PyscesToolbox Documentation

Release 0.9.0

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. Operating system-specific instructions are discussed in the sections below. 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.

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 Windows

Windows requires the manual installation of **Python 3.x**, **PySCeS** and **Maxima**. Installation was tested on Windows 10.

2.2.1 Python

For Windows 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 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.

If you prefer more fine-grained control it is also possible to install Python from Python.org (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.

2.2.2 PySCeS

If you installed Anaconda, PySCeS can be installed by opening a command prompt in an Anaconda environment (Python 3.6 or 3.7) and executing the command:

```
conda install -c sbmlteam -c pysces pysces
```

If you installed Python from Python.org directly, open a Windows Command Prompt, verify that the Python paths are set up correctly by checking the `pip` version and install PySCeS by executing:

```
pip -V
pip install pysces
```

Currently Python versions 3.6-3.8 are supported with a pip install.

2.2.3 Maxima

Maxima is necessary for generating control coefficient expressions using SymCA. 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 `C:\Pysces\psctb_config.ini` by default.

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.

2.2.4 PySCeSToolbox

Now you are ready to install PySCeSToolbox. If you are using Anaconda, open up the Anaconda Command Prompt (Start –> Anaconda Command Prompt), else open up the Windows Command Prompt if you installed PySCeS with `pip`. In the command prompt, type in and execute the command:

```
pip install pyscestoolbox
```

This will automatically download both PySCeSToolbox and any outstanding requirements.

2.2.5 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:

```
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

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

```
jupyter nbextension install --py --user d3networkx_psctb
jupyter nbextension enable --py --user d3networkx_psctb
```

2.3 macOS (Mac OS X)

macOS requires the manual installation of **PySCeS** and **Maxima**. While OS X comes pre-installed with Python 2.7, **Python 3.x** is needed and we recommend installing a Python distribution such as Anaconda as it will take care of many of the SciPy stack requirements. Installation was tested on macOS High Sierra.

2.3.1 Python

One of the easiest ways to get Python on your system is to install the Anaconda Python distribution (https://www.anaconda.com/products/individual#Downloads). Download either of the Python 3.7 installers for macOS from the download page and follow the provided instructions.

If you prefer more fine-grained control, there are other options such as installing directly from Python.org (https://www.python.org/downloads/mac-osx/), or installing Homebrew and then installing Python 3.7 with Homebrew. **These are advanced options for experienced users, and if you are starting out, we recommend Anaconda!**

2.3.2 PySCeS

Binary packages are available for Anaconda, and binary wheels are available for direct installation with `pip`. Depending on your Python installation (see above), the process is similar to the Windows install.

For Anaconda:

```
conda install -c sbmlteam -c pysces pysces
```

For a `pip` based install (Python versions 3.6-3.8 are supported):

```
pip install pysces
```

2.3.3 Maxima

Maxima is necessary for generating control coefficient expressions using SymCA. 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` file:

```
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
```

2.3.4 PySCeSToolbox

Now you are ready to install PySCeSToolbox. If you are using Anaconda, open up a terminal in the Anaconda environment where PySCeS is installed. For `pip` based installations, just open up a Terminal. Execute the command:

```
pip install pyscestoolbox
```

This will automatically download both PySCeSToolbox and any outstanding requirements.

Note: You may encounter an error during the installation of PySCeSToolbox relating to the removal of temporary files on OS X or macOS. This does not impact on the functioning of PySCeSToolbox at all, and we mean to address this bug in the future.

2.3.5 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:

```
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

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

```
jupyter nbextension install --py --user d3networkx_psctb
jupyter nbextension enable --py --user d3networkx_psctb
```

2.4 Linux

Linux requires the manual installation **Maxima** and **PySCeS**. Most Linux systems come pre-installed with a version of **Python 3.x** or it is available from the distribution repositories. However, a Python distribution such as Anaconda may be used instead. Installation was tested on Ubuntu 18.04.

2.4.1 Python

We assume that your system comes with Python 3.x (versions 3.6-3.8 are recommended) and with `pip` (necessary for installing Python packages that are not available in your OS's repositories). In case `pip` is not yet installed, it may be installed from your OS's repositories or by following the instructions found at https://pip.pypa.io/en/stable/installing/.

If you prefer Anaconda, Linux installers are available here.

2.4.2 PySCeS

Binary packages are available for Anaconda, and binary wheels are available for direct installation with `pip`. Depending on your Python installation (see above), the process is similar to the Windows and macOS installs.

For Anaconda:

```
conda install -c sbmlteam -c pysces pysces
```

For a `pip` based install (Python versions 3.6-3.8 are supported):

```
pip install pysces
```

2.4.3 Maxima

Maxima is necessary for generating control coefficient expressions using SymCA. 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.

2.4.4 PySCeSToolbox

Now you are ready to install PySCeSToolbox. Open a terminal in the environment where you installed PySCeS (i.e. Anaconda environment or the native Python environment of your OS), and simply type in and execute the command:

```
pip install pyscestoolbox
```

2.4.5 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:

```
jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

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

