

Sputnik: a Stochastic Petri Net Library in Python

Philipp Buerger, Erik Clark, Benjamin Moore, Oliver Palmer

March 21, 2013

Contents

1	Installation	3
2	Quick start	4
3	Introduction	5
4	Using the library: GUI	7
4.1	Basic functions	9
4.2	Simulation	11
5	Using the library: command-line	13
5.1	Help documentation	13
5.2	Import and export of Petri nets	14
5.3	Manually inputting a Petri net	17
5.4	Create a PetriNet object	18
5.5	Petri net visualisation	20
5.6	Calculating the invariants	24
5.7	Running a simulation	24
5.8	Plotting simulation diagrams	28
6	Examples	30
6.1	Example 1 : Repressilator	30
6.2	Example 2 : Four-reaction system	34
6.3	Example 3 : MAP kinase	36

7	Extending the library	38
7.1	Extending the GUI	38
7.2	Adding additional language support	41
7.3	Adding additional simulation algorithms	42
	References	43

1. Installation

Sputnik encompasses a range of tools that allow the design, simulation and analysis of stochastic Petri nets. **Sputnik** is written for use with Python version 2.7 and has been tested on both Linux and OSX operating systems. It has the following dependencies:

Essential libraries:

NumPy,[†] Scipy,[†] matplotlib,[†] gtk, pygtk.

Optional libraries:

libsbml - Required if SBML import functionality is desired

[†]We recommend using the Enthought Python Distribution which is free for academic use and includes Python 2.7.2, NumPy, Scipy and matplotlib.









Having downloaded the latest version of **Sputnik** from https://github.com/sputnikpetrinets/project_sputnik/, first extract the files using the following command:

```
> unzip project_sputnik-master.zip
```

To run the GUI, simply call `python run-sputnik.py` from the `project_sputnik-master` directory. To import the library for use in other Python scripts, move the directory to somewhere on your `PYTHONPATH`. Similarly, to make the GUI accessible system-wide, move the directory to one listed in your shell's `PATH`.

2. Quick start

To jump into the program without reading the full documentation, follow this short vignette which is explained in more detail in section 6.1.

1. Open the Sputnik GUI with `python start.py`
2. Use  to open the `repressilator.txt` file found in the `examples/` directory.
3. View the calculated net invariants with .
4. Open the simulation  options, set **Runtime** to, for example, 100000 and **Time-Step** to 500 and click **Start Simulation**.
5. After a few seconds, an info-box will inform you the simulation is finished, click **Close**.
6. Now in the lower-right **Places** pane, uncheck ‘Plot’ for all places except **pA**, **pB** and **pC** — these are the three proteins of interest. Click **Show Plot** to see the results.
7. Optionally, use the various plot settings to add a title, choose custom line colours and edit legend-text and use the plot window dialogue to save the simulation plot.
8. Repeat the simulation by increasing the **Number of Simulations** to 4. Check the **Create subplots** checkbox and enter 2 in **Subplot Rows** and **Columns** to create a 2×2 plot of all four simulations — highlighting the stochastic variation between runs.
9. Close the Simulation window and click  to view a token game animation of simulated Petri net events. Click **OK** to use the default parameters.
10. The play  button will then step through the simulation, highlighting firing transitions in **red** and updating the place markings as tokens are exchanged. You can also pause  and step through each event using  and .

For more detailed examples and a full explanation of **Sputnik**’s features and capabilities, consult sections 4, 5 and 6.

3. Introduction

Petri nets^[1] are a versatile and understandable graph notation which can be used to represent many kinds of network. They offer an unambiguous mathematical framework to describe a system, as well as providing the means to test hypotheses regarding its behaviour and properties^[2]. Having evolved in mathematics and computer science^[3], Petri nets have more recently been used by systems biologists for the purpose of modelling biological systems^[4,5,6]. Stochastic Petri nets are a form of Petri net suitable for simulating systems with stochastic mass action kinetics. They are therefore a convenient way to represent and simulate systems where low species abundances result in appreciable intrinsic noise^[7]. Such systems include gene regulatory networks, signalling systems, and many metabolic networks.

A Petri net consists of four types of element: places, transitions, arcs and tokens. Places are nodes representing the components of a system, while transitions represent reactions or events. Arcs are directed edges that connect the two. Each arc is associated with an integer weight representing the stoichiometry of the transitions. The state of the system is given by the distribution of tokens among the places; the number of tokens possessed by a place is known as its marking. The firing of a particular transition, known as an “event”, moves tokens between places as directed by the pre and post arcs, changing the “global marking” of the system. A particular transition is “enabled” only if the markings of all its input places at least equal the weights of the corresponding pre arcs. Places may have a “capacity”, a maximum number of tokens that can be possessed by the place. “Test arcs” and “inhibitory arcs”, which represent minimum and maximum marking conditions respectively, may provide additional checks on whether a particular transition is enabled, but do not directly result in the consumption of tokens. The “ P -invariants” of a net are combinations of places with a constant total marking, while the “ T -invariants” are patterns of transition firings that leave the global marking unchanged^[8]. Firings are probabilistic and associated with stochastic rate constants, with exponentially distributed waiting times between events^[2].

Here we present a Python library, **Sputnik** that provides a range of tools for constructing, visualising, analysing and simulating stochastic Petri nets, accessible either from the command-line or from a user-friendly GUI. Petri net models may be imported from file, or created *de novo* using the Petri net editor. Automatic layouting of the nets is achieved via a choice of drawing algorithms, while net components can also be manually repositioned for fine-tuning. Petri nets may be simulated by either of two stochastic simulation algorithms, the Gillespie^[9] or the tau-leap^[10], and the resulting timecourses can be visualised in customisable plots. Net properties such as the P - and T -invariants, the stoichiometry matrix and the dependency matrix can also be calculated.

In addition to defining a TXT format for storing Petri net models, **Sputnik** also supports the systems biology exchange formats Systems Biology Markup Language (SBML)^[11] and Petri Net Markup Language (PNML)^[12]. Petri nets may be saved or retrieved in any of these three file formats, ensuring cross-compatibility with the growing body of systems biology software already available. Petri net visualisations may be exported in a variety of common image formats, and the layout coordinates may additionally be saved to file, to be re-imported the next time the Petri net is loaded. Simulation visualisations may also be saved, or the raw data exported for further analysis. A basic data flow diagram describing these processes is shown in Figure 3.1.

