_range=numpy.logspace(-
↪1,3,200),
                                                          scan_type='percentage')
```

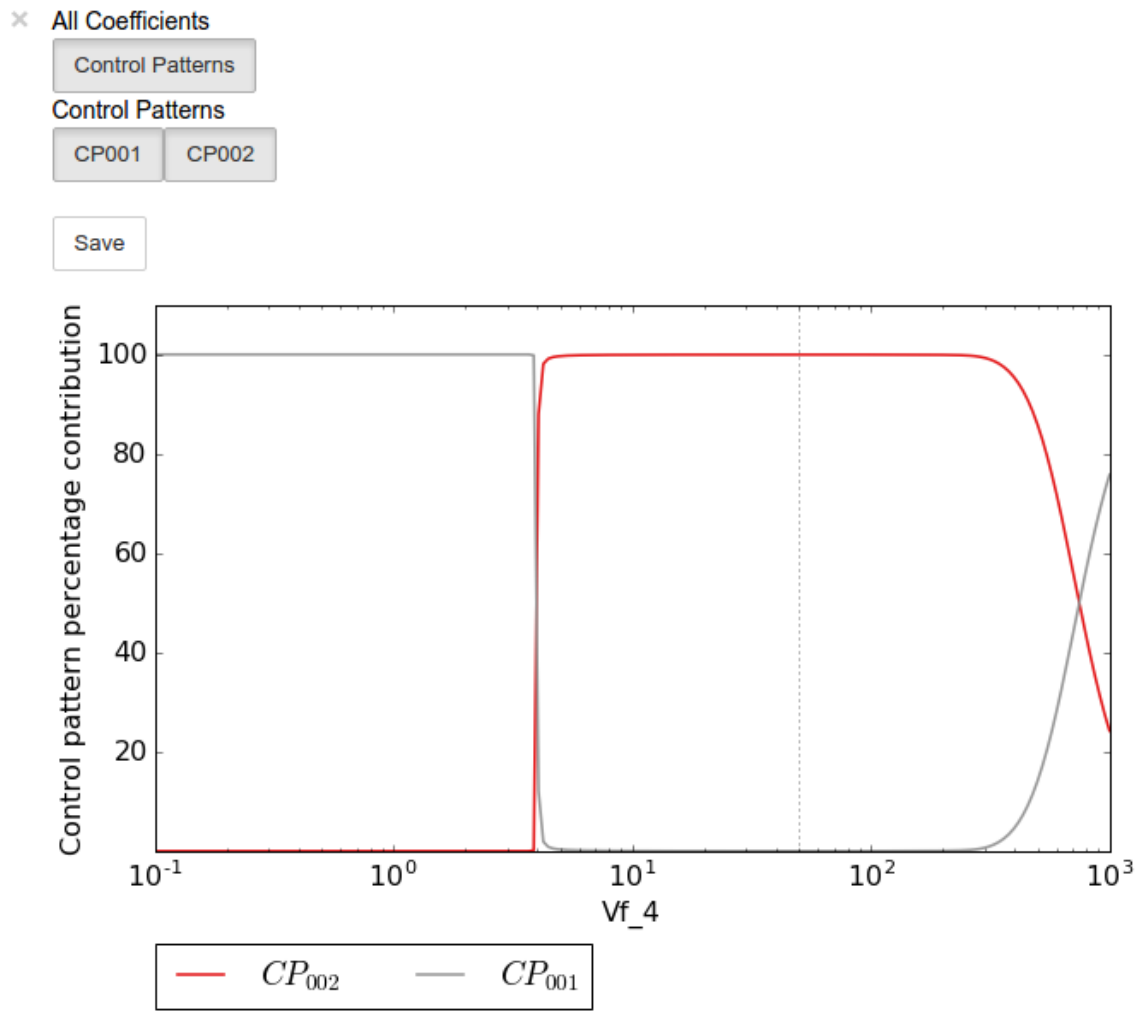
Out[22]:

```
MaxMode 1
0 min 0 sec
SCANNER: Tsteps 200

SCANNER: 200 states analysed

(hybrd) The solution converged.
```

As previously described, these data can be displayed using `ScanFig` by calling the `plot` method of `percentage_scan_data`. Furthermore, lines can be enabled/disabled using the `toggle_category` method of `ScanFig` or by clicking on the appropriate buttons:

In [23]:

```
percentage_scan_plot = percentage_scan_data.plot()

# set the x-axis to a log scale
percentage_scan_plot.ax.semilogx()

# enable all the lines
percentage_scan_plot.toggle_category('Control Patterns', True)
percentage_scan_plot.toggle_category('CP001', True)
percentage_scan_plot.toggle_category('CP002', True)

# display the plot
percentage_scan_plot.interact()
```



A `value` plot can similarly be generated and displayed. In this case, however, an additional line indicating C_4^J will also be present:

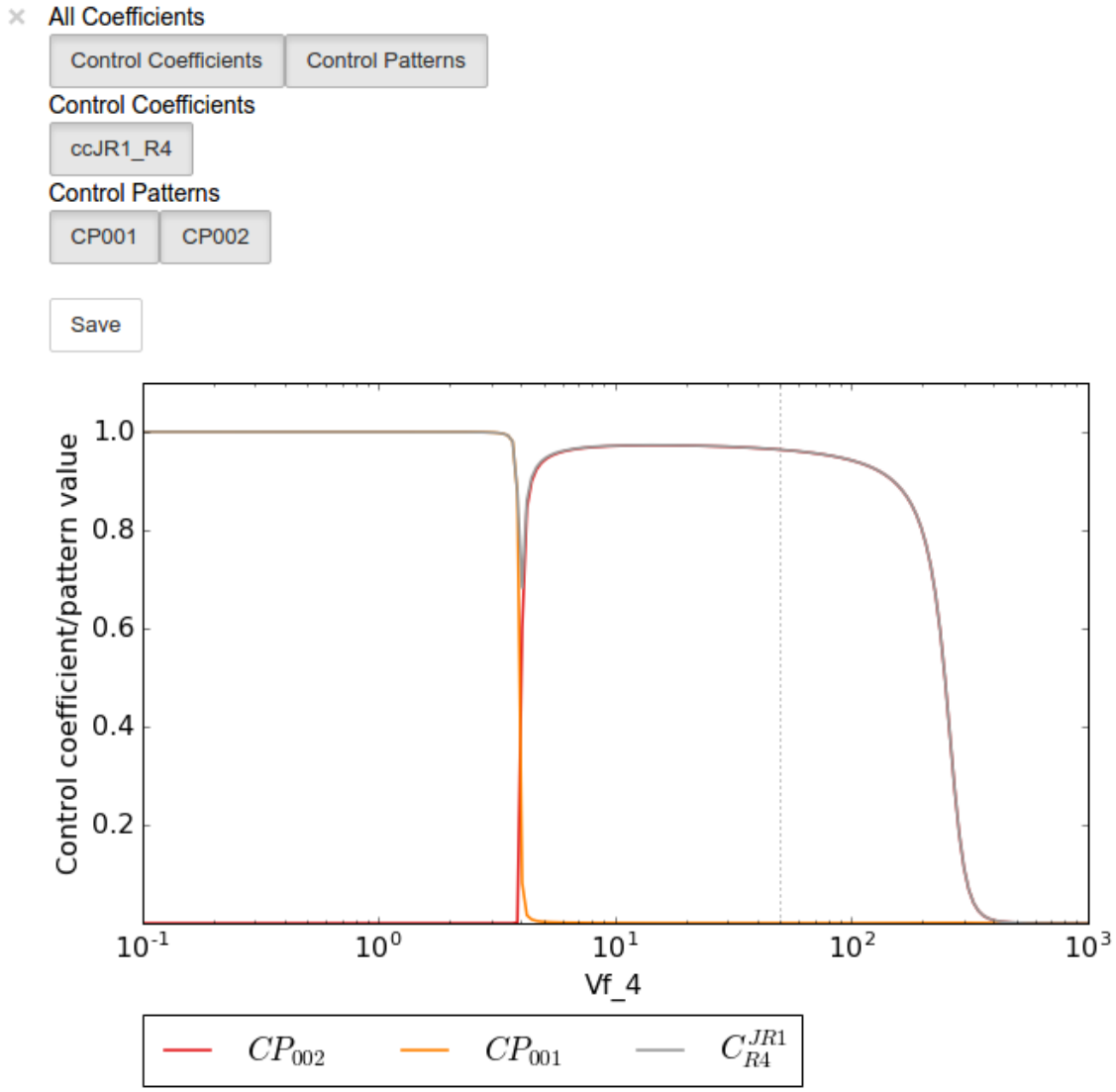
In [24]:

```
value_scan_data = sc.cc_results.ccJR1_R4.do_par_scan(parameter='Vf_4',
                                                     scan_range=numpy.logspace(-1,3,
↪200),
                                                     scan_type='value')

value_scan_plot = value_scan_data.plot()

# set the x-axis to a log scale
value_scan_plot.ax.semilogx()

# enable all the lines
value_scan_plot.toggle_category('Control Coefficients', True)
value_scan_plot.toggle_category('ccJR1_R4', True)

value_scan_plot.toggle_category('Control Patterns', True)
value_scan_plot.toggle_category('CP001', True)
value_scan_plot.toggle_category('CP002', True)
```

(continues on next page)

(continued from previous page)

```
# display the plot
value_scan_plot.interact()
```



5.2.8 Fixed internal metabolites

In the case where the concentration of an internal intermediate is fixed (such as in the case of a GSDA) the `internal_fixed` argument must be set to `True` in either the `do_symca` method, or when instantiating the `Symca` object. This will typically result in the creation of a `cc_results_N` object for each separate reaction block, where `N` is a number starting at 0. Results can then be accessed via these objects as with normal free internal intermediate models.

Thus for a variant of the `lin4_fb` model where the intermediate`S3` is fixed at its steady-state value the procedure is as follows:

`In [25]:`

```
# Create a variant of mod with 'C' fixed at its steady-state value
mod_fixed_S3 = psctb.modeltools.fix_metabolite_ss(mod, 'S3')

# Instantiate Symca object the 'internal_fixed' argument set to 'True'
sc_fixed_S3 = psctb.Symca(mod_fixed_S3,internal_fixed=True)

# Run the 'do_symca' method (internal_fixed can also be set to 'True' here)
sc_fixed_S3.do_symca()
```

```
Out[25]:
```

```
(hybrd) The solution converged.

I hope we have a filebuffer
Seems like it

Reaction stoichiometry and rate equations

Species initial values

Parameters
Assuming extension is .psc
Using model directory: /home/jr/Pysces/psc
Using file: lin4_fb_S3.psc
/home/jr/Pysces/psc/orca/lin4_fb_S3.psc loading .....
Parsing file: /home/jr/Pysces/psc/orca/lin4_fb_S3.psc
Info: "X4" has been initialised but does not occur in a rate equation

Calculating L matrix . . . . . .  done.
Calculating K matrix . . . . . .  done.

(hybrd) The solution converged.
Simplifying matrix with 24 elements
**********
```

The normal `sc_fixed_S3.cc_results` object is still generated, but will be invalid for the fixed model. Each additional `cc_results_N` contains control coefficient expressions that have the same common denominator and corresponds to a specific reaction block. These `cc_results_N` objects are numbered arbitrarily, but consistantly accross different sessions. Each results object accessed and utilised in the same way as the normal `cc_results` object.

For the `mod_fixed_c` model two additional results objects (`cc_results_0` and `cc_results_1`) are generated:

- `cc_results_1` contains the control coefficients describing the sensitivity of flux and concentrations within the supply block of `S3` towards reactions within the supply block.