```
jupyter nbextension install --py --user d3networkx_psctb
jupyter nbextension enable --py --user d3networkx_psctb
```

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

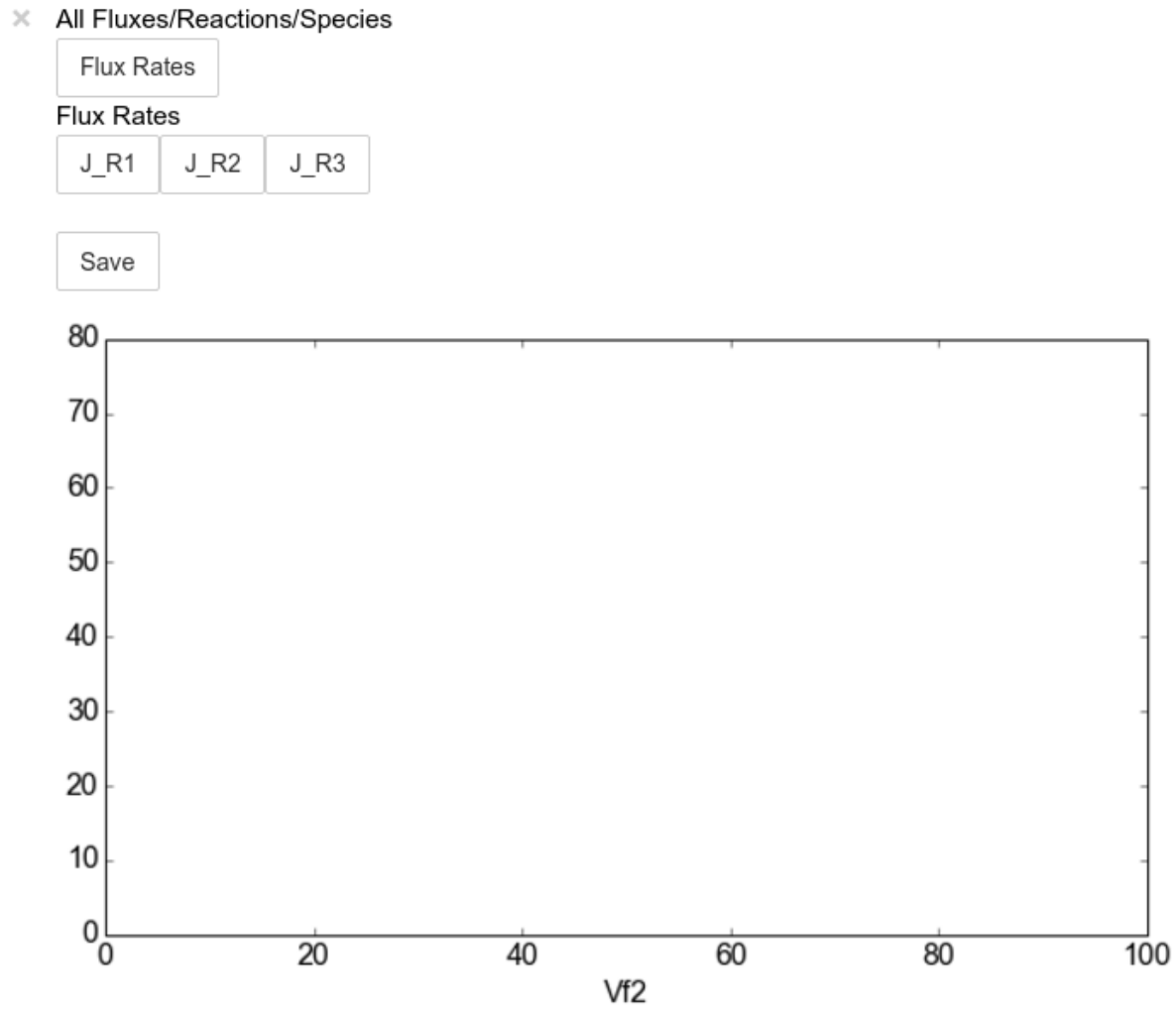
- Interactive plotting via the `interact` method.
- Script based plot generation where certain lines, or categories of lines (based on the type of information they represent), can be enabled and disabled via `toggle_line` or `toggle_category` methods.
- Saving of plots with the `save` method.
- Customisation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

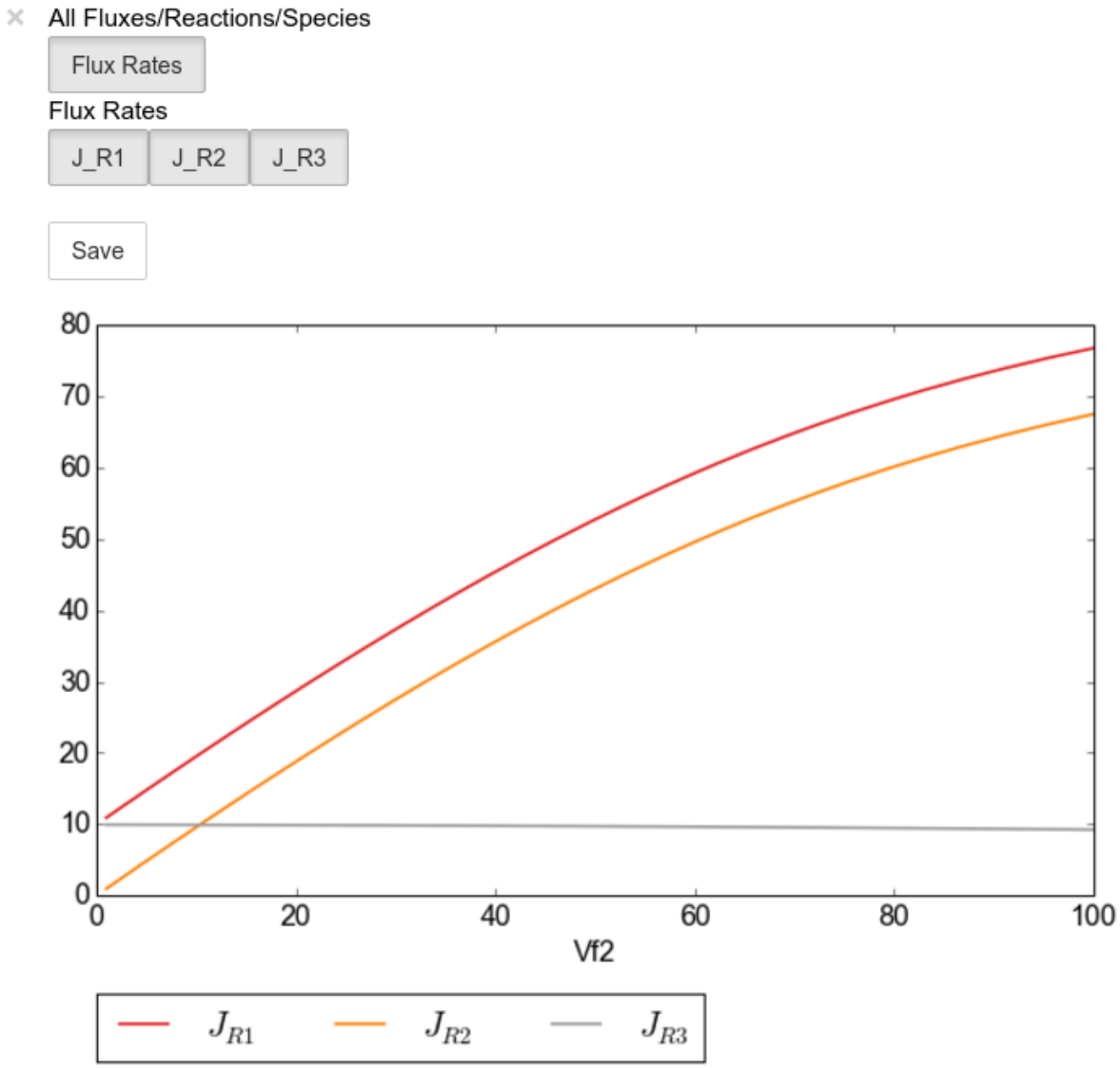
```
scan_figure.interact()
```



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)
```