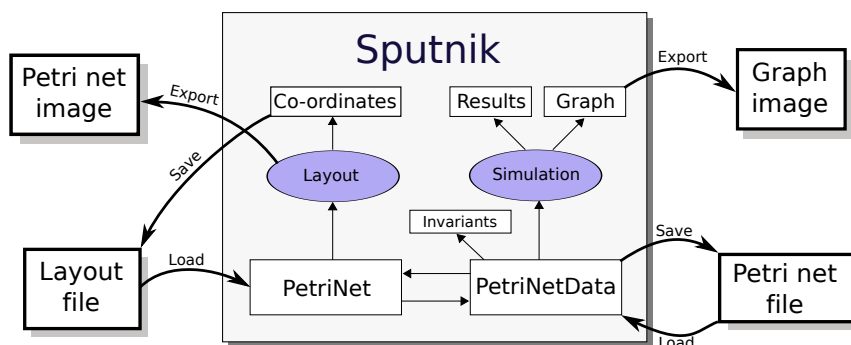


Figure 3.1: A basic data flow diagram showing the library’s inputs, outputs and core internal structure. The two core data containers, **PetriNet** and **PetriNetData**, are further discussed in sections 5.3 and 5.4

We advise that you read through the Examples to get a feel for how the library works: For full instructions on using the GUI, see Chapter 4; if you prefer to write your own scripts, see Chapter 5 for documentation covering the relevant methods and variables.

4. Using the library: GUI

All **Sputnik** functionality is readily accessible through the graphical user interface (GUI).

Figure 4.1 shows a general view of the main window of the GUI.

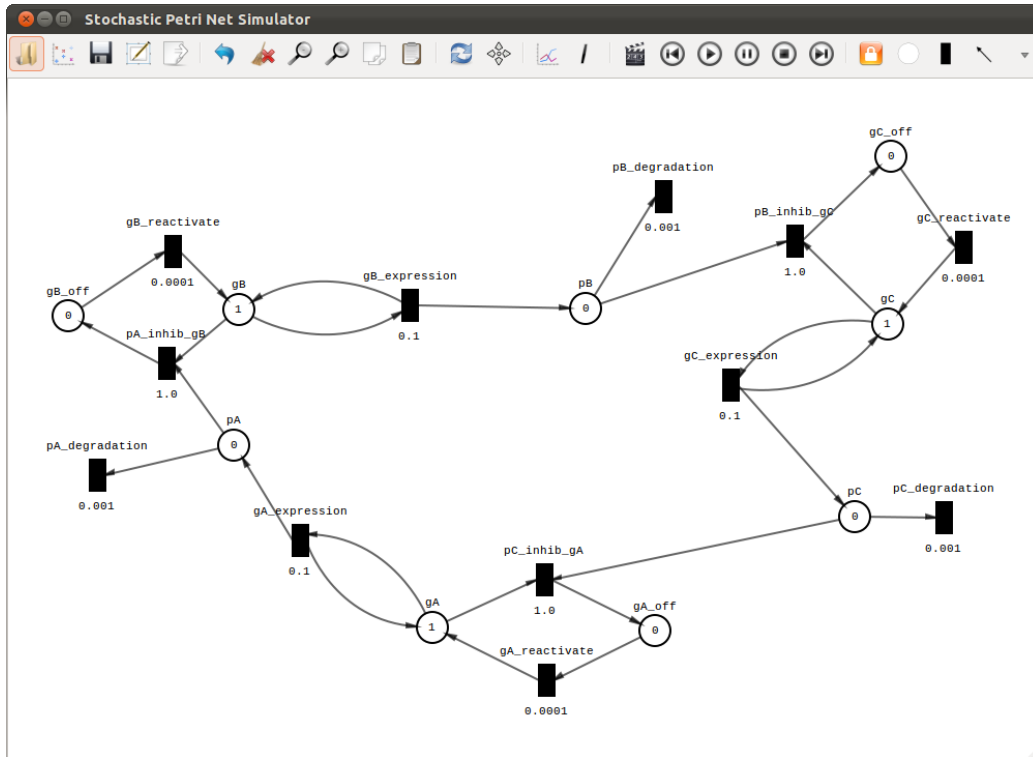


Figure 4.1: The main window of the GUI, shown displaying the “Repressilator” model^[13].

All of the library’s capabilities can be accessed quickly and easily, via the row of icons in the menu bar above the drawing area. All of the icons used are freely available for private and commercial use under a Creative Commons Attribution-Share Alike^[14] (CC BY-SA 3.0) license from the Open Icon Library^[15]. Table 4.1 provides a short explanation for each of the icon buttons in the main window (Figure 4.1).














































Icon	Shortcut	Description
	Ctrl + O	Open Petri net
	Ctrl + I	Open layout
	Ctrl + S	Save Petri net
	Ctrl + E	Save layout
	Ctrl + P	Export Petri Net
	Ctrl + Z	Undo
	Del	Delete
	- (minus)	Zoom out
	+ (plus)	Zoom in
	Ctrl + C	Copy
	Ctrl + V	Paste
	F5	Refresh
	Alt + L	Layout algorithms
	Alt + D	Simulation plot
	Alt + I	Calculate invariants
	Alt + G	Token Game simulation
	B	Previous step
	R	Play
	P	Pause
	S	Stop
	F	Next step
	Ctrl + L	Lock
	F6	Add Place
	F7	Add Transition
	F8	Add Standard Arc
	F9	Add Inhibitory Arc
	F10	Add Test Arc
	Alt + F4	Exit
	Esc	Abort


Table 4.1: Summary of the button icons found in the GUI main toolbar







4.1 Basic functions

Loading and saving Petri nets can be loaded  from *TXT*, *PNML* or *SBML* files, and saved  in any of these formats. The current layout can be saved to file  and loaded  again in the future. The visualisation of the net can be saved as an image file, in PDF, PS or SVG format.

Visualisation On loading a file, the Petri net will be laid out by a spectral graph drawing algorithm using the default settings. To configure the layout, click  to open the layout window. *Width* and *Height* determine the drawing area size, while the *Border* option prevents net components from being drawn too close to the edges of the drawing area. *Displacement Radius* defines the minimum distance between components. Components can also be positioned manually by clicking and dragging. Multiple components can be selected at once by holding **Ctrl** while drawing a box around an area. Zoom in and out using  and .

Editing the Petri net Add components by clicking on the correct icon (place ; transition ; standard arc , test arc ; inhibitory arc ) and then clicking to place the component on screen. When placing arcs, the first click places the start of the arc and the second click places the arrow head end. Standard arcs must connect a place to a transition or vice versa, while other arcs must connect a place to a transition only; acceptable connections will be highlighted **green**, unacceptable connections **red**. Components can be copied and pasted either using keyboard shortcuts or by clicking on  and . The copied component will be overlaid on the original, ready to be manually repositioned. Components can be deleted using **Del** or clicking . Deleting a place or transition will also delete the arcs connected to it. Click  to lock the current state, and to  undo an action. To modify a component, double click the component to bring up a properties window and enter the desired changes.