```
In [26]:
```

```
sc_fixed_S3.cc_results_1
```

C_{R1}^{JR1}	1.000
C_{R2}^{JR1}	8.603e-05
C_{R3}^{JR1}	4.612e-05
C_{R1}^{JR2}	1.000
C_{R2}^{JR2}	8.603e-05
C_{R3}^{JR2}	4.612e-05
C_{R1}^{JR3}	1.000
C_{R2}^{JR3}	8.603e-05
C_{R3}^{JR3}	4.612e-05
C_{R1}^{S1}	0.141

C_{R2}^{S1}	-0.092
C_{R3}^{S1}	-0.049
C_{R1}^{S2}	0.052
C_{R2}^{S2}	4.446e-06
C_{R3}^{S2}	-0.052
Σ	210.616

- `cc_results_0` contains the control coefficients describing the sensitivity of flux and concentrations of either reaction block towards reactions in the other reaction block (i.e., all control coefficients here should be zero). Due to the fact that the `S3` demand block consists of a single reaction, this object also contains the control coefficient of `R4` on `J_R4`, which is equal to one. This results object is useful confirming that the results were generated as expected.

In [27]:

```
sc_fixed_S3.cc_results_0
```

C_{R4}^{JR1}	0.000
C_{R4}^{JR2}	0.000
C_{R4}^{JR3}	0.000
C_{R1}^{JR4}	0.000
C_{R2}^{JR4}	0.000
C_{R3}^{JR4}	0.000
C_{R4}^{JR4}	1.000
C_{R4}^{S1}	0.000
C_{R4}^{S2}	0.000
Σ	1.000

If the demand block of `S3` in this pathway consisted of multiple reactions, rather than a single reaction, there would have been an additional `cc_results_N` object containing the control coefficients of that reaction block.

5.2.9 Saving results

In addition to being able to save parameter scan results (as previously described), a summary of the control coefficient and control pattern results can be saved using the `save_results` method. This saves a `csv` file (by default) to disk to any specified location. If no location is specified, a file named `cc_summary_N` is saved to the `~/Pysces/$modelname/symca/` directory, where `N` is a number starting at 0:

In [28]:

```
sc.save_results()
```

`save_results` has the following optional arguments:

- `file_name`: Specifies a path to save the results to. If `None`, the path defaults as described above.
- `separator`: The separator between fields (*default*: `","`)

The contents of the saved data file is as follows:

In [29]:

```
# the following code requires `pandas` to run
import pandas as pd
# load csv file at default path
results_path = '~/Pysces/lin4_fb/symca/cc_summary_0.csv'

# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    results_path = psctb.utils.misc.unix_to_windows_path(results_path)
else:
    results_path = path.expanduser(results_path)

saved_results = pd.read_csv(results_path)
# show first 20 lines
saved_results.head(n=20)
```

5.2.10 Saving/loading sessions

Saving and loading `Symca` sessions is very simple and works similar to `RateChar`. Saving a session takes place with the `save_session` method, whereas the `load_session` method loads the saved expressions. As with the `save_results` method and most other saving and loading functionality, if no `file_name` argument is provided, files will be saved to the default directory (see also Basic Usage). As previously described, expressions can also automatically be loaded/saved by `do_symca` by using the `auto_save_load` argument which saves and loads using the default path. Models with internal fixed metabolites are handled automatically.

In [30]:

```
# saving session
sc.save_session()

# create new Symca object and load saved results
new_sc = psctb.Symca(mod)
new_sc.load_session()

# display saved results
new_sc.cc_results
```

Out[30]:

```
(hybrd) The solution converged.
```

C_{R1}^{JR1}	0.036
C_{R2}^{JR1}	3.090e-06
C_{R3}^{JR1}	1.657e-06
C_{R4}^{JR1}	0.964
C_{R1}^{JR2}	0.036
C_{R2}^{JR2}	3.090e-06
C_{R3}^{JR2}	1.657e-06
C_{R4}^{JR2}	0.964
C_{R1}^{JR3}	0.036
C_{R2}^{JR3}	3.090e-06

C_{R3}^{JR3}	1.657e-06
C_{R4}^{JR3}	0.964
C_{R1}^{JR4}	0.036
C_{R2}^{JR4}	3.090e-06
C_{R3}^{JR4}	1.657e-06
C_{R4}^{JR4}	0.964
C_{R1}^{S1}	0.323
C_{R2}^{S1}	-0.092
C_{R3}^{S1}	-0.049
C_{R4}^{S1}	-0.182

C_{R1}^{S2}	0.335
C_{R2}^{S2}	2.885e-05
C_{R3}^{S2}	-0.052
C_{R4}^{S2}	-0.284
C_{R1}^{S3}	0.334
C_{R2}^{S3}	2.871e-05
C_{R3}^{S3}	1.539e-05
C_{R4}^{S3}	-0.334
Σ	631.138

CHAPTER 6

Thermokin

Thermokin is used to assess the kinetic and thermodynamic aspects of enzyme catalysed reactions in metabolic pathways [7,8]. It provides the functionality to automatically separate the rate equations of reversible reactions into a *mass-action* (v_{ma}) term and a combined *binding* (v_{Θ}) and *rate capacity* (v_{cap}) term, however rate equations may be manually split into any arbitrary terms if more granularity is required. Additionally Γ/K_{eq} is calculated automatically for reversible reactions. Subsequently, elasticity coefficients for the different rate equation terms are automatically calculated. Similar to symbolic control coefficient and control pattern expressions of Symca, the term and elasticity expressions generated by Thermokin can be inspected and manipulated with standard `SymPy` functionality and their values are automatically recalculated upon a steady-state recalculation.

Note: Here we use the word "term" to refer to the terms of the logarithmic form of a rate equation *as well as to the corresponding factors of its linear (conventional) form*. While not technically correct, this terminology is used in accordance to the original publication [8].

6.1 Features

- Automatically separates rate equations into a mass-action term and a combined binding and rate capacity terms.
- Allows for splitting rate equations into arbitrary terms.
- Determines a Γ/K_{eq} expression for reversible reactions.
- Determines elasticity coefficient expressions for each reaction and its associated terms.
- Calculates values of for reaction rate terms, Γ/K_{eq}, and elasticity coefficients when a new steady-state is reached.
- The effect of a parameter change on the reaction rate terms, Γ/K_{eq}, and elasticity coefficients can be investigated by performing a parameter scan and visualised usig `ScanFig`.
- Loading of split rate equation terms
- Saving of Thermokin results

6.2 Usage and feature walkthrough

6.2.1 Workflow

Assessing the kinetic and thermodynamic aspects of enzyme catalysed reactions using `Thermokin` requires the following steps:

1. Instantiation of a `Thermokin` object using a `PySCeS` model object and (optionally) a file in which the rate equations of the model has been split into separate terms.
2. Accessing rate equation terms via `reaction_results` and the corresponding reaction name, reaction term name, or Γ/K_{eq} name.
3. Accessing elasticity coefficient terms via `ec_results` and the corresponding elasticity coefficient name.
4. Inspection of the values of the various terms and elasticity coefficients.
5. Inspection of the effect of parameter changes on the values of the rate equation terms and elasticity coefficients.
6. Result saving.
7. Further analysis.

6.2.2 Rate term file syntax

As previously mentioned, Thermokin will attempt to automatically split the rate equations of reversible reactions into separate terms. While this feature should work for most common rate equations and does not require any user intervention or knowledge of the parameter names used in the model file, it is limited in two significant ways:

1. The algorithm cannot distinguish between the binding and rate capacity terms and can therefore not separate them. This is a minor issue if the focus of the analysis will be on the elasticity coefficients of the different terms, as the combined rate capacity and binding term elasticity coefficient will be identical to that of the binding term alone.
2. The algorithm cannot separate the effect of single subunit binding from that of cooperative binding.

Additionally, the algorithm can fail in some instances.

For these reasons the separate rate equation terms can be manually defined in a `.reqn` file using a relatively simple syntax. Below follows such a file as automatically generated for the model `lin4_fb.psc`:

```
# Automatically parsed and split rate equations for model: lin4_fb.psc
# generated on: 13:49:07 12-01-2017

# Note that this is a best effort attempt that is highly dependent
# on the form of the rate equations as defined in the model file.
# Check correctness before use.

# R1 :successful separation of rate equation terms
!T{R1}{ma} X0 - S1/Keq_1
!T{R1}{bind_vc} 1.0*Vf_1*(S1/S1_05_1 + X0/X0_05_1)**(h_1 - 1.0)*(a_1*(S3/S3_05_1)**h_
↪1 + 1)/(X0_05_1*(a_1*(S3/S3_05_1)**h_1*(S1/S1_05_1 + X0/X0_05_1)**h_1 + (S3/S3_05_
↪1)**h_1 + (S1/S1_05_1 + X0/X0_05_1)**h_1 + 1))
!G{R1}{gamma_keq} S1/(Keq_1*X0)

# R2 :successful separation of rate equation terms
!T{R2}{ma} S1 - S2/Keq_2
!T{R2}{bind_vc} 1.0*S2_05_2*Vf_2/(S1*S2_05_2 + S1_05_2*S2 + S1_05_2*S2_05_2)
```

(continues on next page)

(continued from previous page)

```
!G{R2}{gamma_keq} S2/(Keq_2*S1)

# R3 :successful separation of rate equation terms
!T{R3}{ma} S2 - S3/Keq_3
!T{R3}{bind_vc} 1.0*S3_05_3*Vf_3/(S2*S3_05_3 + S2_05_3*S3 + S2_05_3*S3_05_3)
!G{R3}{gamma_keq} S3/(Keq_3*S2)

# R4 :rate equation not included - irreversible or unknown form
```

Two types of "terms" can be defined in a `.reqn` file. The first type denoted by `!T`, is factor of the rate equation. When the `!T` terms for a reaction are multiplied together, they should result in the original rate equation.

Secondly `!G` terms are any arbitrary terms that could contain some useful information. Unlike the `!T` terms, the `!G` are not subject to any restrictions in terms of the value of their product or otherwise. For instance, the `!G` terms are used for define Γ/K_{eq} for reversible reactions.

The syntax for `!T` and `!G` terms are as follows:

```
!T{%reaction_name}{%term_name} %term_expression

!G{%reaction_name}{%term_name} %term_expression
```

- `%reaction_name` - The name of the reaction to which the term belongs as defined in the `.psc` file (see the PySCeS MDL documentation).
- `%term_name` - The name of the term. While this name is arbitrary, there can be no duplication for any single reaction.
- `%term_expression` - The expression of the term.

Thus using the example provided above for reaction 3 the line `!T{R3}{ma} S2 - S3/Keq_3` specifies a `!T` term belonging to reaction 3 with the name `ma` and the expression `S2 - S3/Keq_3`.

6.2.3 Object instantiation

Instantiation of a `Thermokin` analysis object requires `PySCeS` model object (`PysMod`) as an argument. Optionally a `.reqn` file can be provided that includes specifically slit rate equations. If path is provided, `Thermokin` will attempt to automatically split the reversible rate equations as described above and save a `.reqn` file at `~/Pysces/psc/%model_name.reqn`. If this file already exists, `ThermiKin` will load it instead. Using the included lin4_fb.psc model a `Thermokin` session is instantiated as follows:

`In [1]:`

```
mod = pysces.model('lin4_fb')
tk = psctb.ThermoKin(mod)
```

`Out[1]:`

```
Assuming extension is .psc
Using model directory: /home/jr/Pysces/psc
/home/jr/Pysces/psc/lin4_fb.psc loading .....
Parsing file: /home/jr/Pysces/psc/lin4_fb.psc
Info: "X4" has been initialised but does not occur in a rate equation

Calculating L matrix . . . . . done.
Calculating K matrix . . . . . done.
```

Now that ThermoKin has automatically generated a .reqn file for lin4_fb.psc, we can load that file manually during instantiation as follows:

In [2]:

```
# This path leads to the provided rate equation file file
path_to_reqn = '~/Pysces/psc/lin4_fb.reqn'

# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    path_to_reqn = psctb.utils.misc.unix_to_windows_path(path_to_reqn)
else:
    path_to_reqn = path.expanduser(path_to_reqn)


tk = psctb.ThermoKin(mod,path_to_reqn)
```

If the path specified does not exist, a new .reqn file will be generated there instead.

Finally, ThermoKin can also be forced to regenerate a the .reqn file by setting the overwrite argument to True:

In [3]:

```
tk = psctb.ThermoKin(mod,overwrite=True)
```

Out[3]:

```
The file /home/jr/Pysces/psc/lin4_fb.reqn will be overwritten with automatically␣
↪generated file.
R1        : successful separation of rate equation terms
R2        : successful separation of rate equation terms
R3        : successful separation of rate equation terms
R4        : rate equation not included - irreversible or unknown form
```

6.2.4 Accessing results

Unlike RateChar and Symca, ThermoKin generates results immediately after instantiation. Results are organised similar to the other two modules, however, and can be found in the reaction_results and ec_results objects:

In [4]:

```
tk.reaction_results
```

J_{R1}	44.618
$J_{R1_{bindvc}}$	44.661
$J_{R1_{gammakeq}}$	9.599e-04
$J_{R1_{ma}}$	0.999
J_{R2}	44.618
$J_{R2_{bindvc}}$	5081.101
$J_{R2_{gammakeq}}$	0.909
$J_{R2_{ma}}$	0.009
J_{R3}	44.618
$J_{R3_{bindvc}}$	1036.279

$J_{R3_{gammakeq}}$	0.951
$J_{R3_{ma}}$	0.043

In [5]:

```
tk.ec_results
```

ε^{R1}_{Keq1}	9.608e-04
ε^{R1}_{S1}	-9.363e-04
ε^{R1}_{S1051}	-2.451e-05
ε^{R1}_{S3}	-2.888
ε^{R1}_{S3051}	2.888
ε^{R1}_{Vf1}	1.000
ε^{R1}_{X0}	3.554
ε^{R1}_{X0051}	-3.553
ε^{R1}_{a1}	0.062
ε^{R1}_{h1}	-1.461

ε^{R2}_{Keq2}	9.931
ε^{R2}_{S1}	10.883
ε^{R2}_{S1052}	-0.951
ε^{R2}_{S2}	-10.374
ε^{R2}_{S2052}	0.443
ε^{R2}_{Vf2}	1.000
ε^{R3}_{Keq3}	19.255
ε^{R3}_{S2}	19.351
ε^{R3}_{S2053}	-0.096
ε^{R3}_{S3}	-19.341

ε^{R3}_{S3053}	0.086
ε^{R3}_{Vf3}	1.000
$\varepsilon^{R1_{bindvc}}_{Keq1}$	0.000
$\varepsilon^{R1_{gammakeq}}_{Keq1}$	-1.000
$\varepsilon^{R1_{ma}}_{Keq1}$	9.608e-04
$\varepsilon^{R1_{bindvc}}_{S1051}$	-2.451e-05
$\varepsilon^{R1_{gammakeq}}_{S1051}$	0.000
$\varepsilon^{R1_{ma}}_{S1051}$	0.000
$\varepsilon^{R1_{bindvc}}_{S1}$	2.451e-05
$\varepsilon^{R1_{gammakeq}}_{S1}$	1.000

$\varepsilon_{S1}^{R1_{ma}}$	-9.608e-04
$\varepsilon_{S3051}^{R1_{bindvc}}$	2.888
$\varepsilon_{S3051}^{R1_{gammakeq}}$	0.000
$\varepsilon_{S3051}^{R1_{ma}}$	0.000
$\varepsilon_{S3}^{R1_{bindvc}}$	-2.888
$\varepsilon_{S3}^{R1_{gammakeq}}$	0.000
$\varepsilon_{S3}^{R1_{ma}}$	0.000
$\varepsilon_{Vf1}^{R1_{bindvc}}$	1.000
$\varepsilon_{Vf1}^{R1_{gammakeq}}$	0.000
$\varepsilon_{Vf1}^{R1_{ma}}$	0.000

$\varepsilon_{X0051}^{R1_{bindvc}}$	-3.553
$\varepsilon_{X0051}^{R1_{gammakeq}}$	0.000
$\varepsilon_{X0051}^{R1_{ma}}$	0.000
$\varepsilon_{X0}^{R1_{bindvc}}$	2.553
$\varepsilon_{X0}^{R1_{gammakeq}}$	-1.000
$\varepsilon_{X0}^{R1_{ma}}$	1.001
$\varepsilon_{a1}^{R1_{bindvc}}$	0.062
$\varepsilon_{a1}^{R1_{gammakeq}}$	0.000
$\varepsilon_{a1}^{R1_{ma}}$	0.000
$\varepsilon_{h1}^{R1_{bindvc}}$	-1.461

$\varepsilon_{h1}^{R1_{gammakeq}}$	0.000
$\varepsilon_{h1}^{R1_{ma}}$	0.000
$\varepsilon_{Keq2}^{R2_{bindvc}}$	0.000
$\varepsilon_{Keq2}^{R2_{gammakeq}}$	-1.000
$\varepsilon_{Keq2}^{R2_{ma}}$	9.931
$\varepsilon_{S1052}^{R2_{bindvc}}$	-0.951
$\varepsilon_{S1052}^{R2_{gammakeq}}$	0.000
$\varepsilon_{S1052}^{R2_{ma}}$	0.000
$\varepsilon_{S1}^{R2_{bindvc}}$	-0.049
$\varepsilon_{S1}^{R2_{gammakeq}}$	-1.000

$\varepsilon_{S1}^{R2_{ma}}$	10.931
$\varepsilon_{S2052}^{R2_{bindvc}}$	0.443
$\varepsilon_{S2052}^{R2_{gammakeq}}$	0.000
$\varepsilon_{S2052}^{R2_{ma}}$	0.000
$\varepsilon_{S2}^{R2_{bindvc}}$	-0.443
$\varepsilon_{S2}^{R2_{gammakeq}}$	1.000
$\varepsilon_{S2}^{R2_{ma}}$	-9.931
$\varepsilon_{Vf2}^{R2_{bindvc}}$	1.000
$\varepsilon_{Vf2}^{R2_{gammakeq}}$	0.000
$\varepsilon_{Vf2}^{R2_{ma}}$	0.000

$\varepsilon_{Keq3}^{R3_{bindvc}}$	0.000
$\varepsilon_{Keq3}^{R3_{gammakeq}}$	-1.000
$\varepsilon_{Keq3}^{R3_{ma}}$	19.255
$\varepsilon_{S2053}^{R3_{bindvc}}$	-0.096
$\varepsilon_{S2053}^{R3_{gammakeq}}$	0.000
$\varepsilon_{S2053}^{R3_{ma}}$	0.000
$\varepsilon_{S2}^{R3_{bindvc}}$	-0.904
$\varepsilon_{S2}^{R3_{gammakeq}}$	-1.000
$\varepsilon_{S2}^{R3_{ma}}$	20.255
$\varepsilon_{S3053}^{R3_{bindvc}}$	0.086

$\varepsilon_{S3053}^{R3_{gammakeq}}$	0.000
$\varepsilon_{S3053}^{R3_{ma}}$	0.000
$\varepsilon_{S3}^{R3_{bindvc}}$	-0.086
$\varepsilon_{S3}^{R3_{gammakeq}}$	1.000
$\varepsilon_{S3}^{R3_{ma}}$	-19.255
$\varepsilon_{Vf3}^{R3_{bindvc}}$	1.000
$\varepsilon_{Vf3}^{R3_{gammakeq}}$	0.000
$\varepsilon_{Vf3}^{R3_{ma}}$	0.000

Each results object contains a variety of fields containing data related to a specific term or expression and may be accessed in a similar way to the results of `Symca`:

- Inspecting an individual reactions, terms, or elasticity coefficient yields a symbolic expression together with a value

In [6]:

```
# The binding*v_cap term of reaction 1
tk.reaction_results.J_R1_bind_vc
```

$$J_{R1_{bindvc}} = \frac{1.0 \cdot Vf_1 \cdot \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1 - 1.0} \cdot \left(a_1 \cdot \left(\frac{S_3}{S_{3051}}\right)^{h_1} + 1\right)}{X_{0051} \cdot \left(a_1 \cdot \left(\frac{S_3}{S_{3051}}\right)^{h_1} \cdot \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1} + \left(\frac{S_3}{S_{3051}}\right)^{h_1} + \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1} + 1\right)} = 44.661$$

- `SymPy` expressions can be accessed via the `expression` field

In [7]:

```
tk.reaction_results.J_R1_bind_vc.expression
```

$$\frac{1.0 Vf_1 \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1 - 1.0} \left(a_1 \left(\frac{S_3}{S_{3051}}\right)^{h_1} + 1\right)}{X_{0051} \left(a_1 \left(\frac{S_3}{S_{3051}}\right)^{h_1} \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1} + \left(\frac{S_3}{S_{3051}}\right)^{h_1} + \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1} + 1\right)}$$

- Values of the reaction, term, or elasticity coefficients

In [8]:

```
tk.reaction_results.J_R1_bind_vc.value
```

Out[8]:

```
44.66092105160845
```

Additionally the `latex_name`, `latex_expression`, and parent model `mod` can also be accessed

In order to promote a logical and exploratory approach to investigating data generated by `ThermoKin`, the results are also arranged in a manner in which terms and elasticity coefficients associated with a certain reaction can be found nested within the results for that reaction. Using reaction 1 (called `J_R1` to signify the fact that its rate is at steady state) as an example, results can also be accessed in the following manner:

In [9]:

```
# The reaction can also be accessed at the root level of the ThermoKin object
# and the binding*v_cap term is nested under it.
tk.J_R1.bind_vc
```

$$J_{R1_{bindvc}} = \frac{1.0 \cdot Vf_1 \cdot \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1-1.0} \cdot \left(a_1 \cdot \left(\frac{S_3}{S_{3051}}\right)^{h_1} + 1\right)}{X_{0051} \cdot \left(a_1 \cdot \left(\frac{S_3}{S_{3051}}\right)^{h_1} \cdot \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1} + \left(\frac{S_3}{S_{3051}}\right)^{h_1} + \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1} + 1\right)} = 44.661$$

In [10]:

```
# A reaction or term specific ec_results object is also available
tk.J_R1.bind_vc.ec_results.pecR1_X0_bind_vc
```

$$\varepsilon_{X0}^{R1_{bindvc}} = -\frac{1.0 \cdot S_{1051} \cdot X_0 \cdot \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{1.0-h_1} \cdot \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1-1.0} \cdot \left(1.0 \cdot a_1 \cdot \left(\frac{S_3}{S_{3051}}\right)^{h_1} \cdot \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1} - 1.0 \cdot h\right.}{(S_1 \cdot X_{0051} + S_{1051} \cdot X_0) \cdot \left(a_1 \cdot \left(\frac{S_3}{S_{3051}}\right)^{h_1} \cdot \left(\frac{S_1}{S_{1051}} + \frac{X_0}{X_{0051}}\right)^{h_1} + \left(\frac{S_3}{S_{3051}}\right.\right.}$$

In [11]:

```
# All the terms of a specific reaction can be accessed via `terms`
tk.J_R1.terms
```

$J_{R1_{bindvc}}$	44.661
$J_{R1_{gammakeq}}$	9.599e-04
$J_{R1_{ma}}$	0.999

While each reaction/term/elasticity coefficient may be accessed in multiple ways, these fields are all references to the same result object. Modifying a term accessed in one way, therefore affects all references to the object.

6.2.5 Dynamic value updating

The values of the reactions/terms/elasticity coefficients are automatically updated when a new steady state is calculated for the model. Thus changing a parameter of `lin4_hill`, such as the V_f value of reaction 3, will lead to new values:

In [12]:

```
# Original value of J_R3
tk.J_R3
```

$$J_{R3} = \frac{1.0 \cdot S_{3053} \cdot Vf_3 \cdot (Keq_3 \cdot S_2 - S_3)}{Keq_3 \cdot (S_2 \cdot S_{3053} + S_{2053} \cdot S_3 + S_{2053} \cdot S_{3053})} = 44.618$$

In [13]:

```
mod.reLoad()
# mod.Vf_3 has a default value of 1000
mod.Vf_3 = 0.1
# calculating new steady state
mod.doState()
```

Out[13]:

```
Parsing file: /home/jr/Pysces/psc/lin4_fb.psc
Info: "X4" has been initialised but does not occur in a rate equation

Calculating L matrix . . . . . .  done.
Calculating K matrix . . . . . .  done.

INFO: (hybrd) Invalid steady state:
(hybrd) The iteration is not making good progress, as measured by the
  improvement from the last ten iterations.
WARNING!! Negative concentrations detected.
INFO: STATE is switching to NLEQ2 solver.
(nleq2) The solution converged.
```

In [14]:

```
# New value (original was 44.618)
tk.J_R3
```

$$J_{R3} = \frac{1.0 \cdot S_{3053} \cdot Vf_3 \cdot (Keq_3 \cdot S_2 - S_3)}{Keq_3 \cdot (S_2 \cdot S_{3053} + S_{2053} \cdot S_3 + S_{2053} \cdot S_{3053})} = 0.100$$

In [15]:

```
# resetting to default Vf_3 value and recalculating
mod.reLoad()
mod.doState()
```

Out[15]:

```
Parsing file: /home/jr/Pysces/psc/lin4_fb.psc
Info: "X4" has been initialised but does not occur in a rate equation

Calculating L matrix . . . . . .  done.
Calculating K matrix . . . . . .  done.

(hybrd) The solution converged.
```

6.2.6 Parameter scans

Parameter scans can be performed in order to determine the effect of a parameter change on a reaction rate and its individual terms or on the elasticity coefficients relating to a particular reaction and its related term elasticity coefficients (denoted as `pec%reaction_%modifier_%term` see Basic Usage - Syntax) . The procedures for both the "value" and "elasticity" scans are very much the same and rely on the same principles as described under Basic Usage - Plotting and Displaying Results.

To perform a parameter scan the `do_par_scan` method is called. This method has the following arguments:

- `parameter`: A String representing the parameter which should be varied.

- `scan_range`: Any iterable representing the range of values over which to vary the parameter (typically a NumPy `ndarray` generated by `numpy.linspace` or `numpy.logspace`).
- `scan_type`: Either `"elasticity"` or `"value"` as described above (*default*: `"value"`).
- `init_return`: If `True` the parameter value will be reset to its initial value after performing the parameter scan (*default*: `True`).
- `par_scan`: If `True`, the parameter scan will be performed by multiple parallel processes rather than a single process, thus speeding performance (*default*: `False`).
- `par_engine`: Specifies the engine to be used for the parallel scanning processes. Can either be `"multiproc"` or `"ipcluster"`. A discussion of the differences between these methods are beyond the scope of this document, see here for a brief overview of Multiprocessing in Python. (*default*: `"multiproc"`).

Below we will perform a value scan of the effect of V_{f^3} on the terms of reaction 1 for 200 points between 0.01 and 100000 in log space:

In [16]:

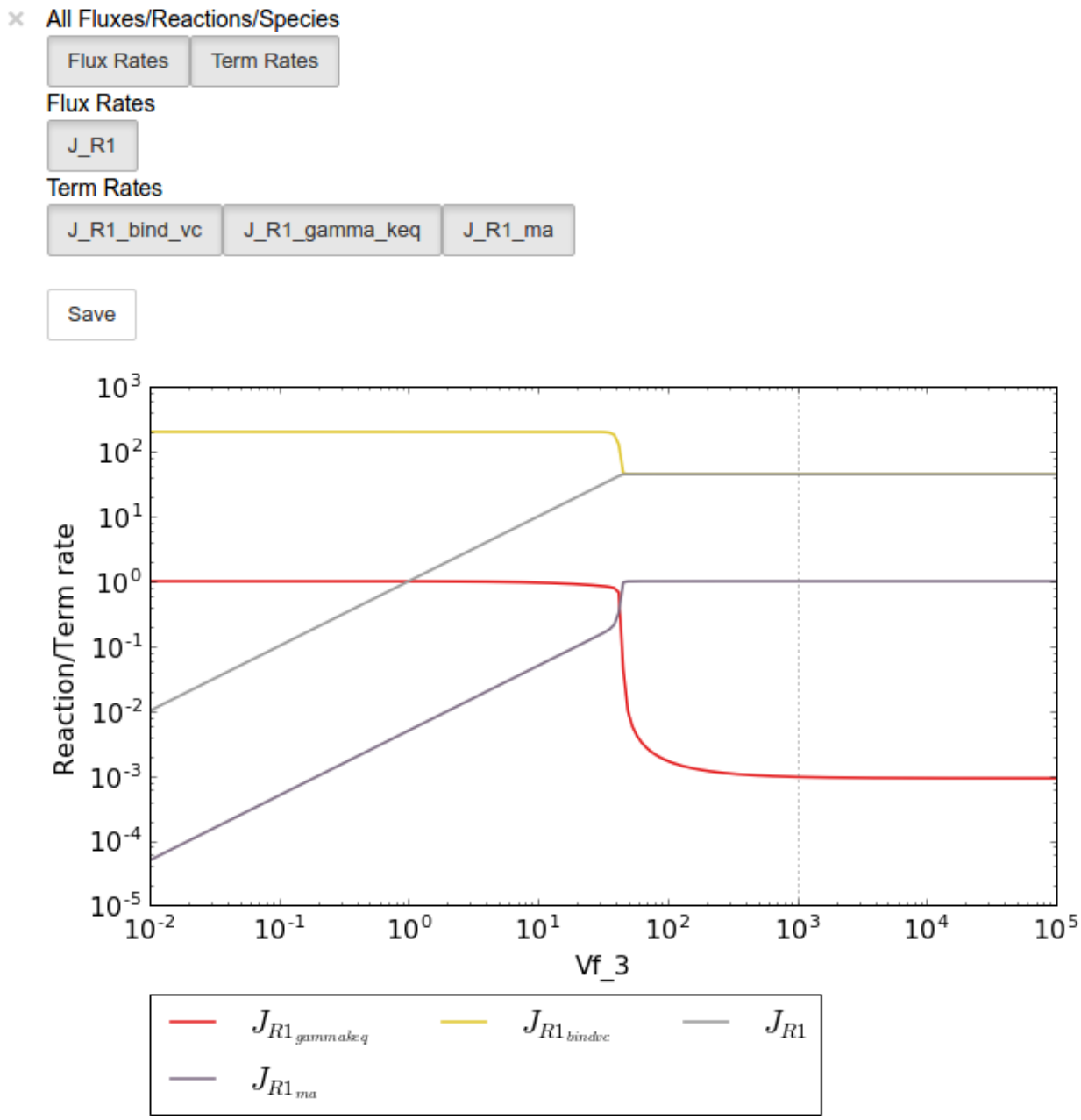
```
valscan = tk.J_R1.do_par_scan('Vf_3',scan_range=numpy.logspace(-2,5,200),scan_type=
↪'value')
```

Out[16]:

```
MaxMode 0
0 min 0 sec
SCANNER: Tsteps 200

SCANNER: 200 states analysed

(hybrd) The solution converged.
```

In [17]:

```
valplot = valscan.plot()

# Equivalent to clicking the corresponding buttons
valplot.toggle_category('J_R1', True)
valplot.toggle_category('J_R1_bind_vc', True)
valplot.toggle_category('J_R1_gamma_keq', True)
valplot.toggle_category('J_R1_ma', True)

valplot.interact()
```



Similarly, we can perform an elasticity scan using the same parameters:

In [18]:

```
ecscan = tk.J_R1.do_par_scan('Vf_3',scan_range=numpy.logspace(-2,5,200),scan_type=
↪'elasticity')
```

Out[18]:

```
MaxMode 0
0 min 0 sec
SCANNER: Tsteps 200

SCANNER: 200 states analysed
```

(continues on next page)

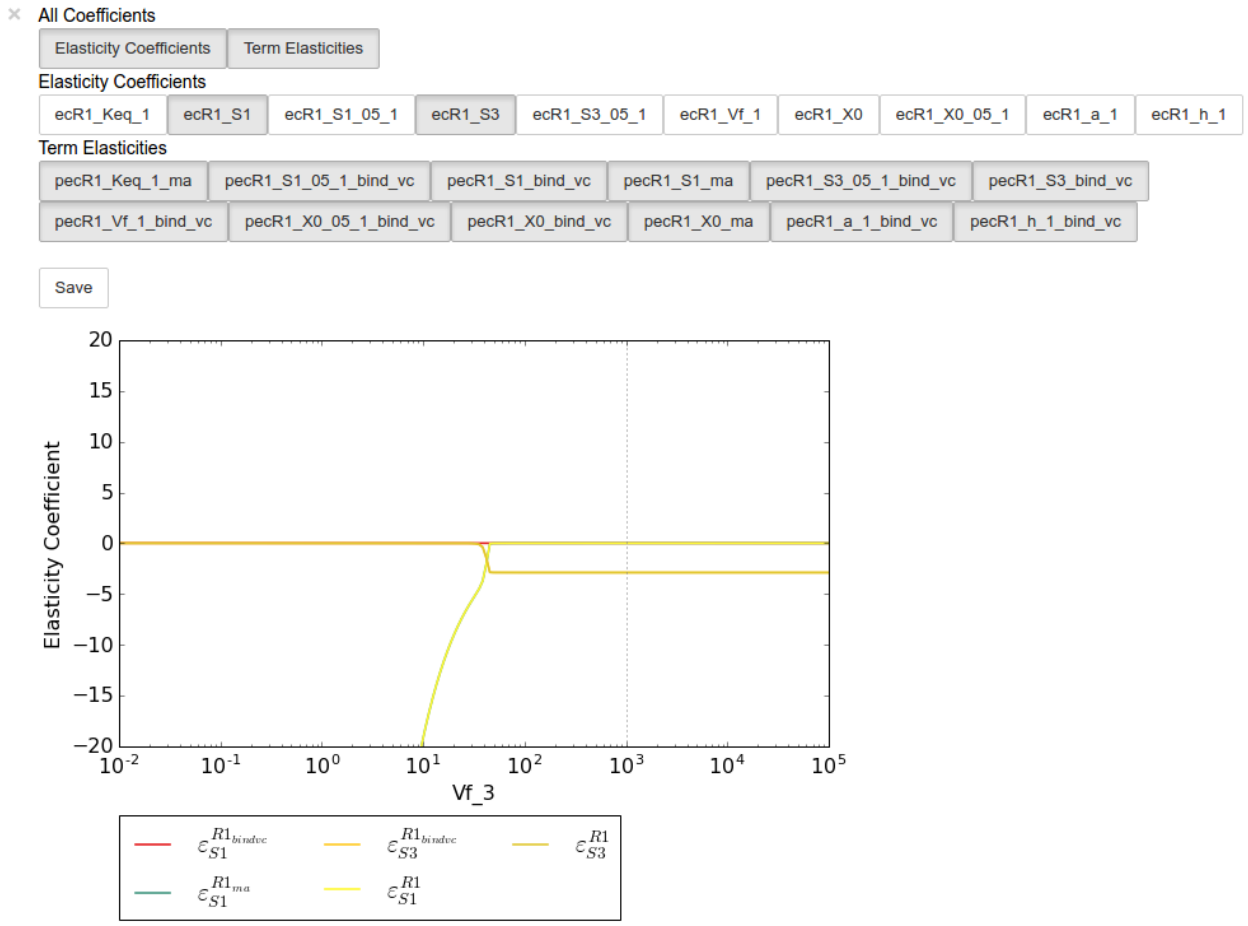
(continued from previous page)

```
(hybrd) The solution converged.
```

Note: Elasticity coefficients with expression equal to zero (which will by definition have zero values regardless of any parameter values) are ommitted from the parameter scan results even though they are included in the `ec_results` objects.

In [19]:

```
ecplot = ecscan.plot()

# All term elasticity coefficients are enabled
# by default, thus only the "full" elasticity
# coefficients need to be enabled. Here we
# switch on the elasticity coefficients
# representing the sensitivity of R1 with
# respect to the substrate S1 and the inhibitor
# S3.
ecplot.toggle_category('ecR1_S1', True)
ecplot.toggle_category('ecR1_S3', True)

# The y limits are adjusted below as the elasticity
# values of this parameter scan have extremely
# large magnitudes at low Vf_3 values
ecplot.ax.set_ylim((-20,20))

ecplot.interact()
```



6.2.7 Saving results

In addition to being able to save parameter scan results (as previously described in Basic Usage - ScanFig), a summary of the results found in `reaction_results` and `ec_results` can be saved using the `save_results` method. This saves a `csv` file (by default) to disk to any specified location. If no location is specified, a file named `tk_summary_N` is saved to the `~/Pysces/$modelname/thermokin/` directory, where `N` is a number starting at 0:

In [20]:

```
tk.save_results()
```

`save_results` has the following optional arguments:

- `file_name`: Specifies a path to save the results to. If `None`, the path defaults as described above.
- `separator`: The separator between fields (*default*: `","`)

The contents of the saved data file is as follows:

In [21]:

```
# the following code requires `pandas` to run
import pandas as pd
# load csv file at default path
results_path = '~/Pysces/lin4_fb/thermokin/tk_summary_0.csv'
```

(continues on next page)

(continued from previous page)

```
# Correct path depending on platform - necessary for platform independent scripts
if platform == 'win32' and pysces.version.current_version_tuple() < (0,9,8):
    results_path = psctb.utils.misc.unix_to_windows_path(results_path)
else:
    results_path = path.expanduser(results_path)

saved_results = pd.read_csv(results_path)


# show first 20 lines
saved_results.head(n=20)
```

CHAPTER 7

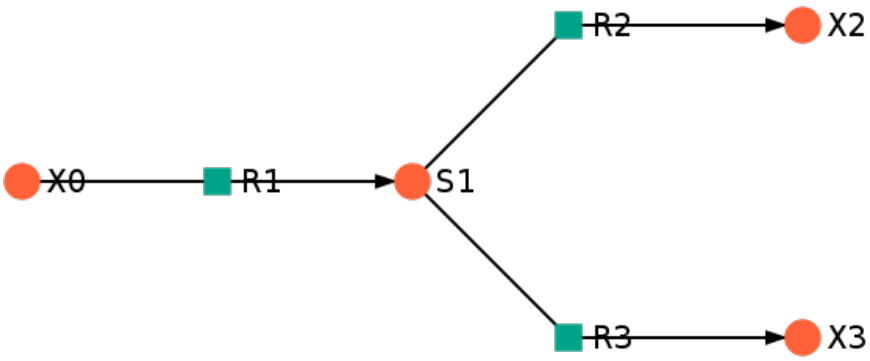
Included Files

Here are the files that are used in the examples as well as the interactive notebook versions of the documentation.

7.1 Models

The models used in this documentation are included below (together with other additional files). These files must be downloaded to the `psc` directory to be used in the example notebooks unless otherwise specified.

7.1.1 example_model.psc



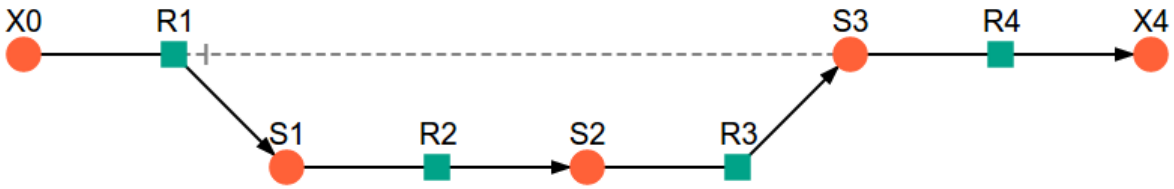
`model`

layout file

The text of example_model.psc is included below:

```
# example_model.psc
# ----------------------------------------------------------------------------
# Fixed Species

FIX: X0 X2 X3

# ----------------------------------------------------------------------------
# Reaction definitions

R1:
    X0 = S1
    ((Vf1 / Km1_X0) * (X0 - S1 / Keq1)) / (1 + X0/Km1_X0 + S1/Km1_S1)

R2:
    S1 = X2
    ((Vf2 / Km2_S1) * (S1 - X2 / Keq2)) / (1 + S1/Km2_S1 + X2/Km2_X2)

R3:
    S1 = X3
    ((Vf3 / Km3_S1) * (S1 - X3 / Keq3)) / (1 + S1/Km3_S1 + X3/Km3_X3)

# ----------------------------------------------------------------------------
# Variable species initial concentrations

S1 = 1

# ----------------------------------------------------------------------------
# Fixed species concentrations

X0 = 100
X2 = 10
X3 = 1

# ----------------------------------------------------------------------------
# Parameters

Vf1 = 100.0
Keq1 = 10.0
Km1_X0 = 1.0
Km1_S1 = 1.0

Vf2 = 50.0
Keq2 = 10.0
Km2_S1 = 1.0
Km2_X2 = 1.0

Vf3 = 10.0
Keq3 = 10.0
Km3_S1 = 1.0
Km3_X3 = 1.0
# ----------------------------------------------------------------------------
```

7.1.2 lin4_fb.psc



`model`

`layout file`

`separated rate equations file`

The text of `lin4_fb.psc` is included below:

```
# lin4_fb.psc
# ----------------------------------------------------------------------------
# Fixed Species

FIX: X0 X4

# ----------------------------------------------------------------------------
# Reaction definitions

R1:
    X0 = S1
    (Vf_1 * (X0 / X0_05_1) * (1 - ((S1/X0)/Keq_1)) *
    (X0/X0_05_1 + S1/S1_05_1)**(h_1-1)) /
    ((X0/X0_05_1 + S1/S1_05_1)**(h_1) +
    (1 + (S3/S3_05_1)**(h_1))/(1 + a_1 * (S3/S3_05_1)**(h_1)))

R2:
    S1 = S2
    (Vf_2 * (S1 / S1_05_2) *
    (1 - ((S2/S1)/Keq_2))) / (1 + S1/S1_05_2 + S2/S2_05_2)