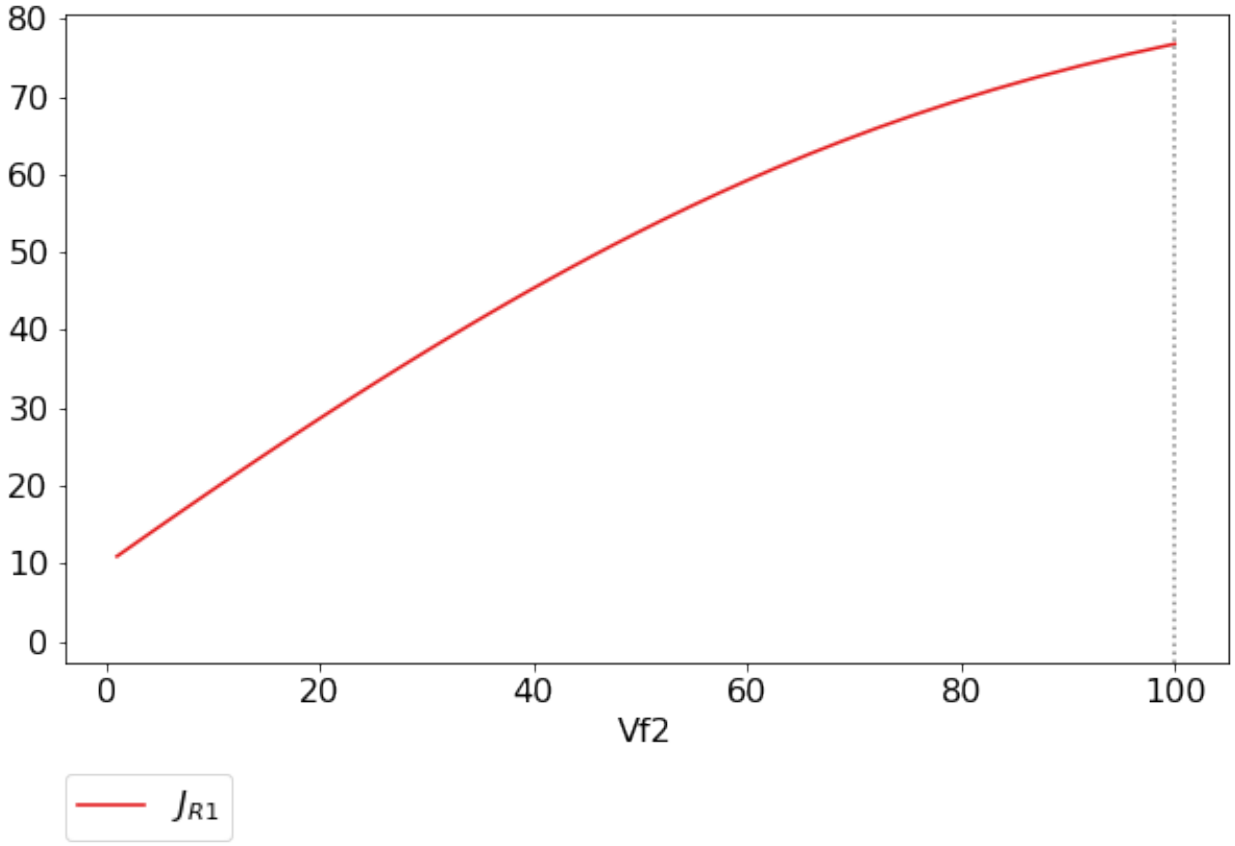
Out[9]:

```
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]:

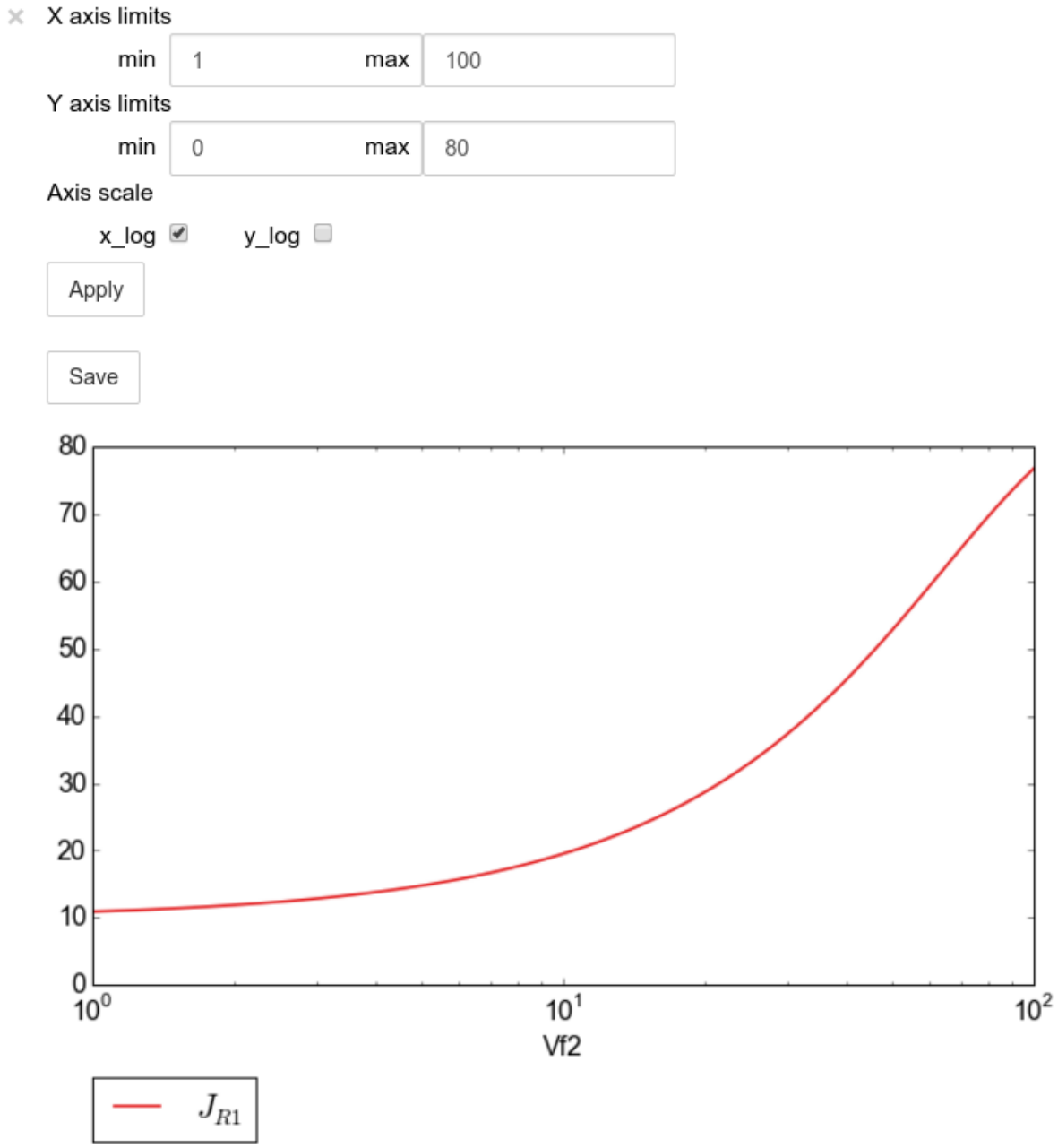
```
scan_figure.toggle_category('Flux Rates',False)
scan_figure.toggle_line('J_R1',True)
scan_figure.show()
```



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]:

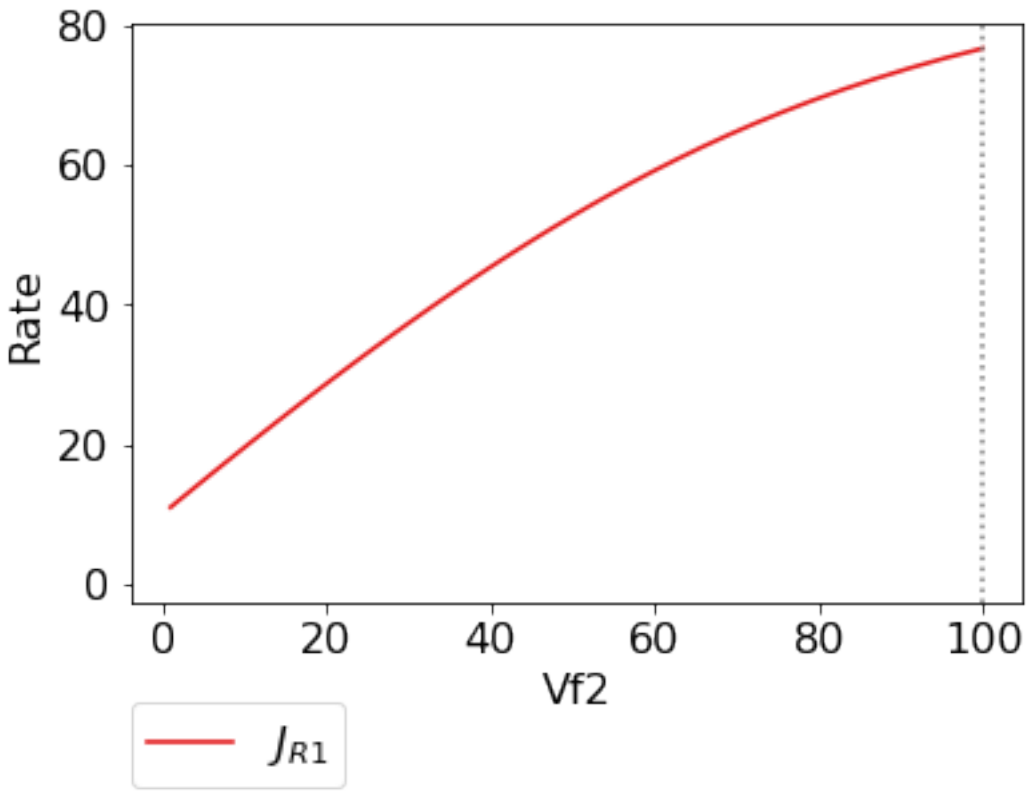
```
scan_figure.adjust_figure()
```



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.

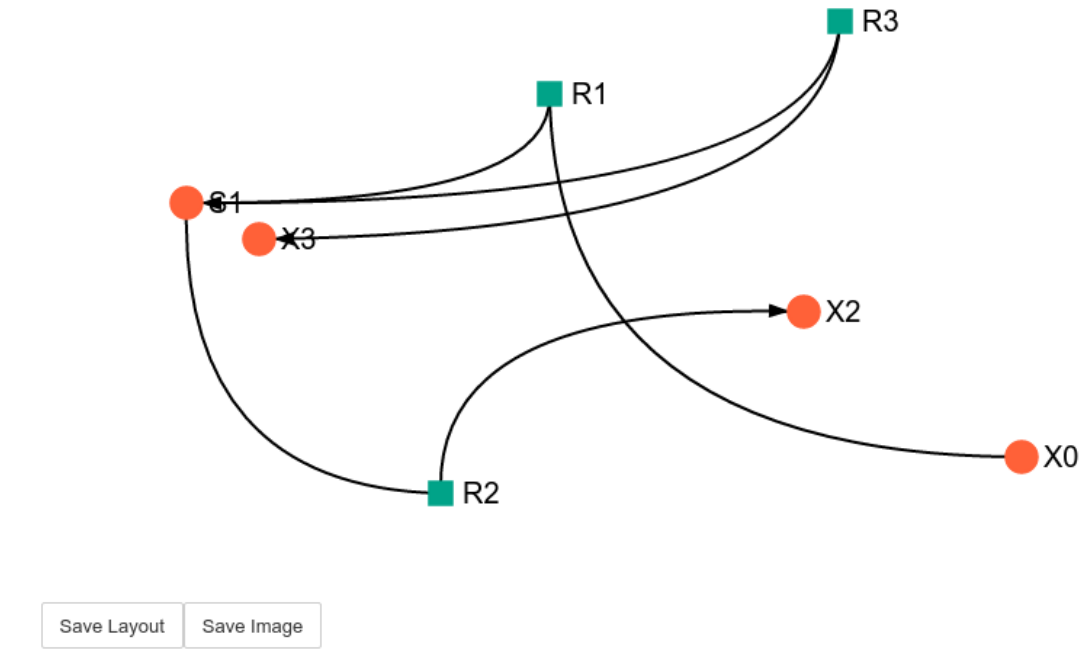
In [20]:

```
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]:

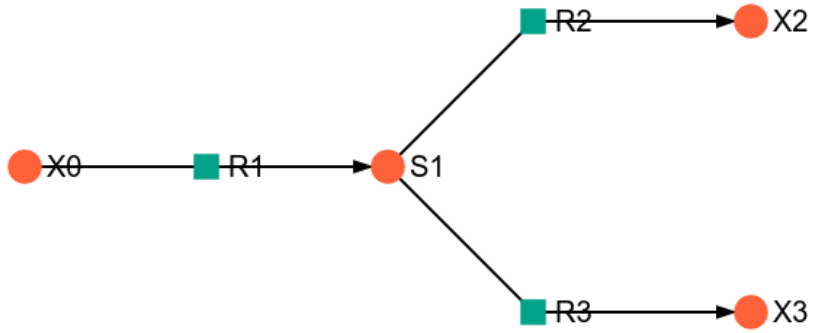
```
model_graph.show()
```



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()
```

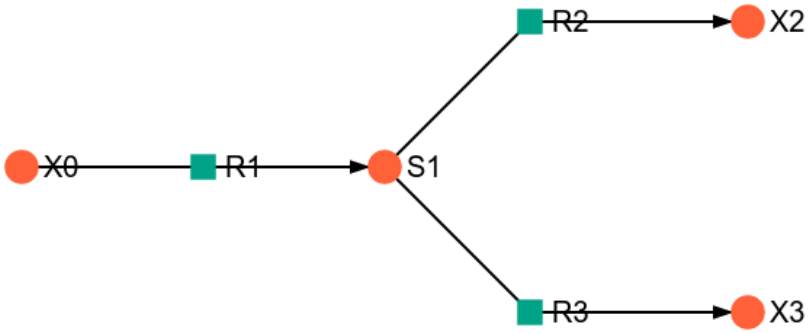


Clicking the `Save Layout` button saves this layout to the `~/Pysces/example_model/model_graph` or `C:\Pysces\example_model\model_graph` directory for later use. The `Save Image` Button wil save an svg image of the graph to the same location.

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:

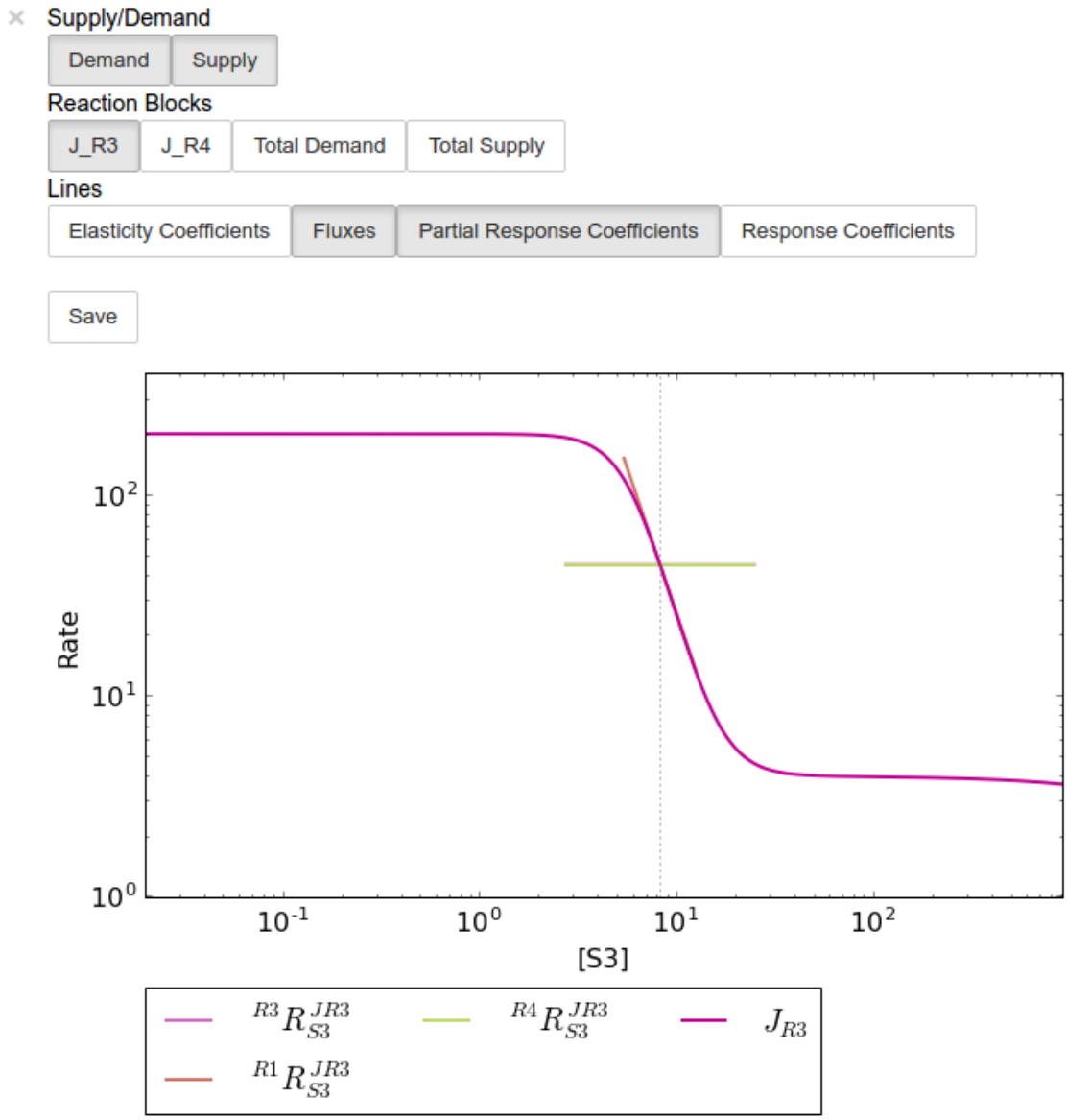
In [18]:

```
# 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()
```

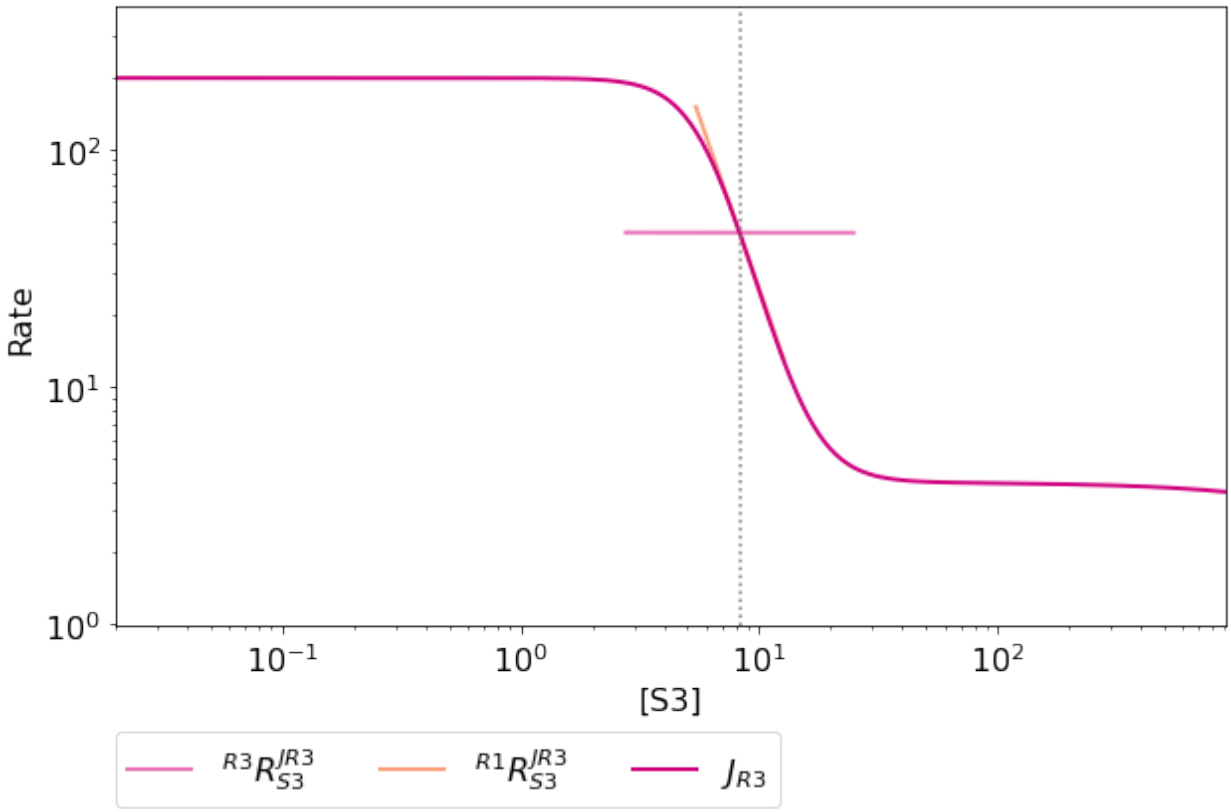


Modifying the status of individual lines is still supported, but has to take place via the `toggle_line` method. As an

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`

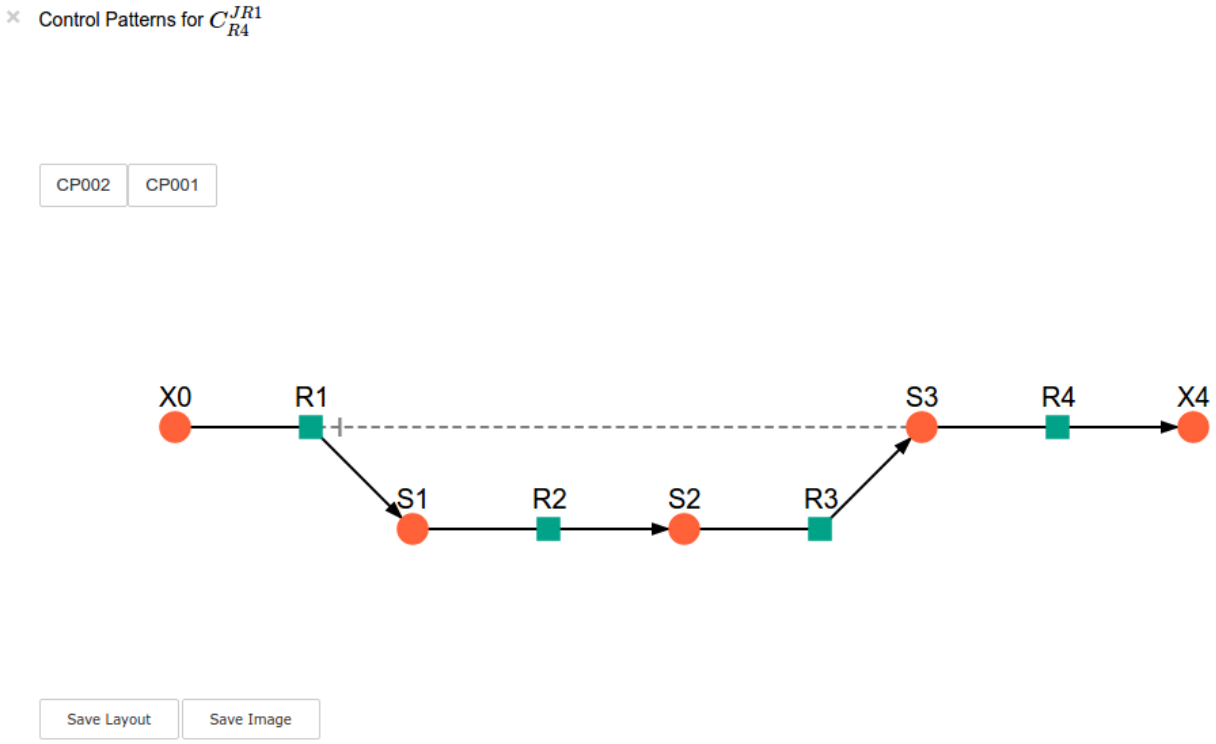
method:

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)
```