***P*- and *T*-invariants** Calculate and display the invariants by clicking .

Animation To display an animation of the Petri net simulated by the Gillespie algorithm, click . The animation shows each iteration of the simulation, highlighting the event that occurred. The transition, place(s) and arc(s) involved are highlighted in red, and the markings are updated appropriately. The animation is controlled by a player-like menu, giving options to jump forward () or backward () one iteration, play continuously (), pause (), or stop (). There is also a progress bar displayed below the icons, which can be clicked to jump to a specific position in the simulation. Figure 4.2 shows a view of an animation of the Lotka-Volterra model in progress.

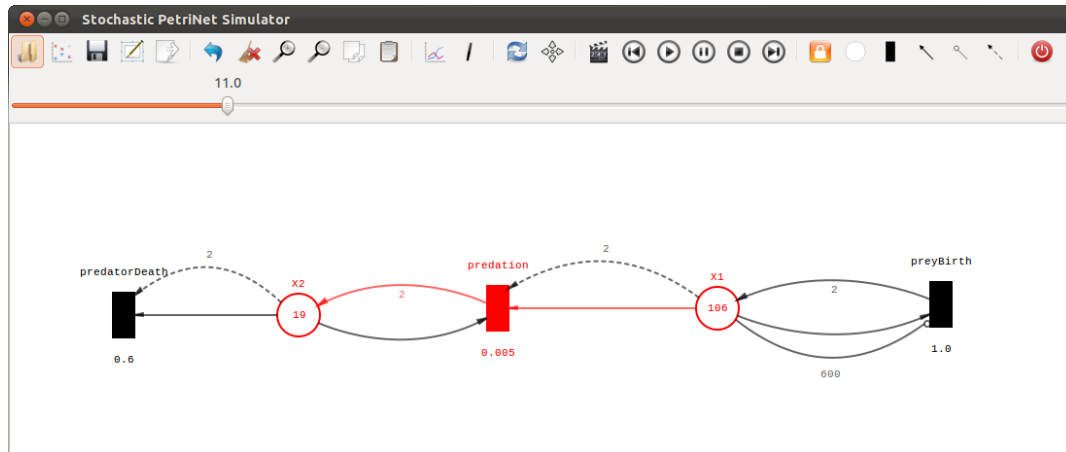


Figure 4.2: Petri net simulation animation The “predation” transition, which involves both places, “X1” and “X2” is firing.

4.2 Simulation

Figure 4.3 shows the simulation window.

Configuration: Simulation

Algorithm: ☒ Gillespie ☐ Tau Leap

Runtime: 50

Time Step: 1

Number of Simulations: 1

Epsilon (Tau Leap only): 0.03

Control Parameter (Tau Leap): 10

Number of SSA Runs (Tau Leap): 100

Plot Settings:

Title: Simulation ☒ Show Title

X Label: Runtime

Y Label: Marking

Line Width: 1

Legend Position: 0 ☒ Show Legend

☐ Create Subplots ☒ Automatic Position Allocation

Subplot Rows and Columns: 1 1

Subplot Settings:

Title: ☒ Show Subtitle

X Label: Runtime

Y Label: Marking

Subplot Position: 0

Legend Position: 0 ☒ Show Subplot Legend

Line Settings:

Legend Text:

Colour: ☒ Automatic Colour Allocation

What should be displayed within the diagrams?

Simulations:

Plot	Simulation	Index
<input checked="" type="checkbox"/>	Simulation 1	1

Places:

Plot	Place-Key	Place-Label
<input checked="" type="checkbox"/>	gC_off	gC_off
<input checked="" type="checkbox"/>	gB_off	gB_off
<input checked="" type="checkbox"/>	pB	pB
<input checked="" type="checkbox"/>	pC	pC
<input checked="" type="checkbox"/>	pA	pA
<input checked="" type="checkbox"/>	gC	gC
<input checked="" type="checkbox"/>	gB	gB
<input checked="" type="checkbox"/>	gA	gA
<input checked="" type="checkbox"/>	gA_off	gA_off

Figure 4.3: Simulation window showing simulation and plotting options.

Simulation options In most cases the Gillespie algorithm should be used, and the only settings to adjust are *Run Time* and *Time Step*. Multiple runs can be carried out at once by changing the *Number of Simulations* option. See section 5.7 for information on selecting a simulation algorithm and adjusting simulation parameters.

Plot options These options control settings for the whole plot, such as title, line width and axis labels. Legend position is controlled in the same way as for the `matplotlib` package. Save alterations by clicking *Save Settings*. The default is a single plot; create subplots by checking the *Create Subplots* button.

Subplot options These options control the settings for each individual subplot. If multiple simulations have been performed when the “Create Subplots” option is selected, each

simulation will be rendered as a subplot. If only one simulation was run, subplots will be generated for each species' timecourse. Save alterations by clicking *Save Settings*.

Line options These options settings for each individual timecourse within a plot, for example line colour and the label to be displayed in the legend. Checkboxes determine which timecourses to plot. To customise line colour, deselect the *Automatic Colour Allocation* checkbox and then use the colour selector. Save alterations by clicking *Save Settings*.

5. Using the library:

command-line

5.1 Help documentation

PyDoc^[16] is a module within Python used for documentation purposes. It generates documentation for classes and modules automatically from “docstring” comments within the code and can be displayed on the console, exported as HTML files or handed directly to a Web browser.

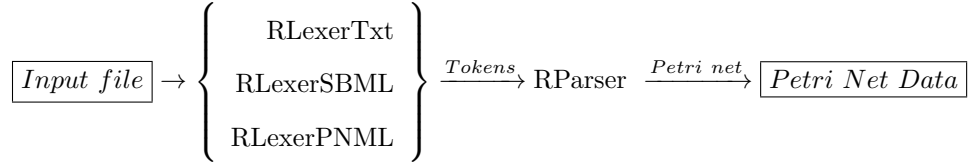
The *documentation* folder contains PyDoc-generated HTML files providing extensive documentation for all the classes, methods and variables of the **Sputnik** code. To consult PyDoc documentation from within the Python Interactive Environment, issue one of the commands:

```
\begin{small} 1
## For information about a specific class: 2
help(<module_name>.<ClassName>) 3
4
## For information about a specific method: 5
help(<module_name>.<ClassName>.<method_name>) 6
7
\end{small} 8
```