R3:
    S2 = S3
    (Vf_3 * (S2 / S2_05_3) *
    (1 - ((S3/S2)/Keq_3))) / (1 + S2/S2_05_3 + S3/S3_05_3)

R4:
    S3 = X4
    (Vf_4*S3)/(S3 + S3_05_4)

# ----------------------------------------------------------------------------
# Variable species initial concentrations

S1 = 1
S2 = 1
S3 = 1
```

(continues on next page)

(continued from previous page)

```
# ----------------------------------------------------------------------------
# Fixed species concentrations

X0 = 1
X4 = 1

# ----------------------------------------------------------------------------
# Parameters

Vf_1 = 400.0
Keq_1 = 100.0
X0_05_1 = 1.0
S1_05_1 = 10000.0
h_1 = 4
S3_05_1 = 5.0
a_1 = 0.01

Vf_2 = 10000.0
Keq_2 = 10.0
S1_05_2 = 1.0
S2_05_2 = 1.0

Vf_3 = 1000.0
Keq_3 = 10.0
S2_05_3 = 0.01
S3_05_3 = 1.0

Vf_4 = 50.0
S3_05_4 = 1.0

# ----------------------------------------------------------------------------
```

7.2 Example Notebooks

The example Jupyter notebooks are runnable versions of the pages Basic Usage, RateChar, Symca and Thermokin found in this documentation.

`basic_usage.ipynb`

`RateChar.ipynb`

`Symca.ipynb`

`Thermokin.ipynb`

CHAPTER 8

References

[1] Olivier, B. G., Rohwer, J. M. & Hofmeyr, J.-H. S. Modelling cellular systems with PySCeS *Bioinformatics*, 2005, 21, 560-561

[2] Christensen, C. D., Hofmeyr, J.-H. S. & Rohwer, J. M. PySCeSToolbox: a collection of metabolic pathway analysis tools *Bioinformatics*, 2018, 34, 124-125

[3] Hofmeyr, J.-H. S. Control-pattern analysis of metabolic pathways *Eur. J. Biochem.*, 1989, 186, 343-354

[4] Hofmeyr, J.-H. S. Metabolic control analysis in a nutshell *In:* Yi, T.-M., Hucka, M., Morohashi, M. & Kitano, H. (Eds.) *Proceedings of the 2nd International Conference on Systems Biology*, Omnipress, Madison, WI, USA, 2001, pp. 291-300

[5] Hofmeyr, J.-H. S. & Cornish-Bowden, A. Regulating the cellular economy of supply and demand *FEBS Lett.*, 2000, 476, 47-51

[6] Rohwer, J. M. & Hofmeyr, J.-H. S. Identifying and characterising regulatory metabolites with generalised supply-demand analysis *J. Theor. Biol.*, 2008, 252, 546-554

[7] Hofmeyr, Jan-Hendrik. S. Metabolic regulation: A control analytic perspective *J. Bioenerg. Biomembr.*, 1995, 27, 479-490

[8] Rohwer, J. M. & Hofmeyr, J.-H. S. Kinetic and thermodynamic aspects of enzyme control and regulation *J. Phys. Chem. B*, 2010, 114, 16280-16289

CHAPTER 9

Module reference

9.1 psctb package

9.1.1 Subpackages

psctb.analyse package

Subpackages

psctb.analyse._symca package

Submodules

psctb.analyse._symca._symca module

class psctb.analyse._symca._symca.**Symca**(*mod*, *auto_load=False*, *internal_fixed=False*, *ignore_steady_state=False*, *keep_zero_elasticities=True*)

Bases: object

A class that performs Symbolic Metabolic Control Analysis.

This class takes pysces model as an input and performs symbolic inversion of the E matrix using Sympy by calculating the determinant and adjoint matrices of this E matrix.

Parameters

mod [PysMod] The pysces model on which to perform symbolic control analysis.

auto_load [boolean] If true

Returns

———

Attributes

ematrix

esL

es_matrix

fluxes

fluxes_dependent

fluxes_independent

kmatrix

lmatrix

nmatrix

num_ind_fluxes

num_ind_species

scaled_k

scaled_k0

scaled_l

scaled_l0

species

species_dependent

species_independent

subs_fluxes

Methods

do_symca	
load_session	
path_to	
save_results	
save_session	

do_symca(*internal_fixed=None*, *auto_save_load=False*)

ematrix

esL

es_matrix

fluxes

fluxes_dependent

fluxes_independent

kmatrix

lmatrix

load_session(*file_name=None*)

nmatrix

num_ind_fluxes

num_ind_species

path_to(*path*)

save_results(*file_name=None*, *separator=', '*, *fmt='%.9f'*)

save_session(*file_name=None*)

scaled_k

scaled_k0

scaled_l

scaled_l0

species

species_dependent

species_independent

subs_fluxes

psctb.analyse._symca.ccobjects module

class psctb.analyse._symca.ccobjects.**CCBase**(*mod*, *name*, *expression*, *ltxe*)
Bases: object

The base object for the control coefficients and control patterns

Attributes

latex_expression

latex_name

value The value property.

latex_expression

latex_name

value
The value property. Calls self._calc_value() when self._value is None and returns self._value

class psctb.analyse._symca.ccobjects.**CCoef**(*mod*, *name*, *expression*, *denominator*, *ltxe*)
Bases: *psctb.analyse._symca.ccobjects.CCBase*

The object the stores control coefficients. Inherits from CCBase

Attributes

abs_value

latex_expression

latex_expression_full

latex_name

latex_numerator

value The value property.

Methods

do_par_scan	
highlight_patterns	

abs_value

do_par_scan(*parameter*, *scan_range*, *scan_type='percentage'*, *init_return=True*, *par_scan=False*, *par_engine='multiproc'*, *force_legacy=False*)

highlight_patterns(*width=None*, *height=None*, *show_dummy_sinks=False*, *show_external_modifier_links=False*, *pos_dic=None*)

latex_expression

latex_expression_full

latex_name

latex_numerator

class psctb.analyse._symca.ccobjects.**CPattern**(*mod*, *name*, *expression*, *denominator*, *parent*, *ltxe*)

Bases: *psctb.analyse._symca.ccobjects.CCBase*

docstring for CPattern

Attributes

latex_expression

latex_expression_full

latex_name

latex_numerator

percentage

value The value property.

latex_expression

latex_expression_full

latex_name

latex_numerator

percentage

psctb.analyse._symca.ccobjects.**cctype**(*obj*)

psctb.analyse._symca.ccobjects.**get_state**(*mod*, *do_state=False*)

psctb.analyse._symca.symca_toolbox module

class psctb.analyse._symca.symca_toolbox.**SymcaToolBox**

Bases: object

The class with the functions used to populate SymcaData. The project is structured in this way to abstract the 'work' needed to build the various matrices away from the SymcaData class.

Methods

adjugate_matrix(matrix)	Returns the adjugate matrix which is the transpose of the cofactor matrix.
build_cc_matrix(j, jind, sind, jdep, sdep)	Produces the matrices j_cci, j_ccd, s_cci and s_ccd which holds the symbols for the independent and dependent flux control coefficients and the independent and dependent species control coefficients respectively
det_bareis(matrix)	Adapted from original det_bareis function in Sympy 0.7.3.
get_es_matrix(mod, nmatrix, fluxes, species)	Gets the esmatrix.
get_es_matrix_no_mca(mod, nmatrix, fluxes, ...)	Gets the esmatrix.
get_fluxes_vector(mod)	Gets the dependent and independent fluxes (in the correct order)
get_nmatrix(mod)	Returns a sympy matrix made from the N matrix in a Pysces model where the elements are in the same order as they appear in the k and l matrices in pysces.
get_species_vector(mod)	Returns a vector (sympy matrix) with the species in the correct order
invert(matrix, path_to)	Returns the numerators of the inverted martix separately from the common denominator (the determinant of the matrix)
maxima_factor(expression, path_to)	This function is equivalent to the sympy.cancel() function but uses maxima instead
scale_matrix(all_elements, mat, inds)	Scales the k or l matrix.
simplify_matrix(matrix)	Replaces floats with ints and puts elements with fractions on a single demoninator.
solve_dep(cc_i_num, scaledk0, scaledl0, ...)	Calculates the dependent control matrices from the independent control matrix CC_i_solution
substitute_fluxes(all_fluxes, kmatrix)	Substitutes equivalent fluxes in the kmatrix (e.i.

build_inner_dict	
build_outer_dict	
fix_expressions	
generic_populate	
get_fix_denom	
get_fix_denom_jannie	
get_num_ind_fluxes	
get_num_ind_species	
make_CC_dot_dict	
make_inner_dict	
make_internals_dict	
populate_with_fake_elasticities	
populate_with_fake_fluxes	
populate_with_fake_ss_concentrations	
spawn_cc_objects	

static adjugate_matrix(*matrix*)
Returns the adjugate matrix which is the transpose of the cofactor matrix.

Contains code adapted from sympy. Specifically:

cofactorMatrix() minorEntry() minorMatrix() cofactor()

static build_cc_matrix(*j*, *jind*, *sind*, *jdep*, *sdep*)
Produces the matrices j_cci, j_ccd, s_cci and s_ccd which holds the symbols for the independent and dependent flux control coefficients and the independent and dependent species control coefficients respectively

static build_inner_dict(*cc_object*)

static build_outer_dict(*symca_object*)

static det_bareis(*matrix*)
Adapted from original det_bareis function in Sympy 0.7.3. cancel() and expand() are removed from function to speed up calculations. Maxima will be used to simplify the result

Original docstring below:

Compute matrix determinant using Bareis' fraction-free algorithm which is an extension of the well known Gaussian elimination method. This approach is best suited for dense symbolic matrices and will result in a determinant with minimal number of fractions. It means that less term rewriting is needed on resulting formulae.

static fix_expressions(*cc_num*, *common_denom_expr*, *lmatrix*, *species_independent*, *species_dependent*)

static generic_populate(*mod*, *function*, *value=1*)

static get_es_matrix(*mod*, *nmatrix*, *fluxes*, *species*)
Gets the esmatrix.

Goes down the columns of the nmatrix (which holds the fluxes) to get the rows of the esmatrix.

Nested loop goes down the rows of the nmatrix (which holds the species) to get the columns of the esmatrix

so the format is

ecReationN0_M0 ecReationN0_M1 ecReationN0_M2 ecReationN1_M0 ecReationN1_M1 ecReationN1_M2 ecReationN2_M0 ecReationN2_M1 ecReationN2_M2

static get_es_matrix_no_mca(*mod*, *nmatrix*, *fluxes*, *species*)
Gets the esmatrix.

Goes down the columns of the nmatrix (which holds the fluxes) to get the rows of the esmatrix.

Nested loop goes down the rows of the nmatrix (which holds the species) to get the columns of the esmatrix

so the format is

ecReationN0_M0 ecReationN0_M1 ecReationN0_M2 ecReationN1_M0 ecReationN1_M1 ecReationN1_M2 ecReationN2_M0 ecReationN2_M1 ecReationN2_M2

static get_fix_denom(*lmatrix*, *species_independent*, *species_dependent*)

get_fix_denom_jannie(*species_independent*, *species_dependent*)

static get_fluxes_vector(*mod*)
Gets the dependent and independent fluxes (in the correct order)

static get_nmatrix(*mod*)
Returns a sympy matrix made from the N matrix in a Pysces model where the elements are in the same order as they appear in the k and l matrices in pysces.

We need this to make calculations easier later on.

static get_num_ind_fluxes(*mod*)

static get_num_ind_species(*mod*)

static get_species_vector(*mod*)
Returns a vector (sympy matrix) with the species in the correct order

static invert(*matrix*, *path_to*)
Returns the numerators of the inverted martix separately from the common denominator (the determinant of the matrix)

static make_CC_dot_dict(*cc_objects*)

static make_inner_dict(*cc_container*, *cc_container_name*)

static make_internals_dict(*cc_sol*, *cc_names*, *common_denom_expr*, *path_to*)

static maxima_factor(*expression*, *path_to*)
This function is equivalent to the sympy.cancel() function but uses maxima instead

static populate_with_fake_elasticities(*mod*)

static populate_with_fake_fluxes(*mod*)

static populate_with_fake_ss_concentrations(*mod*)

static scale_matrix(*all_elements*, *mat*, *inds*)
Scales the k or l matrix.

The procedure is the same for each matrix: (D^x)^(-1) * y * D^(x_i)

Inverse diagonal The matrix to be The diagonal of of the x where scaled. i.e. the the independent x x is either the k or l matrix where x is the species or the species or the fluxes fluxes

static simplify_matrix(*matrix*)
Replaces floats with ints and puts elements with fractions on a single demoninator.

static solve_dep(*cc_i_num*, *scaledk0*, *scaledl0*, *num_ind_fluxes*, *path_to*)
Calculates the dependent control matrices from the independent control matrix CC_i_solution

static spawn_cc_objects(*mod*, *cc_names*, *cc_sol*, *common_denom_exp*, *ltxe*)

static substitute_fluxes(*all_fluxes*, *kmatrix*)
Substitutes equivalent fluxes in the kmatrix (e.i. dependent fluxes with independent fluxes or otherwise equal fluxes)

Module contents

Submodules

psctb.analyse._ratechar module

class psctb.analyse._ratechar.**RateChar**(*mod*, *min_concrange_factor=100*, *max_concrange_factor=100*, *scan_points=256*, *auto_load=False*)

Bases: object

Methods

do_ratechar	
load_session	
save_results	
save_session	

do_ratechar(*fixed='all'*, *scan_min=None*, *scan_max=None*, *min_concrange_factor=None*, *max_concrange_factor=None*, *scan_points=None*, *solver=0*, *auto_save=False*)

load_session(*file_name=None*)

save_results(*folder=None*, *separator=', '*, *format='%f'*)

save_session(*file_name=None*)

psctb.analyse._thermokin module

class psctb.analyse._thermokin.**ThermoKin**(*mod*, *path_to_reqn_file=None*, *overwrite=False*, *warnings=True*, *ltxe=None*)

Bases: object

Methods

save_results	

save_results(*file_name=None*, *separator=', '*, *fmt='%.9f'*)

psctb.analyse._thermokin_file_tools module

exception psctb.analyse._thermokin_file_tools.**FormatException**

Bases: Exception

psctb.analyse._thermokin_file_tools.**check_for_negatives**(*terms*)