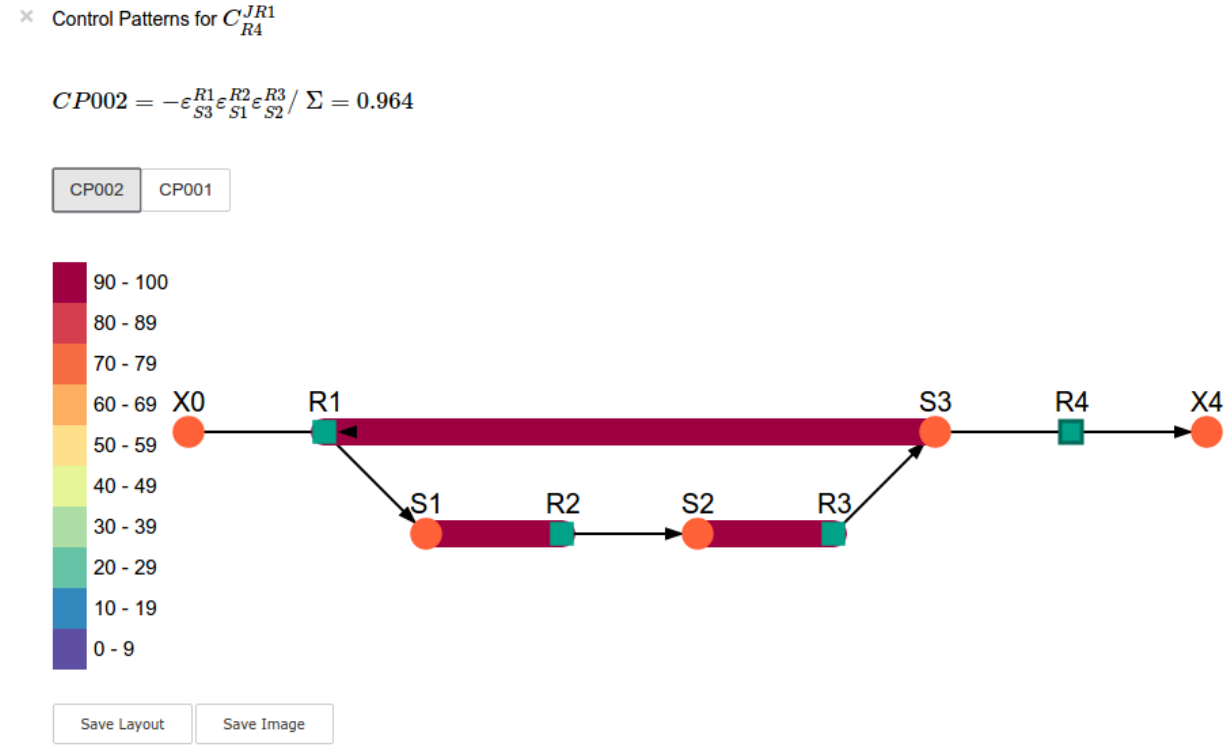
`highlight_patterns` has the following optional arguments:

- `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.

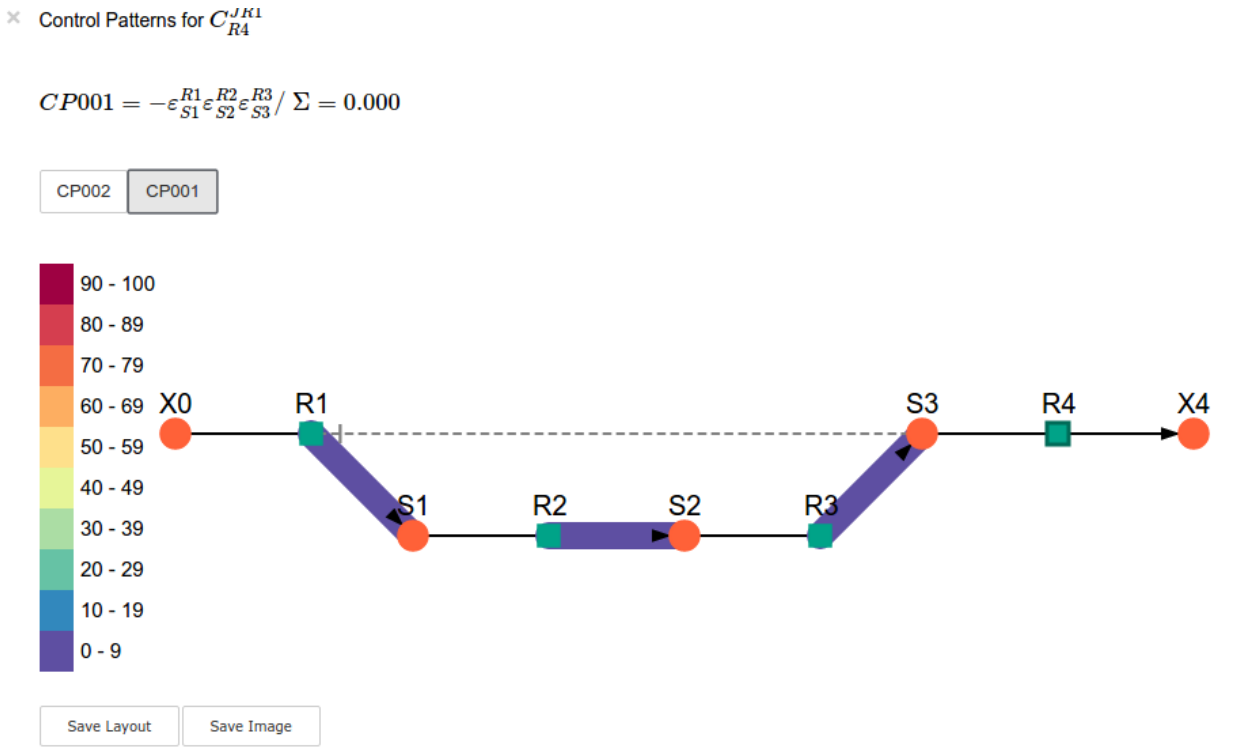
In [20]:

```
# 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')
```