5.2 Import and export of Petri nets

5.2.1 Loading a Petri net

In **Sputnik**, file input is parsed to **PetriNetData** via:

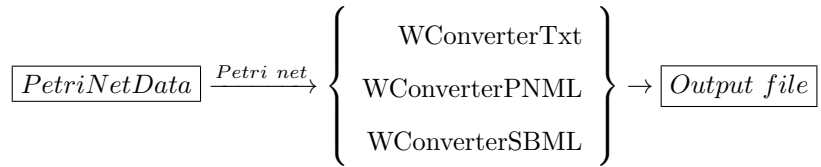


Loading a Petri net in this way, through iPython or using a script, requires the import of two classes, a general parser `RParser()` and a lexer that is specific to the file type you are opening, i.e. `RLexerTxt()`, `RLexerSBML()` or `RLexerPNML()`. As an example, the following commands will parse a `.txt` file to a **PetriNetData** object `p`:

```
import sputnik_io 1
2
lexer = sputnik_io.RLexerTxt() 3
parser = sputnik_io.RParser() 4
inputfile = open('examplefile.txt') 5
tokens = lexer.lex(inputfile) 6
parser.data = tokens 7
parser.parse() 8
p = parser.output 9
```

5.2.2 Saving a Petri net

In order to save a Petri net stored in a **PetriNetData** object `p`, the following process is implemented:



To perform these actions as part of a script, for example to save to PNML format, the following commands can be issued:

```
import sputnik_io 1
2
converter = sputnik_io.WConverterPNML(p) 3
converter.save('outputfilename.pnml') 4
```

5.2.3 Text format specification

Writing a plain text file is often the fastest way to define a new Petri net for use with **Sputnik**. The text format closely mirrors a mathematical Petri net definition, and is specified below.

Places – a list of strings that represent identifiers for each Petri net place.

Required: Yes

Syntax: `p / places`

Example: `p = [prey,predator]`

Transitions – a list of strings identifying transition names.

Required: Yes

Syntax: `t / transitions`

Example: `t = [preyBirth,predation,predatorDeath]`

Pre – a matrix describing the weights of arcs connecting places to transitions.

Required: Yes

Syntax: `pre / pre_arcs / pre arcs`

Example: `pre = [[1,0],[1,1],[0,1]]`

Post – a matrix describing the weights of arcs connecting transitions to places.

Required: Yes

Syntax: `post / post_arcs / post arcs`

Example: `post = [[2,0],[0,2],[0,0]]`

Initial marking – a list of integers that count the number of tokens in each place prior to simulation. Should be the same length as places.

Required: No (set to 0 if absent)

Syntax: `m / markings`

Example: `m = [100,20]`

Rates – a list of numbers that represent the stochastic rate constants associated with each transition.

Required: No (set to 0 if absent)

Syntax: `r / rates`

Example: `r = [1,0.005,0.6]`

Capacities – a list of integers defining the maximum number of tokens allowed in each place. If present, must have the same length as places.

Required: No

Syntax: `c / capacities`

Example: `c = [500,500]`

Inhibitory arcs – a matrix describing a logical test performed on place markings to enable or disable a transition. Transitions will not fire if stated markings are above the values stated in this matrix (zeros are ignored), otherwise they fire as normal.

Required: No

Syntax: `i / inhib / inhibitory_arcs / inhibitory arcs`

Example: `inhib = [[600,0],[0,0],[400,0]]`

Test arcs – a matrix describing a logical test performed on place markings to enable or disable a transition. Transitions will not fire if stated markings are below or equal to the values stated in this matrix, otherwise they fire as normal.

Required: Yes

Syntax: `test / test_arcs / test arcs`

Example: `test = [[2,0],[2,0],[0,2]]`

5.2.4 Standalone file conversion

Standalone file conversion can be performed using `fconvert`. This must be run with two parameters, input and output filenames (relative or absolute paths), in the form:

```
$ ./fconvert inputfile.txt outputfile.xml
```

File extensions `.txt`, `.sbml`, `.pnml` and `.xml` are automatically detected, otherwise the user will be prompted to state the file type.

Note: the ability to read and write SBML requires the freely available `libSBML` Python API (<http://sbml.org/Software/libSBML/docs/python-api/>).

5.3 Manually inputting a Petri net

A stochastic Petri net is defined by the n -tuple $N = \{P, T, Pre, Post, M_0, R, C^*, Test^*, Inhib^*\}$, where the starred elements are optional. A Petri net model may be input manually, by creating a `sputnik_petrinet.PetriNetData()` object and then setting its instance variables as the appropriate NumPy matrices and arrays. The *Pre* and *Post* matrices are stored in a `sputnik_petrinet.Stoich()` object that is set as the `stoichiometry` instance variable of the `PetriNetData()` object. Refer to the TXT input format specification for the required dimensions of the data elements. A `PetriNetData` object `p` should be created using the following template:

```
import numpy as np 1
import sputnik_petrinet 2
3
p = sputnik_petrinet.PetriNetData() 4
p.stoichiometry = sputnik_petrinet.Stoich() 5
p.stoichiometry.pre_arcs = np.matrix(<post_arc_matrix>, dtype=int 6
    )
p.stoichiometry.post_arcs = np.matrix(<post_arc_matrix>, dtype= 7
    int)
8
p.places = np.array(<place_labels>) 9
p.transitions = np.array(<transition_labels>) 10
p.rates = np.array(<rates>, dtype=float) 11
p.initial_marking = np.array(<initial_marking>, dtype=int) 12
13
```

```

## OPTIONAL 14
p.capacities = np.array(<capacities>, dtype=int) 15
p.test_arcs = np.matrix(<post_arc_matrix>, dtype=int) 16
p.inhibitory_arcs = np.matrix(<post_arc_matrix>, dtype=int) 17
18
## Calculate net properties (required for simulations) 19
p.stoichiometry.calculate_stoichiometry_matrix() 20
p.stoichiometry.calculate_dependency_matrix() 21
p.stoichiometry.calculate_consumed() 22
p.stoichiometry.calculate_species_hors() 23

```

5.4 Create a PetriNet object

Whereas the **PetriNetData** object created in the above sections contains a mathematical representation of a Petri net (data elements are matrices and arrays), a **PetriNet** object contains representations of each individual component (place, transition or arc) which each have certain properties (labels, markings, rates, weights etc.). **PetriNet** objects are used by the visualisation part of the library, while the simulation, invariant calculation and parsers use **PetriNetData** objects. The corresponding **PetriNetData** object of a **PetriNet** object is accessed through its **petrinetdata** variable. All component classes inherit from a general **Component** parent class.