Returns *True* for a list of sympy expressions contains any expressions that are negative.

Parameters

terms [list of sympy expressions] A list where expressions may be either positive or negative.

Returns

bool *True* if any negative terms in expression. Otherwise *False*

psctb.analyse._thermokin_file_tools.**check_term_format**(*lines*, *term_type*)

Inspects a list of string for the correct ThermoKin syntax. Returns *True* in case of correct format. Throws exception otherwise.

Correct format is a str matching the pattern "X{w*}{w*} .*" . Where "X" is either "!G" or "!T" as specified by *term_type*.

Parameters

lines [list of str] Clean list of lines from a '.reqn' file.

term_type [str] This string specifies the type of term.

Returns

bool

psctb.analyse._thermokin_file_tools.**construct_dict**(*lines*)
Constructs a dictionary of dictionaries for each reaction.

Here keys of the outer dictionary is reaction name strings while the inner dictionary keys are the term names. The inner dictionary values are the term expressions

Parameters

lines [list of str]

Returns

dict of str:{str:str}

psctb.analyse._thermokin_file_tools.**create_gamma_keq_reqn_data**(*mod*)

psctb.analyse._thermokin_file_tools.**create_reqn_data**(*mod*)

psctb.analyse._thermokin_file_tools.**filter_irreversible**(*sympy_terms*)

psctb.analyse._thermokin_file_tools.**get_all_terms**(*path_to_read*)

psctb.analyse._thermokin_file_tools.**get_binding_vc_terms**(*sympy_formulas*, *ma_terms*)
Returns dictionary with a combined "rate capacity" and "binding" term as values.

Uses the symbolic rate equations dictionary and mass action term dictionaries to construct a new dictionary with "rate capacity- binding" terms. The symbolic rate equations are divided by their mass action terms. The results are the "rate capacity-binding" terms. This use case requires reaction names as they appear in pysces as keys for both dictionaries.

Parameters

sympy_formulas [dict of str:sympy expression] Full rate equations for all reactions in model. Keys are reaction names and correspond to this in *ma_terms*.

ma_terms [dict of str:sympy expression] Mass action terms for all reactions in model. Keys are reaction names and correspond to this in *sympy_formulas*.

Returns

dict of str:sympy expression A dictionary with reaction names as keys and sympy expressions representing "rate capacity-binding" terms as values.

psctb.analyse._thermokin_file_tools.**get_gamma_keq_terms**(*mod*, *sympy_terms*)

psctb.analyse._thermokin_file_tools.**get_ma_terms**(*mod*, *sympy_terms*)
Returns dict with reaction names as keys and mass action terms as values from a dict with reaction names as keys and lists of sympy expressions as values.

Only reversible reactions are handled. Any list in the `sympy_terms` dict that does not have a length of 2 will be ignored.

Parameters

mod [PysMod] The model from which the *sympy_terms* dict was originally constructed.

sympy_terms: dict of str:list of sympy expressions This dictionary should be created by *get_sympy_terms*.

Returns

dict of str:sympy expression Each value will be a mass action term for each reaction key with a form depending on reversibility as described above.

See also:

get_st_pt_keq

get_sympy_terms

sort_terms

psctb.analyse._thermokin_file_tools.**get_reqn_path**(*mod*)

Gets the default path and filename of'.reqn' files belonging to a model

The *.reqn* files which contain rate equations split into different (arbitrary) components should be saved in the same directory as the model file itself by default. It should have the same filename (sans extension) as the model file.

Parameters

mod [PysMod] A pysces model which has corresponding *.reqn* file saved in the same directory with the same file name as the model file.

Returns

str A sting with the path and filename of the *.reqn* file.

psctb.analyse._thermokin_file_tools.**get_st_pt_keq**(*expression*, *substrates*, *products*)

Takes an expression representing "substrates/products * Keq_expression" and returns substrates, products and keq_expression separately.

Parameters

expression [sympy expression] The expression containing "substrates/products * Keq_expression"

substrates [list of sympy symbols] List with symbolic representations for each substrate involved in the reaction which *expression* represents.

products [list of sympy symbols] List with symbolic representations for each product involved in the reaction which *expression* represents. Returns

——-

tuple of sympy expressions and int This tuple contains sympy expressions for the substrates, products and keq_expression in that order. The final value will be an int which indicates the strategy followed.

See also:

st_pt_keq_from_expression

psctb.analyse._thermokin_file_tools.**get_str_formulas**(*mod*)

Returns a dictionary with reaction_name:string_formula as key:value pairs.

Goes through mod.reactions and constructs a dictionary where reaction_name is the key and mod.reaction_name.formula is the value.

Parameters

mod [PysMod] The model which will be used to construct the dictionary

Returns

dict of str:str A dictionary with reaction_name:string_formula as key:value pairs

psctb.analyse._thermokin_file_tools.**get_subs_dict**(*expression*, *mod*)
Builds a substitution dictionary of an expression based of the values of these symbols in a model.

Parameters

expression [sympy expression]

mod [PysMod]

Returns

dict of sympy.Symbol:float

psctb.analyse._thermokin_file_tools.**get_sympy_formulas**(*str_formulas*)
Converts dict with str values to sympy expression values.

Used to convert key:string_formula to key:sympy_formula. Intended use case is for automatic separation of rate equation terms into mass action and binding terms. This use case requires reaction names as they appear in pysces as keys.

Parameters

str_formulas [dict of str:str] Dictionary with str values that represent reaction expressions. This dictionary needs to have already passed through all sanitising functions/methods (e.g. *replace_pow*).

Returns

dict with sympy_expression values and original keys Dictionary where values are symbolic sympy expressions

psctb.analyse._thermokin_file_tools.**get_sympy_terms**(*sympy_formulas*)
Converts a dict with sympy expressions as values to a new dict with list values containing either the original expression or a negative and a positive expressions.

This is used to separate reversible and irreversible reactions. Reversible reactions will have two terms, one negative and one positive. Here expressions are expanded and split into terms and tested for the above criteria: If met the dict value will be a list of two expressions, each representing a term of the rate equation. Otherwise the dict value will be a list with a single item - the original expression. This use case requires reaction names as they appear in pysces as keys.

Parameters

sympy_formulas [dict of str:sympy expression values] Dictionary with values representing rate equations as sympy expressions. Keys are reaction names

Returns

dict of str:list sympy expression Each list will have either have one item, the original dict value OR two items -the original dict value split into a negative and positive expression.

See also:

check_for_negatives

psctb.analyse._thermokin_file_tools.**get_term_dict**(*raw_lines*, *term_type*)
Returns the term dictionary from a list of raw lines from a file.

The contents of a '.reqn' file is read and passed to this function. Here the contents is parsed and 'main terms' are extracted and returned as a dict of str:{str:str}.

Parameters

raw_lines [list of str] List of lines from a '.reqn' file.

Returns

dict of str:{str:str}

psctb.analyse._thermokin_file_tools.**get_term_types_from_raw_data**(*raw_data_dict*)

Determines the types of terms defined for ThermoKin based on the file contents. This allows for generation of latex expressions based on these terms.

Parameters

raw_data_dict [dict of str:{str:str}]

Returns

set of str

psctb.analyse._thermokin_file_tools.**get_terms**(*raw_lines*, *term_type*)

Takes a list of strings and returns a new list containing only lines starting with *term_type* and strips line endings.

Term can be either of the "main" (or *!T*) type or additional (or *!G*) type

Parameters

raw_lines [list of str] List of lines from a '.reqn' file.

term_type [str] This string specifies the type of term.

Returns

list of str

psctb.analyse._thermokin_file_tools.**read_reqn_file**(*path_to_file*)

Reads the contents of a file and returns it as a list of lines.

Parameters

path_to_file [str] Path to file that is to read in

Returns

list of str The file contents as separate strings in a list

psctb.analyse._thermokin_file_tools.**replace_pow**(*str_formulas*)

Creates new dict from an existing dict with "pow(x,y)" in values replaced with "x**y".

Goes through the values of an dictionary and uses regex to convert the pysces internal syntax for powers with standard python syntax. This is needed before conversion to sympy expressions. This use case requires reaction names as they appear in pysces as keys.

Parameters

str_formulas [dict of str:str] A dictionary where the values as contain pysces format strings representing rate equation expressions with powers in the syntax "pow(x,y)"

Returns

dict of str:str A new dictionary with str rate equations where powers are represented by standard python syntax e.g. x**y

psctb.analyse._thermokin_file_tools.**sort_terms**(*terms*)

Returns a list of two sympy expressions where the expression is positive and the second expression is negative.

Parameters

terms [list of sympy expressions] A list with length of 2 where one element is positive and the other is negative (starts with a minus symbol)

Returns

tuple of sympy expressions A tuple where the first element is positive and the second is negative.

psctb.analyse._thermokin_file_tools.**st_pt_keq_from_expression**(*expression*, *substrates*, *products*, *failure_threshold=10*)

Take an expression representing "substrates/products * Keq_expression" and returns substrates, products and keq_expression separately.

In this strategy there is no inspection of the stoichiometry as provided by the model map. Here the expressions is divided/multiplied by each substrate/product until it no longer appears in the expression. If the substrates or products are not removed after a defined number of attempts a total failure occurs and the function returns *None*

This is a fallback for cases where defined stoichiometry does not correspond to the actual rate equation.

Here cases where the substrate/product do not appear in the rate equation at all throws an assertion error.

Parameters

expression [sympy expression] The expression containing "substrates/products * Keq_expression"

substrates [list of sympy symbols] List with symbolic representations for each substrate involved in the reaction which *expression* represents.

products [list of sympy symbols] List with symbolic representations for each product involved in the reaction which *expression* represents.

failure_threshold [int, optional (Default: 10)] A threshold value the defines the number of times the metabolite removal strategy should be tried before failure.

Returns

tuple of sympy_expressions or *None* This tuple contains sympy expressions for the substrates, products and keq_expression in that order. None is returned if this strategy fails.

psctb.analyse._thermokin_file_tools.**term_to_file**(*file_name*, *expression*, *parent_name=None*, *term_name=None*)

psctb.analyse._thermokin_file_tools.**write_reqn_file**(*file_name*, *model_name*, *ma_terms*, *vc_binding_terms*, *gamma_keq_terms*, *messages*)

Module contents

psctb.latextools package

Submodules

psctb.latextools._expressions module

class psctb.latextools._expressions.**LatexExpr**(*mod*)

Bases: object

docstring for LatexExpr

Attributes

prc_subs

subs_dict

tk_subs

Methods

add_term_types	
expression_to_latex	

add_term_types(*term_types*)

expression_to_latex(*expression*, *mul_symbol=None*)

prc_subs

subs_dict

tk_subs

Module contents

psctb.modeltools package

Submodules

psctb.modeltools._paths module

psctb.modeltools._paths.**get_model_name**(*mod*)

Returns the file name of a pysces model object sans the file extension.

Parameters

mod [PysMod] Model of interest.

Returns

str File name of a *mod* sans extension.

psctb.modeltools._paths.**make_path**(*mod*, *analysis_method*, *subdirs=[]*)

Creates paths based on model name and analysis type.

This function is used to create directories (in the case where they don't already exist) to write analysis results to and return the path name. Subdirectories can also be created.

/path/to/Pysces/model_name/analysis_method/subdir1/subdir2/

Parameters

mod [PysMod] The model being analysed.

analysis_method [str] The name of the tool being used to analyse the model.

subdirs [list of str] An optional list of subdirectories where each additional entry in the list will create a subdirectory in the previous directories.

Returns

str The directory string

Examples