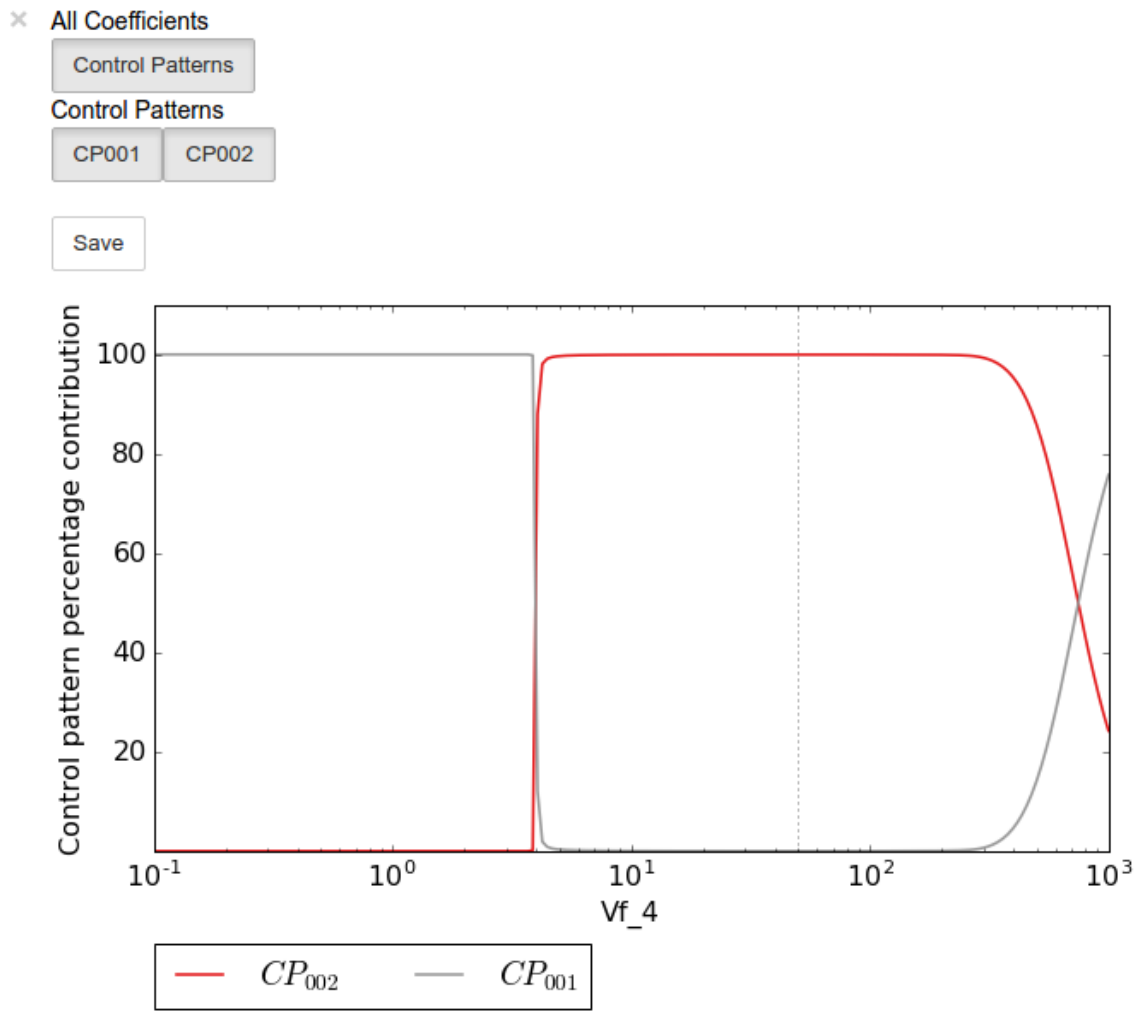
Out[22]:

```
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:

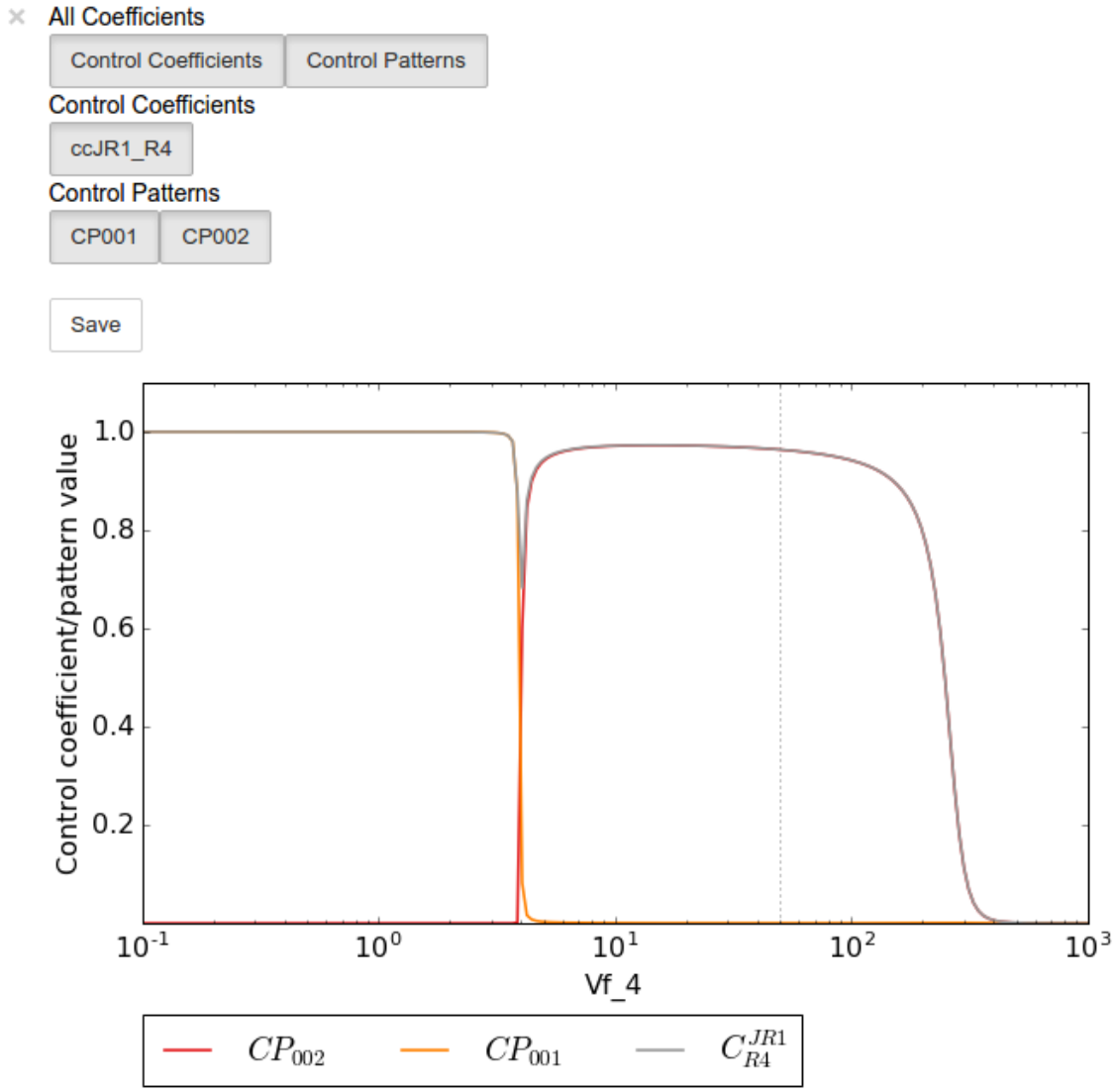
In [24]:

```
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]:

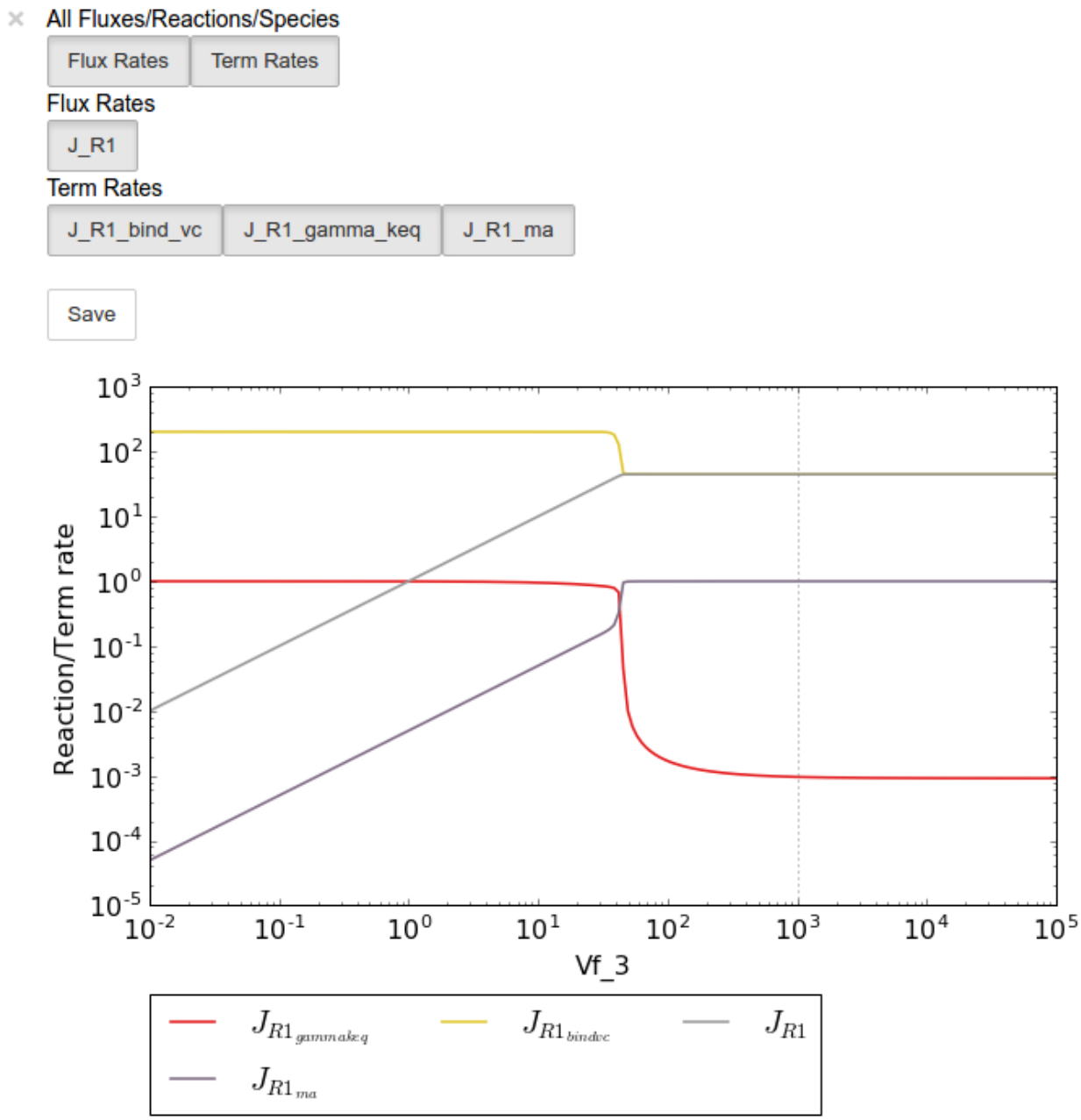
```
valscan = tk.J_R1.do_par_scan('Vf_3',scan_range=numpy.logspace(-2,5,200),scan_type=
↪'value')
```