To create a **PetriNet** object **pn** from a **PetriNetData** object **pn**:

```

## instantiate a PetriNet object 1
pn = sputnik_petrinet.PetriNet() 2
3
## assign the PetriNetData object 4
pn.petri_net_data = p 5
6
## create the components 7
pn.convert_components() 8

```

5.4.1 Edit a PetriNet object

The framework permits individual components to be instantiated from the command line and added to a **PetriNet** object. After editing a Petri net in this way, the **PetriNet** class method

`convert_matrices()` method must be run to update the corresponding `PetriNetData` object. Example code is presented below for completeness, although we strongly suggest the user either uses the GUI for this, or just alters the `PetriNetData` properties directly, then reruns the converter to update the components. Another simple solution would be to alter a `TXT` input file appropriately and then reimport it.

```

import sputnik_petrinet 1
2
## create the single components of a petri net 3
4
## create a place 5
p = sputnik_petrinet.Place([100.0, 50.0], 15., [0., 0., 0.], 6
    [255., 255., 255.])
p.label = "Place" 7
p.key = "Place" 8
p.marking = <marking> 9
10
## create a transition 11
t = sputnik_petrinet.Transition([50.0, 50.0], [15, 30], [0., 0., 12
    0.], [0., 0., 0.])
t.label = "Transition" 13
t.key = "Transition" 14
t.rate = <rate> 15
16
## create a pre arc that connects the place and transition 17
a = sputnik_petrinet.Arc() 18
a.line_type = sputnik_petrinet.Arc.LINE_TYPE_STRAIGHT 19
a.label = str("StandardArc") 20
a.key = "StandardArc" 21
a.origin = p 22
a.target = t 23
a.weight = <weight> 24
25
## Other constructors: TestArc(), InhibitoryArc() 26
## Other line types: LINE_TYPE_ARC_UPPER, LINE_TYPE_ARC_LOWER 27
28
# instantiate a PetriNet object 29
pn = sputnik_petrinet.PetriNet() 30

```

```

31
32 ## add all the components
33 pn.add_place(p)
34 pn.add_transition(t)
35 pn.add_arc(a)
36
37 # update the PetriNetData object
38 pn.convert_matrices()

```

5.5 Petri net visualisation

5.5.1 Obtaining Petri net layout coordinates

Spectral algorithm

To obtain raw layout coordinates for a `PetriNetData` object `p`:

```

1 import spectral_a
2
3 v = spectral_a.Spectral()
4 v.petri_net = p
5 v.get_petri_net()
6
7 ## OPTIONAL
8 v.width = <width>          ## Default = 1000
9 v.height = <height>        ## Default = 1000
10 v.border = <border>         ## Default = 20
11 v.d_radius = <d_radius>     ## Default = 60
12
13 coordinates = v.calculate()

```

The `width`, `height`, `border` and `d_radius` variables have default values within the `spectral_a` class but can be overridden if the user desires.

Force-directed algorithm

For the force directed algorithm the process is the same other than the `iterations` variable can also be optionally set:

```

import force_a 1
2
v = force_a.ForceDirected() 3
v.petri_net = p 4
v.get_petri_net() 5
6
## OPTIONAL 7
v.width = <width> ## Default = 1000 8
v.height = <height> ## Default = 1000 9
v.border = <border> ## Default = 20 10
v.iterations = <iterations> ## Default = 50 11
12
coordinates = v.calculate() 13

```

The number of iterations required to reach an acceptable layout varies between graphs. For small graphs (on the order of tens of vertices) 100 iterations may be sufficient, for graphs containing hundreds of vertices a thousand or more iterations may be required. Some trial and error may be required.

5.5.2 Drawing a Petri net

It is possible to draw a Petri net without using the GUI.

To perform these actions as part of a script for a `PetriNet` object `pn`:

```

import spectral_a 1
2
v = spectral_a.Spectral() 3
v.petri_net = pn.petri_net_data 4
v.get_petri_net() 5
6
## OPTIONAL 7
v.width = <width> ## Default = 1000 8
v.height = <height> ## Default = 1000 9
v.border = <border> ## Default = 20 10
v.d_radius = <d_radius> ## Default = 60 11
12
v.calculate() 13
## set the positions and draw the Petri net 14

```

5.5.3 Drawing an undirected graph

To draw a graph which is not a Petri net, the graph's *Adjacencymatrix* must be available.

```

import numpy as np
import spectral_a

v = spectral_a.Spectral()

## OPTIONAL

v.width = <width>          ## Default = 1000
v.height = <height>        ## Default = 1000
v.border = <border>         ## Default = 20

## In this context places refers to a unique identifier for each
vertex in the graph
v.places = np.array(['a', 'b', 'c', 'd', 'e'])

## The adjacency matrix is ordered as the places array:
##
      a    b    c    d    e
v.adjacency = np.matrix([[ 0,  0,  1,  1,  0], #a
                          [ 0,  0,  0,  1,  1], #b
                          [ 1,  0,  0,  0,  0], #c
                          [ 1,  1,  0,  0,  0], #d
                          [ 0,  1,  0,  0,  0]]) #e

v.render_graph()
```

5.5.4 Save an image of the visualisation

To save an image of a petri net visualisation, a `gtk.DrawingArea` object needs to be instantiated which is used to represent the visualisation of the graph. The `gtk.GraphicsContext`, which is bound to the drawing area, is assigned to the petri net via the `PetriNet.draw(ctx)` method and the petri net components will be drawn onto the drawing area. The drawing area can then be assigned to an instantiation of `ExportDrawingArea`. Finally, an export

method can be called to save an image of the drawing area. Here is code for an example class to carry out these functions:

```

class Export:
    """
    Simple example class of how the visualisation of the graph can be
    exported to a defined location.
    """
    def __init__(self, pn):
        """
        Constructor of the sample Export class that defines the basic
        settings for a gtk.Window.
        """
        ## set petri net
        self._pn = pn    ## pn is a PetriNet object
        ## instantiate a gtk.Window
        window = gtk.Window()
        ## instantiate a gtk.DrawingArea
        self._drawing_area = gtk.DrawingArea()
        self._drawing_area.size(900, 600)
        ## embed the drawing area into a viewport
        viewport = gtk.Viewport()
        viewport.connect("expose-event", self.update_drawing_area)
        viewport.add(self._drawing_area)
        ## embed the viewport into the window
        window.add(viewport)
        window.show_all()
        window.show()

    def update_drawing_area(self, widget, event):
        """
        Update the drawing area in case of an expose event.
        """
        ctx = self._drawing_area.window.cairo_create()
        self._pn.draw(ctx)
        ctx.clip()

    def export(self):
        """
        Export the visualisation of the graph as a PDF to a defined
        location.
        """
        ## choose absolute path