```
>>> print make_path(mod, 'analysis_method', subdirs = ['subdir1', subdir2])
'/path/to/Pysces/model_name/analysis_method/subdir1/subdir2/'
```

psctb.modeltools._paths.**next_suffix**(*directory*, *base_name*, *ext=None*)

Returns the number of the next suffix to be appended to a base file name when saving a file.

This function checks a `directory` for files containing `base_name` and returns a number that is equal to the suffix of a file named `base_name` with the largest suffix plus one.

Parameters

directory [str] The directory to inspect for files.

base_name [str] The base name (sans suffix) to check for.

Returns

int The next suffix to write

psctb.modeltools._paths.**get_file_path**(*working_dir*, *internal_filename*, *fmt*, *fixed=None*, *file_name=None*, *write_suffix=True*)

An heuristic for determining the correct file name.

This function determines the file name according to the information supplied by the user and the internals of a specific class.

Parameters

working_dir [str] The working dir of the specific class (where files are saved if no file name is supplied)

internal_filename [str] The default base name (sans numbered suffix) of files when no other details are provided.

fmt [str] The format (extension) that the file should be saved in. This is used both in determining file name if no file name is provided as well as when a file name without extension is provided.

fixed [str, Optional (Default]

In the case that a metabolite is fixed, files will be saved in a subdirectory of the working directory that corresponds to the fixed metabolite.

file_name [str, Optional (Default][None)] If a file name is supplied it overwrites all other options except `fmt` in the case where no extension is supplied.

Returns

——-

str The final file name

psctb.modeltools._paths.**get_fmt**(*file_name*)

Gets the extension (fmt) from a file name.

Parameters

file_name [str] The file to get an extension from

Returns

str The extension string

psctb.modeltools._pscmanipulate module

psctb.modeltools._pscmanipulate.**psc_to_str**(*name*)
Takes a filename and returns a path of where this file should be found.

Parameters

name [str] A string containing a filename.

Returns

str A string indicating the path to a psc file.

psctb.modeltools._pscmanipulate.**mod_to_str**(*mod*)
Converts an instantiated PySCeS model to a string.

Parameters

mod [PysMod] A Pysces model.

Returns

str A string representation of the contents of a PySCeS model file.

psctb.modeltools._pscmanipulate.**strip_fixed**(*fstr*)
Take a psc file string and return two strings: (1) The file header containing the "FIX: " line and (2) the remainder of file.

Parameters

fstr [str] String representation of psc file.

Returns

tuple of str 1st element contains file header, second the remainder of the file.

See also:

psc_to_str

mod_to_str

psctb.modeltools._pscmanipulate.**augment_fix_sting**(*OrigFix*, *fix*)
Adds a species to a psc file header.

Parameters

OrigFix [str] A psc file header

fix [str] Additional species to add to psc file header.

Returns

str A new psc file header that contains the contents of the original together with the new fixed species.

psctb.modeltools._pscmanipulate.**fix_metabolite**(*mod*, *fix*, *model_name=None*)
Fix a metabolite in a model and return a new model with the fixed metabolite.

Parameters

mod [PysMod] The original model.

fix [str] The metabolite to fix.

model_name [str, optional (Default] The file name to use when saving the model (in psc/orca). If None it defaults to original_model_name_fix.

Returns

PysMod A new model instance with an additional fixed species.

psctb.modeltools._pscmanipulate.**fix_metabolite_ss**(*mod*, *fix*, *model_name=None*)

Fix a metabolite at its steady state in a model and return a new model with the fixed metabolite.

Parameters

mod [PysMod] The original model.

fix [str] The metabolite to fix.

model_name [str, optional (Default] The file name to use when saving the model (in psc/orca). If None it defaults to original_model_name_fix.

Returns

PysMod A new model instance with an additional fixed species.

See also:

fix_metabolite

Module contents

psctb.utils package

Subpackages

psctb.utils.misc package

Submodules

psctb.utils.misc._misc module

psctb.utils.misc._misc.**cc_list**(*mod*)

Retuns a list of control coefficients of a model.

The list contains both flux and species control coefficients and control coefficients follow the syntax of 'cc_controlled_controller'.

Parameters

mod [PysMod] The Pysces model contains the reactions and species which is used to construct the control coefficient list.

Returns

list of str The cc_list is sorted alphabetically.

See also:

ec_list, ***rc_list***, ***prc_list***

psctb.utils.misc._misc.**ec_list**(*mod*)

Retuns a list of elasticity coefficients of a model.

The list contains both species and parameter elasticity coefficients and elasticity coefficients follow the syntax of 'ec_reaction_sp-or-param'.

Parameters

mod [PysMod] The Pysces model contains the reactions, species and parameters which is used to construct the elasticity coefficient list.

Returns

list of str The ec_list is sorted alphabetically.

See also:

cc_list, ***rc_list***, ***prc_list***

psctb.utils.misc._misc.**prod_ec_list**(*mod*)

Returns a list of product elasticity coefficients of a model.

Returns

list of str The prod_ec_list is sorted alphabetically.

See also:

ec_list, ***cc_list***, ***rc_list***, ***prc_list***

psctb.utils.misc._misc.**mod_ec_list**(*mod*)

Returns a list of modifier elasticity coefficients of a model.

Returns

list of str The mod_ec_list is sorted alphabetically.

See also:

ec_list, ***cc_list***, ***rc_list***, ***prc_list***

psctb.utils.misc._misc.**rc_list**(*mod*)

Retuns a list of response coefficients of a model.

The list contains both species and flux response coefficients and response coefficients follow the syntax of 'rc_responder_parameter'.

Parameters

mod [PysMod] The Pysces model contains the reactions, species and parameters which is used to construct the response coefficient list.

Returns

list of str The rc_list is sorted alphabetically.

See also:

cc_list, ***ec_list***, ***prc_list***

psctb.utils.misc._misc.**prc_list**(*mod*)

Retuns a list of partial response coefficients of a model.

The list contains both species and flux partial response coefficients and partial response coefficients follow the syntax of 'prc_responder_parameter_route'.

Parameters

mod [PysMod] The Pysces model contains the reactions, species and parameters which is used to construct the partial response coefficient list.

Returns

list of str The prc_list is sorted alphabetically.

See also:

cc_list, *ec_list*, *rc_list*

psctb.utils.misc._misc.**silence_print**(*func*)

A function wrapper that silences the stdout output of a function.

This function is *very* useful for silencing pysces functions that print a lot of unneeded output.

Parameters

func [function] A function that talks too much.

Returns

function A very quiet function

class psctb.utils.misc._misc.**DotDict**(**args*, ***kwargs*)

Bases: dict

A class that inherits from dict.

The DotDict class has the same functionality as dict but with the added feature that dictionary elements may be accessed via dot notation.

See also:

dict

PseudoDotDict

Methods

clear()	
copy()	
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

Continued on next page

Table 2 – continued from previous page

values()

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

class psctb.utils.misc._misc.**PseudoDotDict**(**args*, ***kwargs*)
Bases: object

A class that acts like a dictionary with dot accessible elements.

This class is not subsclassed from dict like DotDict, but rather wraps dictionary functionality.

This object has trouble being pickled :'(

See also:

dict

DotDict

Methods

update	

update(*dic*)

psctb.utils.misc._misc.**is_number**(*suspected_number*)
Test if an object is a number

Parameters

suspected_number: object This can be any object which might be a number.

Returns

boolean True if object is a number, else false

psctb.utils.misc._misc.**formatter_factory**(*min_val=None*, *max_val=None*, *default_fmt=None*, *outlier_fmt=None*)
Returns a custom *html_table* object cell content formatter function.

Parameters

min_val [int or float, optional (Default] The minimum value for float display cutoff.

max_val [int of float, optional (Default] The maximum value for float display cutoff.

default_fmt [str, options (Default] The default format for any number within the range of min_val to max_val.

outlier_fmt [str, optional (Default] The format for any number not in the range of min_val to max_val

Returns

formatter [function] A function which formats input for *html_table* using the values set up by this function.

Examples

```
>>> f = formatter_factory(min_val=1,
                          max_val=10,
                          default_fmt='%.2f',
                          outlier_fmt='%.2e')
>>> f(1)
'1.00'
>>> f(5.235)
'5.24'
>>> f(10)
'10.00'
>>> f(0.99842)
'9.98e-01'
>>> f('abc')
'abc'
```

psctb.utils.misc._misc.**html_table**(*matrix_or_array_like*, *float_fmt=None*, *raw=False*, *first_row_headers=False*, *caption=None*, *style=None*, *formatter=None*)

Constructs an html compatible table from 2D list, numpy array or sympy matrix.

Parameters

matrix_or_array_like [list of lists or array or matrix] A compatible object to be converted to an html table

float_fmt [str, optional (Default] The formatter string for numbers. This formatter will be applied to all numbers. This optional argument is only used when the argument *formatter* is None. Useful for simple tables where different types of formatting is not needed.

raw [boolean, optional (Default] If True a raw html string will be returned, otherwise an IPython *HTML* object will be returned.

first_row_headers [boolean, optional (Default] If True elements in the fist row in *matrix_or_array_like* will be considered as part of a header and will get the <th></th> tag, otherwise there will be no header.

caption [str, optional (Default] An optional caption for the table.

style [str, optional (Default] An optional html table style

formatter: function, optional (Default [None)] An optional *formatter* function. If none float_fmt will be used to format numbers.

Returns

str A string containing an html table.

OR

HTML An IPython notebook *HTML* object.

See also:

formatter_factory

psctb.utils.misc._misc.**do_safe_state**(*mod*, *parameter*, *value*, *type='ss'*)

psctb.utils.misc._misc.**find_min**(*array_like*)

psctb.utils.misc._misc.**find_max**(*array_like*)

psctb.utils.misc._misc.**split_coefficient**(*coefficient_name*, *mod*)

psctb.utils.misc._misc.**ec_dict**(*mod*)

psctb.utils.misc._misc.**cc_dict**(*mod*)

psctb.utils.misc._misc.**rc_dict**(*mod*)

psctb.utils.misc._misc.**prc_dict**(*mod*)

psctb.utils.misc._misc.**group_sort**(*old_list*, *num_of_groups*)

psctb.utils.misc._misc.**extract_model**(*obj*)

psctb.utils.misc._misc.**get_value**(*expression*, *subs_dict*)

psctb.utils.misc._misc.**get_value_eval**(*expression*, *subs_dict*)

psctb.utils.misc._misc.**get_value_sympy**(*expression*, *subs_dict*)

psctb.utils.misc._misc.**print_f**(*message*, *status*)

Prints a message if status is *True* Parameters ———- message : object

Any object with a *__str__* method.

status : bool

psctb.utils.misc._misc.**stringify**(*symbol_or_list*)

Returns a list of strings from a list of sympy.Symbol objects or a string from a sympy.Symbol.

Parameters

symbol_or_list [sympy.Symbol or list of sympy.Symbol.]

Returns

str or list of str A str or list of str representation of the sympy.Symbol or list of sympy.Symbol.

Examples

```
>>> import sympy
>>> symbol_list = sympy.sympify(['a','c','d'])
>>> a = stringify(symbol_list[0])
>>> a
'a'
>>> type(a)
<type 'str>'
>>> str_list = stringify(symbol_list)
>>> str_list
['a', 'b', 'c']
>>> type(str_list[0])
<type 'str>'
```

psctb.utils.misc._misc.**is_iterable**(*obj*)

Returns True if an object is iterable and False if it is not.

This function makes the assumtion that any iterable object can be cast as an iterator using the build-in function *iter*. This might not be the case, but works within the context of PySCeSToolbox.

Parameters

obj [object] Any object that might or might not be iterable.

Returns

bool A boolean indicating if *ob* is iterable.

psctb.utils.misc._misc.**scanner_range_setup**(*scan_range*)

From a range of numbers, returns its start point, end point, number of points and if it is a log range.

The assumption is made that only log or linear ranges are valid inputs, thus lists of random numbers would likely be classified as logarithmic.

Parameters

scan_range [iterable] Any iterable object containing a range of numbers. Most probably numpy.ndarray.

Returns

start [number] A number indicating the start point of the scan range.

end: number A number indicating the end point of the scan range.

scan_points: number A number indicating the number of scan point in the scan range

is_log_range: bool A boolean indicating whether the scan range has a logarithmic scale or not.

psctb.utils.misc._misc.**is_linear**(*scan_range*)

For any 1-demensional data structure containing numbers return True if the numbers follows a linear sequence.

Within the context of PySCeSToolbox this function will be called on either a linear range or a log range. Thus, while not indicative of log ranges, this is what a False return value indicates in this software.

Parameters

scan_range [iterable] Any iterable object containing a range of numbers.

Returns

bool A boolean indicating if *scan_range* is a linear sequence of numbers.

psctb.utils.misc._misc.**column_multiply**(*arr*)

For any 2d array returns a column vector with the product of the columns of each row.

Parameters

arr [numpy.ndarray]

Returns

numpy.ndarray A ndarray (column vector) with the products of columns of each row of arr as values

Examples