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)

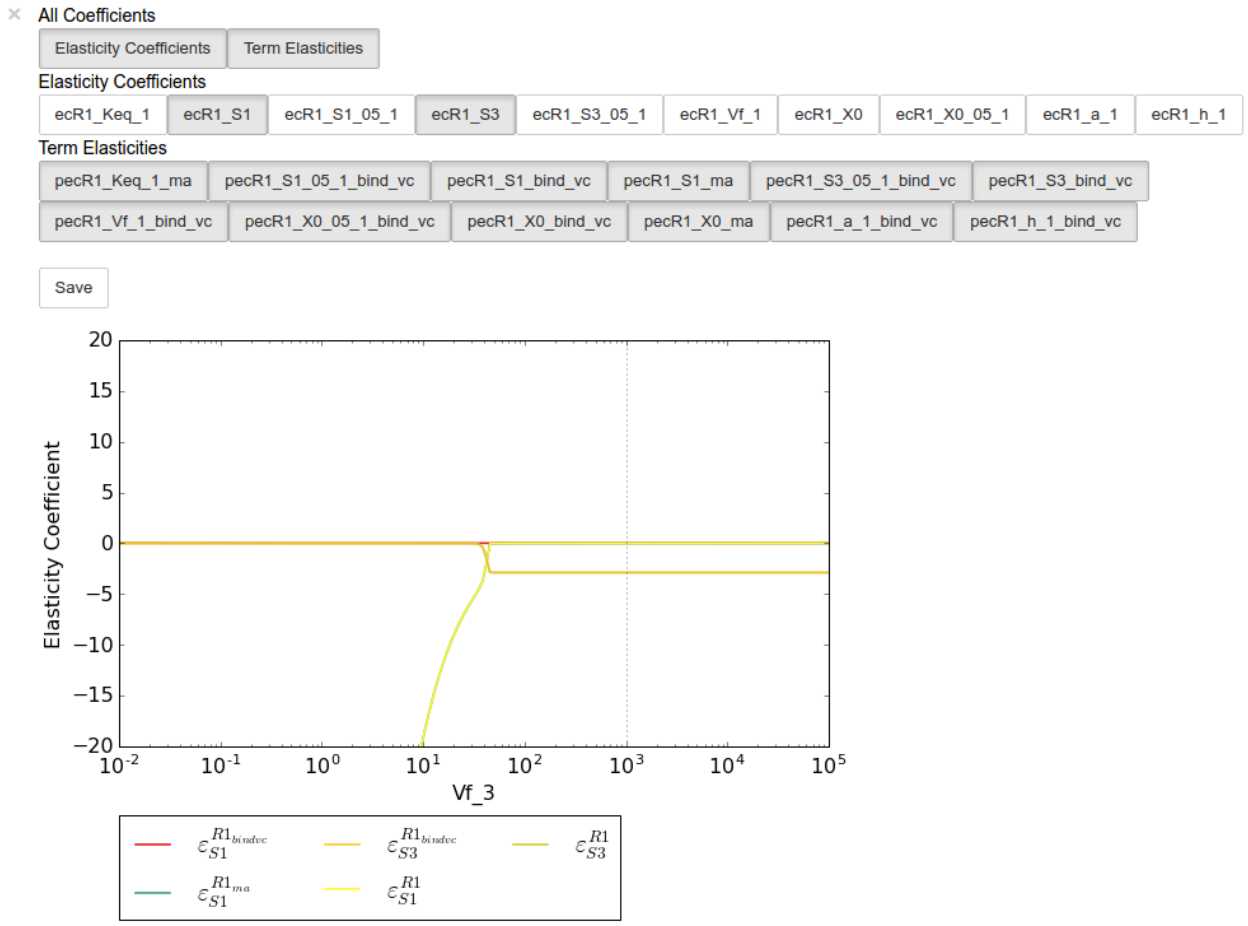
(continued from previous page)

```
(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

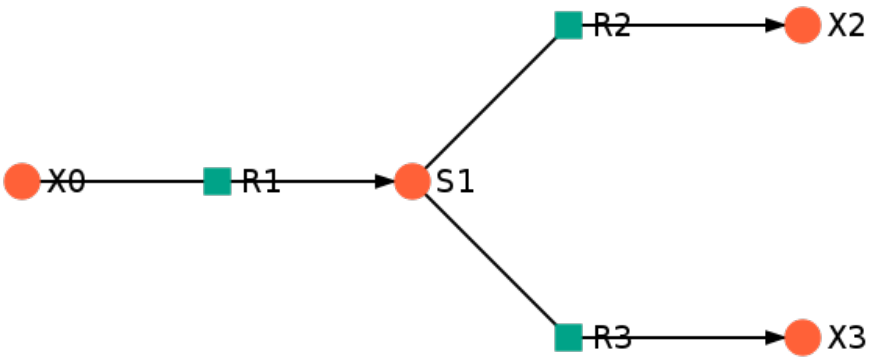
Included Files

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



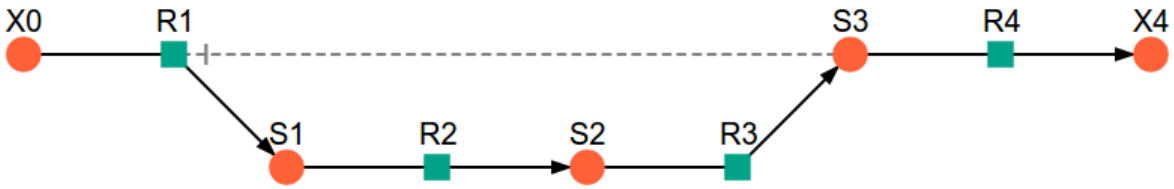
`model`

`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

psctb.analyse._symca.ccobjects module

psctb.analyse._symca.symca_toolbox module

Module contents

Submodules

psctb.analyse._ratechar module

psctb.analyse._thermokin module

psctb.analyse._thermokin_file_tools module

Module contents

psctb.latextools package

Submodules

psctb.latextools._expressions module

Module contents

psctb.modeltools package

Submodules

psctb.modeltools._paths module

psctb.modeltools._pscmanipulate module

Module contents

psctb.utils package

Subpackages

psctb.utils.misc package

Submodules

psctb.utils.misc._misc module

Module contents

psctb.utils.model_graph package

Submodules

psctb.utils.model_graph._model_graph module

Module contents

psctb.utils.plotting package

Submodules

9.1.2 Module contents

CHAPTER 10

Indices and tables

- genindex
- modindex
- search