```



```

file_chooser = gtk.FileChooserDialog("Export File", None, gtk. 38
    FILE_CHOOSER_ACTION_SAVE, (gtk.STOCK_CANCEL, gtk.
    RESPONSE_CANCEL, gtk.STOCK_OK, gtk.RESPONSE_OK))
if file_chooser.run() == gtk.RESPONSE_OK: 39
    path = file_chooser.get_filename() 40
    ## export graph as pdf 41
    exp = drawing_area.export.DrawingAreaExport() 42
    exp.drawing_area = self._drawing_area 43
    exp.path = path 44
    exp.export_as_pdf() 45

```

5.6 Calculating the invariants

The P - and T -invariants of a net are calculated by a `sputnik_petrinet.PTInvariants()` object associated with a `sputnik_petrinet.PetriNetData()` object. The invariants can then be accessed via the `p_invariants` and `t_invariants` instance variables of this object. To calculate and display the invariants of a `PetriNetData()` object, `p`, issue the following commands:

```

import sputnik_petrinet 1
2
i = sputnik_petrinet.PTInvariants() 3
i.set_petri_net(p) 4
i.calculate_p_invariants() 5
i.calculate_t_invariants() 6
7
print i.p_invariants 8
print i.t_invariants 9

```

5.7 Running a simulation

Sputnik can simulate Petri nets using either an exact stochastic simulation algorithm (SSA), the Gillespie, or an approximate SSA, the tau leap. Exact SSAs sample the waiting time to each new reaction explicitly, using exact reaction hazards. Approximate SSAs use approximate hazards, sacrificing some accuracy for a shorter runtime. The Gillespie SSA can handle test arcs, inhibitory arcs and capacities, and is the recommended algorithm for most situations (it must be used if knowing the time of occurrence of each and every event is

desired). The tau leap SSA may be used when the model to be simulated involves large species abundances. Under these conditions, the tau leap offers significant speed advantages over the Gillespie, while still providing a very accurate simulation. The tau leap cannot be used with test arcs, inhibitory arcs or capacities.

`sputnik_simulation.Gillespie()` and `sputnik_simulation.TauLeap()` are subclasses of `sputnik_simulation.Algorithm()`, an abstract base class providing the general instance variables `algorithm`, `num_runs`, `num_iterations`, `run_time` and `time_step`, plus the general interface method `run_simulation()`. When a simulation algorithm subclass is instantiated, `algorithm` is automatically set to the correct algorithm type and `num_runs` is initialized as the default value 1. The simulation may be run for a specified `num_iterations` or until a certain time t is reached, with output data being stored at regular timepoints. In the latter case, both `run_time` and `time_step` should be set.

Each simulation run initialises an object from the appropriate `sputnik_simulation.SimData()` subclass, namely `sputnik_simulation.GillespieData()` or `sputnik_simulation.TauLeapData()`.

Five types of output data are stored in this class:

- **times**: a 1D NumPy array of timepoints of length n
- **markings**: a 2D NumPy array ($n \times P$) of net markings at each timepoint
- **events**: usually a 2D NumPy array ($(n - 1) \times P$) of event firing frequencies between each timepoint. For a Gillespie simulation run `by_iteration`, this is a 1D array where each entry gives the index of the event that fired that iteration.
- **event_freqs**: a list (length T) giving the total frequency with which each transition fired
- **iterations**: the number of iterations simulated

Each `SimData()` object is appended to a list, which is set as the `simulation_data` instance variable of the `Gillespie()` or `TauLeap()` object.

5.7.1 Running the Gillespie SSA

```
## Initialise a simulation object 1
import sputnik_simulation          2
g = sputnik_simulation.Gillespie() 3
```

```

g.petri_net = p      ## p is a PetriNetData object
4
5
## OPTIONAL set the number of runs if multiple runs are desired
6
g.num_runs = <num_runs>
7
8
## EITHER choose a number of iterations
9
g.num_iterations = <num_iterations>
10
11
## OR set a run time and time step
12
g.run_time = <run_time>
13
g.time_step = <time_step>
14
15
## Run the simulation
16
g.run_simulation()
17
output = g.simulation_data
18
19
## You may print the raw data for a particular run (uses zero
    indexing)
20
print output[<run>].times
21
print output[<run>].markings
22
print output[<run>].events
23
print output[<run>].event_freqs
24
print output[<run>].iterations
25

```

5.7.2 Running the tau leap SSA

There are three additional parameters for the tau leap algorithm compared to the Gillespie. The default values will be suitable in most circumstances. For a full explanation of the algorithm and its parameters, consult Cao *et al.* (2006)^[10].

1. **epsilon** (default = 0.03; sensible range = 0-0.1) is a value between 0 and 1 that adjusts the stringency of the algorithm, ϵ . The smaller the value of ϵ , the smaller the calculated values of τ , since ϵ is the maximum permitted relative change in any reaction hazard each timestep.
2. **control_parameter** (default = 10; sensible range = 2-20) determines the threshold place marking that will cause the Gillespie algorithm to be invoked.

3. `num_ssa_runs` (default = 100) determines how many iterations of the Gillespie algorithm should be run each time the marking of a place drops below the control parameter value.

```
## Initialise a simulation object 1
import sputnik_simulation 2
t = sputnik_simulation.TauLeap() 3
t.petri_net = p    ## p is a PetriNetData object 4
5
## OPTIONAL set the number of runs if multiple runs are desired 6
t.num_runs = <num_runs> 7
8
## EITHER choose a number of iterations 9
t.num_iterations = <num_iterations> 10
11
## OR set a run time and time step 12
t.run_time = <run_time> 13
t.time_step = <time_step> 14
15
## OPTIONAL change the simulation parameters from default 16
t.epsilon = <epsilon> 17
t.control_parameter = <control_parameter> 18
t.num_ssa_runs = <num_ssa_runs> 19
20
## Run the simulation 21
t.run_simulation() 22
output = t.simulation_data 23
24
## You may print the raw data for a particular run (uses zero
    indexing) 25
print output[<run>].times 26
print output[<run>].markings 27
print output[<run>].events 28
print output[<run>].event_freqs 29
print output[<run>].iterations 30
```