```
>>> arr = np.arrange(10).reshape(5,2)
>>> arr
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> column_multiply(arr)
array([[  0.],
       [  6.],
       [ 20.],
       [ 42.],
       [ 72.]])
```

psctb.utils.misc._misc.**unix_to_windows_path**(*path_to_convert*, *drive_letter='C'*)

For a string representing a POSIX compatible path (usually starting with either '~' or '/'), returns a string representing an equivalent Windows compatible path together with a drive letter.

Parameters

path_to_convert [string] A string representing a POSIX path

drive_letter [string (Default] A single character string representing the desired drive letter

Returns

string A string representing a Windows compatible path.

psctb.utils.misc._misc.**flux_list**(*mod*)

psctb.utils.misc._misc.**ss_species_list**(*mod*)

psctb.utils.misc._misc.**get_filename_from_caller**()

psctb.utils.misc._misc.**memoize**(*function*)

psctb.utils.misc._misc.**is_reaction**(*attr*, *model*)

psctb.utils.misc._misc.**is_species**(*attr*, *model*)

psctb.utils.misc._misc.**is_parameter**(*attr*, *model*)

psctb.utils.misc._misc.**is_variable**(*attr*, *model*)

psctb.utils.misc._misc.**is_attr**(*attr*, *model*)

psctb.utils.misc._misc.**is_mca_coef**(*attr*, *model*)

psctb.utils.misc._misc.**is_ec**(*attr*, *model*)

psctb.utils.misc._misc.**is_cc**(*attr*, *model*)

psctb.utils.misc._misc.**is_rc**(*attr*, *model*)

psctb.utils.misc._misc.**is_prc**(*attr*, *model*)

Module contents

psctb.utils.model_graph package

Submodules

psctb.utils.model_graph._model_graph module

class psctb.utils.model_graph._model_graph.**ModelGraph**(*mod*, *pos_dic=None*, *analysis_method=None*, *base_name=None*)

Bases: object

Attributes

height

nodes_fixed

straight_links

width

Methods

save([file_name])	Saves the image.

change_link_properties	
change_node_properties	
draw_all_links	
highlight_cc	
highlight_cp	
remove_dummy_sinks	
remove_external_modifier_links	
reset_node_properties	
show	

DUMMY_SINK_NAMES = ['dummy', 'sink']

REACTION_NODE = {'color': 'black', 'dx': -12, 'dy': -15, 'fill': '#00A388', 'fixed

RGB_RGB_RGB_ = {0: 'rgb(94,79,162)', 1: 'rgb(50,136,189)', 2: 'rgb(102,194,165)', 3

SPECIES_NODE = {'color': 'black', 'dx': -12, 'dy': -15, 'fill': '#FF6138', 'fixed'

change_link_properties(*elas*, *prop_dic=None*, *only_overwrite=False*)

change_node_properties(*node_name*, *prop_dic=None*)

draw_all_links()

height

highlight_cc(*cc*, *show_dummy_sinks=False*, *show_external_modifier_links=False*)

highlight_cp(*cp*, *show_dummy_sinks=False*, *show_external_modifier_links=False*)

nodes_fixed

remove_dummy_sinks()

remove_external_modifier_links()

reset_node_properties()

save(*file_name=None*)

Saves the image.

Saves the image to either the default working directory or to an a specified location. Parameters ———- file_name : str, Optional (default : None)

An optional path to which the image will be saved.

Returns

None

show(*no_links=False*, *clear_top_box=True*)

straight_links

width

Module contents

psctb.utils.plotting package

Submodules

psctb.utils.plotting._plotting module

class psctb.utils.plotting._plotting.**LineData**(*name*, *x_data*, *y_data*, *categories=None*, *properties=None*)

Bases: object

An object that contains data and metadata used by ScanFig to draw a matplotlib line with interactivity.

This object is used to initialise a ScanFig object together with a Data2D object. Once a ScanFig instance is initialised, the LineData objects are saved in a list _raw_line_data. Changing any values there will have no effect on the output of the ScanFig instance. Actual x,y data, matplotlib line metadata, and ScanFig category metadata is stored.

Parameters

name [str] The name of the line. Will be used as a label if none is specified.

x_data [array_like] The x data.

y_data [array_like] The y data.

categories [list, optional] A list of categories that a line falls into. This will be used by ScanFig to draw buttons that enable/disable the line.

properties [dict, optional] A dictionary of properties of the line to be drawn. This dictionary will be used by the generic set() function of matplotlib.Lines.Line2D to set the properties of the line.

See also:

ScanFig

Data2D

RateChar

Methods

add_property(key, value)	Adds a property to the properties dictionary of the LineData object.

add_property(*key*, *value*)

Adds a property to the properties dictionary of the LineData object.

The properties dictionary of LineData will be used by the generic set() function of matplotlib.Lines.Line2D to set the properties of the line.

Parameters

key [str] The name of the matplotlib.Lines.Line2D property to be set.

value [sting, int, bool] The value of the property to be set. The type depends on the property.

class psctb.utils.plotting._plotting.**ScanFig**(*line_data_list*, *category_classes=None*, *fig_properties=None*, *ax_properties=None*, *base_name=None*, *working_dir=None*)

Bases: object

Uses data in the form of a list of LineData objects to display interactive plots.

Interactive plots can be customised in terms of which data is visible at any one time by simply clicking a button to toggle a line. Matplotlib figures are used internally, therefore ScanFig figures can be altered by changing the properties of the internal figure.

Parameters

line_data_list [list of LineData objects] A LineData object contains the information needed to draw a single curve on a matplotlib figure. Here a list of these objects are used to populate the internal matplotlib figure with the various curves that represent the results of a parameter scan or simulation.

category_classes [dict, Optional (Default] Each line on a ScanFig plot falls into a different category. Each of these categories in turn fall into a different class. Each category represents a button which toggles the lines which fall into the category while the button is arranged under a label which is represented by a category class. Each key in this dict is a category class and the value is a list of categories that fall into this class. If None all categories will fall into the same class.

fig_properties [dict, Optional (Default] A dictionary of properties that will be used to adjust the appearance of the figure. These properties should compatible with matplotlib.figure.Figure'' object in a way that its ``set method can be used to change its properties. If None, default matplotlib figure properties will be used.

ax_properties [dict, Optional (Default] A dictionary of properties that will be used to adjust the appearance of plot axes. These properties should compatible with matplotlib.axes.AxesSubplot'' object in a way that its ``set method can be used to change its properties. If None default matplotlib axes properties will be used.

base_name [str, Optional (Default] Base name that will be used when an image is saved by ScanFig. If None, then scan_fig will be used.

working_dir [str, Optional (Default] The directory in which files figures will be saved. If None, then it will default to the directory specified in pysces.output_dir.

See also:

LineData

Data2D

Attributes

categories_status

category_names

line_names

Methods

adjust_figure()	Provides widgets to set the limits and scale (log/linear) of the figure.
interact()	Displays the figure in a IPython/Jupyter notebook together with buttons to toggle the visibility of certain lines.
save([file_name, dpi, fmt, include_legend])	Saves the figure in it's current configuration.
show()	Displays the figure.
toggle_category(cat, value)	Changes the visibility of all the lines in a certain line category.
toggle_line(name, value)	Changes the visibility of a certain line.

adjust_figure()
Provides widgets to set the limits and scale (log/linear) of the figure.

As with `interact`, the plot is displayed in the notebook. Here no widgets are provided the change the visibility of the data displayed on the plot, rather controls to set the limits and scale are provided.

See also:

show

interact

categories_status

category_names

interact()
Displays the figure in a IPython/Jupyter notebook together with buttons to toggle the visibility of certain lines.

See also:

show

adjust_figure

line_names

save(*file_name=None*, *dpi=None*, *fmt=None*, *include_legend=True*)
Saves the figure in it's current configuration.

Parameters

file_name [str, Optional (Default] The file name to be used. If None is provided the file will be saved to `working_dir/base_name.fmt`

dpi [int, Optional (Default] The dpi to use. Defaults to 180.

fmt [str, Optional (Default] The image format to use. Defaults to `svg`. If `file_name` contains a valid extension it will supersede `fmt`.

show()
Displays the figure.

Depending on the matplotlib backend this function will either display the figure inline if running in an `IPython` notebook with the `--pylab=inline` switch or with the %matplotlib inline IPython line magic, alternately it will display the figure as determined by the `rcParams['backend']` option of `matplotlib`. Either the inline or nbAgg backends are recommended.

See also:

interact

adjust_figure

toggle_category(*cat*, *value*)

Changes the visibility of all the lines in a certain line category.

When used all lines in the provided category's visibility is changed according to the `value` provided.

Parameters

cat: str The name of the category to change.

value: bool The visibility status to change the lines to (True for visible, False for invisible).

See also:

toggle_line

toggle_line(*name*, *value*)

Changes the visibility of a certain line.

When used a specific line's visibility is changed according to the `value` provided.

Parameters

name: str The name of the line to change.

value: bool The visibility status to change the line to (True for visible, False for invisible).

See also:

toggle_category

class psctb.utils.plotting._plotting.**Data2D**(*mod*, *column_names*, *data_array*, *ltxe=None*, *analysis_method=None*, *ax_properties=None*, *file_name=None*, *additional_cat_classes=None*, *additional_cats=None*, *num_of_groups=None*, *working_dir=None*, *category_manifest=None*, *axvline=True*)

Bases: `object`

An object that wraps results from a PySCeS parameter scan.

Results from parameter scan or timecourse are used to initialise this object which in turn is used to create a `ScanFig` object. Here results can easily be accessed and saved to disk.

The `Data2D` is also responsible for setting up a `ScanFig` object from analysis results and therefore contains optional parameters for setting up this object.

Parameters

mod [PysMod] The model for which the parameter scan was performed.

column_names [list of str] The names of each column in the data_array. Columns should be arranged with the input values (scan_in, time) in the first column and the output values (scan_out) in the columns that follow.

data_array [ndarray] An array containing results from a parameter scan or tome simulation. Arranged as described above.

ltxe [LatexExpr, optional (Default] A LatexExpr object that is used to convert PySCeS compatible expressions to LaTeX math. If None is supplied a new LatexExpr object will be instantiated. Sharing a single instance saves memory.

analysis_method [str, Optional (Default] A string that indicates the name of the analysis method used to generate the results that populate `Data2D`. This will determine where results are saved by `Data2D` as well as any `ScanFig` objects that are produced by it.

ax_properties [dict, Optional (Default] A dictionary of properties that will be used by `ScanFig` to adjust the appearance of plots. These properties should compatible with `matplotlib.axes.AxesSubplot'' object in a way that its ``set` method can be used to change its properties. If none, a default `ScanFig` object is produced by the `plot` method.

file_name [str, Optional (Default] The name that should be prepended to files produced any `ScanFig` objects produced by `Data2D`. If None, defaults to 'scan_fig'.

additional_cat_classes [dict, Optional (Default] A dictionary containing additional line class categories for `ScanFig` construction. Each `data_array` column contains results representing a specific category of result (elasticity, flux, concentration) which in turn fall into a larger class of data types (All Coefficients). This dictionary defines which line classes fall into which class category. (k = category class; v = line categories)

additional_cats [dict, Optional (Default] A dictionary that defines additional result categories as well as the lines that fall into these categories. (k = line category, v = lines in category).

num_of_groups [int, Optional (Default] A number that defines the number of groups of lines. Used to ensure that the lines that are closely related (e.g. elasticities for one reaction) have colors assigned to them that are easily differentiable.

working_dir [str, Optional (Default] This string sets the working directory directly and if provided supersedes `analysis_method`.

See also:

ScanFig

Data2D

RateChar

Methods

plot()	Creates a `ScanFig` object using the data stored in the current instance of `Data2D`
save_results([file_name, separator, fmt])	Saves data stores in current instance of `Data2D` as a comma separated file.

plot()
Creates a `ScanFig` object using the data stored in the current instance of `Data2D`

Returns

ScanFig A *ScanFig'* object that is used to visualise results.

save_results (*file_name=None*, *separator=', '*, *fmt='%f'*)
Saves data stores in current instance of `Data2D` as a comma separated file.

Parameters

file_name [str, Optional (Default] The file name, extension and path under which data should be saved. If None the name will default to scan_data.csv and will be saved either under the directory specified under the directory specified in `folder`.

separator [str, Optional (Default] The symbol which should be used to separate values in the output file.

format [str, Optional (Default] Format for the data.

`psctb.utils.plotting._plotting.`**load_data2d**(*file_name*, *mod=None*, *ltxe=None*)
Loads a gzipped cPickle file containing a Data2D object. Optionally a model can be provided (which is useful when loading data that reference the same model. For the same reason a LatexExpr object can be supplied.

`psctb.utils.plotting._plotting.`**save_data2d**(*data_2dobj*, *file_name*)
Saves a Data2D object to a gzipped cPickle to a specified file name.

class `psctb.utils.plotting._plotting.`**SimpleData2D**(*column_names*, *data_array*, *mod=None*)
Bases: `object`

Methods

plot()	Creates a `ScanFig` object using the data stored in the current instance of `Data2D`
save_results([file_name, separator, fmt])	Saves data stores in current instance of `Data2D` as a comma separated file.

plot()
Creates a `ScanFig` object using the data stored in the current instance of `Data2D`

Returns

ScanFig A *ScanFig'* object that is used to visualise results.

save_results(*file_name=None*, *separator=', '*, *fmt='%f'*)
Saves data stores in current instance of `Data2D` as a comma separated file.

Parameters

file_name [str, Optional (Default] The file name, extension and path under which data should be saved. If None the name will default to scan_data.csv and will be saved either under the directory specified under the directory specified in `folder`.

separator [str, Optional (Default] The symbol which should be used to separate values in the output file.

fmt [str, Optional (Default] Format for the data.

Module contents

Submodules

psctb.utils.config module

class `psctb.utils.config.`**ConfigChecker**
Bases: `object`

Methods

check_config	
warn_user	

static check_config(*config_name*, *config_path*, *config_dict*)

static warn_user(*exception*, *solution*)

class psctb.utils.config.**ConfigReader**
Bases: object

Methods

get_config	
reload_config	

classmethod get_config()

classmethod reload_config()

class psctb.utils.config.**ConfigWriter**
Bases: object

Methods

write_config	

static write_config(*config_dict*, *config_path*)

exception psctb.utils.config.**MissingSection**
Bases: Exception

exception psctb.utils.config.**MissingSetting**
Bases: Exception

class psctb.utils.config.**PathFinder**
Bases: object

Methods

find_match	
find_path_to	
which	

static find_match(*base_dir*, *to_match*)

static find_path_to(*wildcard_path*)

static which(*program*)

Module contents

9.1.2 Module contents

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

P

R

S

T

U

V

W