5.8 Plotting simulation diagrams

The three classes `Trajectory`, `Diagram` and `DiagramVisualisation` are needed to create a flexible representation of simulation results. A `Trajectory` object is used to define a single trajectory (place timecourse) and its properties; a `Diagram` object combines multiple `Trajectory` objects and defines properties of a diagram or subplot and finally `DiagramVisualisation` is used to define the actual representation of the defined `Diagram` objects. The following script is for visualising an `Algorithm` subclass object `sim` of a `PetriNetData` object `p`.

```
import matplotlib_visualisation 1
2
# create a PetriNetData object p and simulate it to get sim 3
... 4
5
# instantiate a general object that includes the single diagrams 6
vis = matplotlib_visualisation.DiagramVisualisation() 7
# set properties 8
vis.title = <title> 9
vis.legend_visibility = True 10
vis.title_visibility = True 11
vis.subplots = True 12
vis.line_width = <line width> ## Default = 1 13
14
# iterate through all simulation runs 15
for i in range(<num_sims>): 16
    # instantiate an object that combines single trajectories 17
    d_obj = matplotlib_visualisation.Diagram() 18
    # set properties 19
    d_obj.title = "Simulation_" + str(i + 1) 20
    d_obj.xlabel = "Runtime" 21
    d_obj.ylabel = "Markings" 22
    d_obj.legend_position = 0 23
    d_obj.title_visibility = True 24
    d_obj.legend_visibility = True 25
    # iteration through all trajectories 26
    for j in range(len(p.places)): 27
        # instantiate a trajectory object 28
```

```

t_obj = matplotlib_visualisation.Trajectory() 29
# set properties 30
t_obj.legend_text = p.places[j] + "└─└" + d_obj.title 31
t_obj.auto_color_allocation = True 32
t_obj.x_data = sim.simulation_data[i].times 33
t_obj.y_data = sim.simulation_data[i].markings[:,j]) 34
# add trajectory object to the diagram object 35
d_obj.add(t_obj, t_obj.legend_text) 36
# add diagram object to the visualisation object 37
vis.add(d_obj, d_obj.title) 38
# visualise diagrams 39
vis.plot() 40

```

6. Examples

In the following sections, three example systems are analysed using the **Sputnik** framework. Example 1 is run via the graphical user interface while Example 2 is demonstrated through a command-line terminal. Example 3 is a demonstration of adjusting the graph layout parameters.

6.1 Example 1 : Repressilator

The repressilator is a synthetic biochemical network designed for *Escherichia coli*. The system acts as a cellular clock, exhibiting regular oscillatory behaviour involving three protein-coding genes^[13]. Figure 6.1 shows a Petri net representing this network.

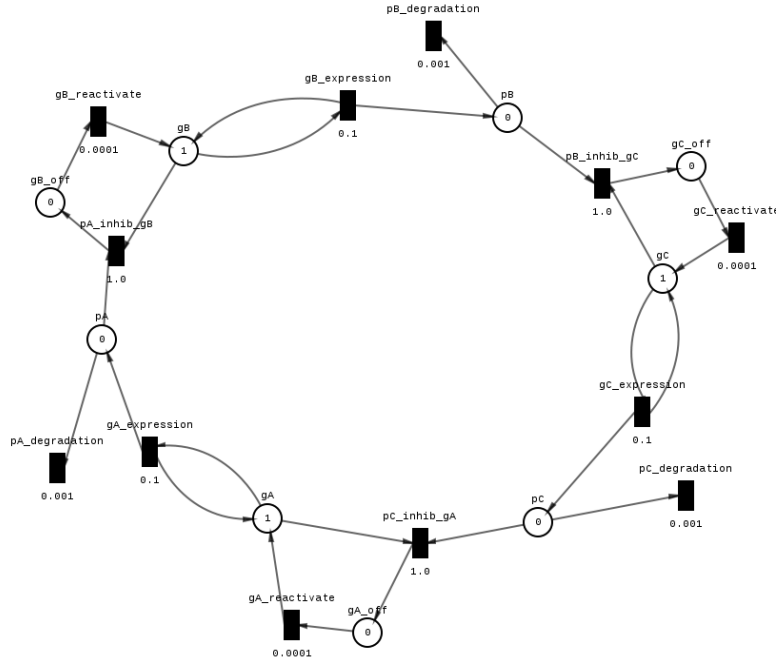


Figure 6.1: The Repressilator genetic system.

The program is supplied with the following example file which represents the above Petri net:

```
## Repressilator text file ##

places= [gA, pA, gA_off, gB, pB, gB_off, gC, pC, gC_off]

t = [gA_expression, pA_degradation, pA_inhib_gB, gB_reactivate,
      gB_expression, pB_degradation, pB_inhib_gC, gC_reactivate,
      gC_expression, pC_degradation, pC_inhib_gA, gA_reactivate]


pre arcs= [[1, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 1, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1],
            [0, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 0],
            [1, 0, 0, 0, 0, 0, 0, 1, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0]]


post_arcs =[[1, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 1, 0, 0, 0],
            [0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 1, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1],
            [0, 0, 0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 1, 1, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 0, 0, 0, 0, 0],
            [1, 0, 0, 0, 0, 0, 0, 0, 0]]

rates = [0.1, 0.001, 1, 0.0001, 0.1, 0.001,
          1, 0.0001, 0.1, 0.001, 1, 0.0001]

marking = [1, 0, 0, 1, 0, 0, 1, 0, 0]
```

Figure 6.2: A plain text file which specifies the repressilator Petri net model.

To load this input file from the GUI, select open file , then locate and open **repressilator.txt**, included in the **Sputnik** package in the **examples/** directory.

The Petri net will now be displayed. To calculate P - and T -invariants, select the invariants icon . The following results window will then appear:

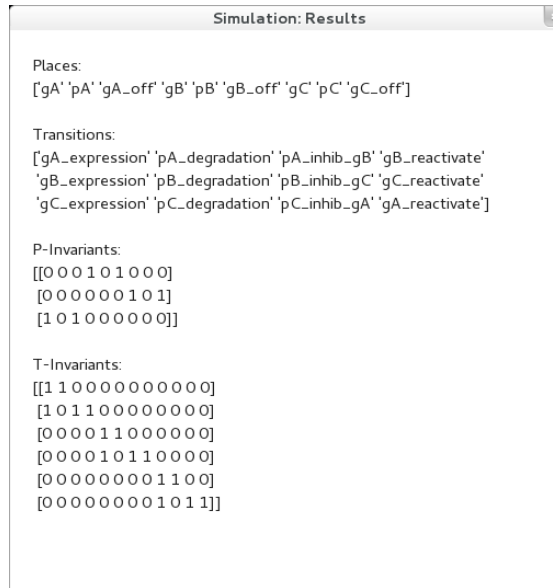



Figure 6.3: Calculated T - and P -invariants.

The Petri net can now be simulated using the Gillespie algorithm. To do this, click the simulation icon , you will then be presented with the simulation configuration options:

Configuration: Simulation

Algorithm: ☒ Gillespie ☐ Tau Leap

Runtime: 300000

Time Step: 500

Number of Simulations: 1

Epsilon (Tau Leap only): 0.03

Control Parameter (Tau Leap): 10

Number of SSA Runs (Tau Leap): 100

Plot Settings:

Title: Repressilator simulation ☒ Show Title

X Label: Runtime

Y Label: Marking

Line Width: 1

Legend Position: 0 ☒ Show Legend

☐ Create Subplots ☒ Automatic Position Allocation

Subplot Rows and Columns: 1 1

Subplot Settings:

Title: ☒ Show Subtitle

X Label: Runtime

Y Label: Marking

Subplot Position: 0

Legend Position: 0 ☒ Show Subplot Legend

Line Settings:

Legend Text:

Colour: ☐ Automatic Colour Allocation

What should be displayed within the diagrams?

Simulations:

Plot	Simulation	Index
<input checked="" type="checkbox"/>	Simulation 1	1

Places:

Plot	Place-Key	Place-Label
<input checked="" type="checkbox"/>	gC_off	gC_off
<input checked="" type="checkbox"/>	gB_off	gB_off
<input checked="" type="checkbox"/>	pB	pB
<input checked="" type="checkbox"/>	pC	pC
<input checked="" type="checkbox"/>	pA	pA
<input checked="" type="checkbox"/>	gC	gC
<input checked="" type="checkbox"/>	gB	gB
<input checked="" type="checkbox"/>	gA	gA
<input checked="" type="checkbox"/>	gA_off	gA_off

Figure 6.4: Configuration options for simulating the firing of a Petri net.

In this example, the **runtime** has been set to 300,000 and the **Step-Size** to 500. The Petri net can now be simulated by clicking **Start Simulation**. An info box will inform the user when the simulation is complete. Before plotting the results, a number of parameters can be adjusted: For the example shown below (Figure 6.6), the **Title** was changed and **Show Legend** was deselected, additionally some of the line colors were set manually, using the **Specific Plot-Options**; to save these changes, click **Save Specific Plot Setting**, then to visualise the plot, click **Show Plot**.

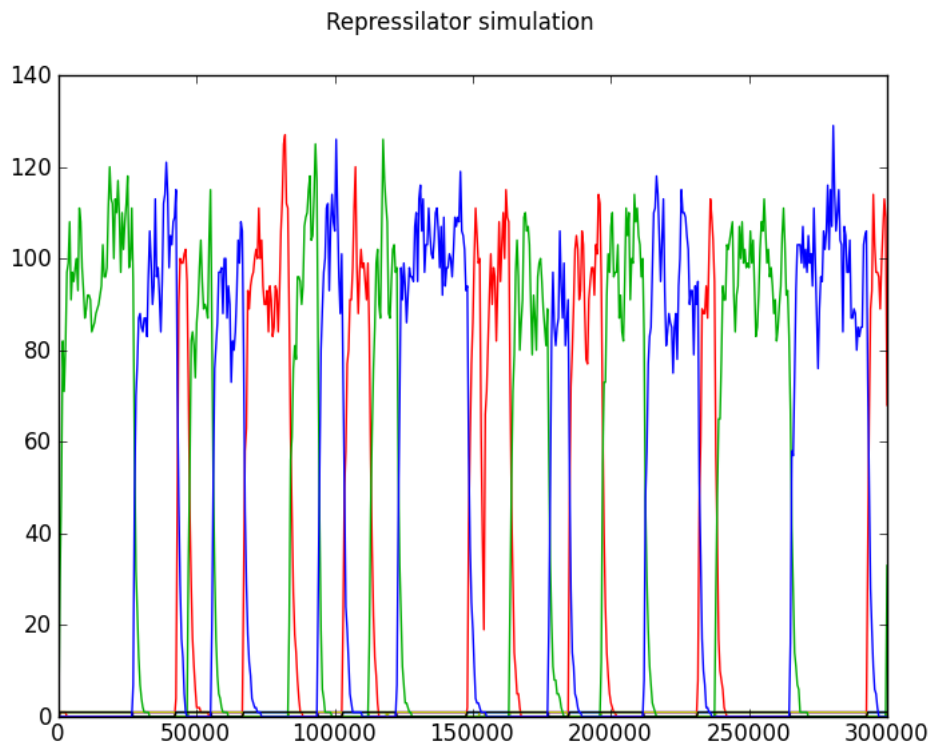


Figure 6.5: Graph output of stochastic simulation results for the repressilator Petri net model. The y -axis denotes place markings and the x -axis shows the runtime. The *red*, *green* and *blue* lines represent the three proteins in the repressilator model.

This graph can then be exported to numerous image formats, including **pdf**, **svg** and **png**.

6.2 Example 2 : Four-reaction system

A second example system, the four-reaction system^[17], will now be entered and simulated using a command-line interface. These commands can be run using iPython or as part of a Python script.

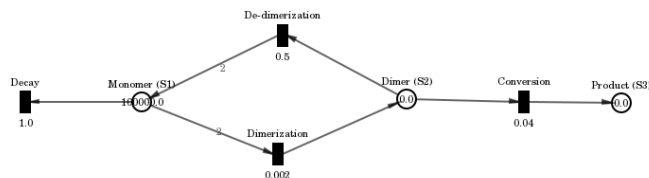


Figure 6.6: A Petri net representing the four-reaction system.

The first step required is to import specific modules of the **Sputnik** framework. In this example we will be using the tau-leap simulation method, due to the large number of molecules involved in this system. The required imports are:

```
import sputnik_petrinet 1
import petrinetdata_simulation 2
```

Other imports, required for plotting and data entry, are:

```
import matplotlib.pyplot as plt 1
import numpy as np 2
```

The Petri net can now be specified using the following statements (Note the large initial marking used in this example):

```
p = sputnik_petrinet.PetriNetData() 1
p.places = np.array(['S1', 'S2', 'S3']) 2
p.transitions = np.array(['R1', 'R2', 'R3', 'R4']) 3
p.rates = np.array([1, 0.002, 0.5, 0.04], dtype=float) 4
p.initial_marking = np.array([100000, 0, 0], dtype=int) 5
```

Pre and post arcs are stored in a separate stoichiometry class:

```
s = sputnik_petrinet.Stoich() 1
s.pre_arcs = np.matrix([[1, 0, 0], [2, 0, 0], [0, 1, 0], [0, 1, 0]], 2
                      dtype=int)
s.post_arcs = np.matrix([[0, 0, 0], [0, 1, 0], [2, 0, 0], [0, 0, 1]], 3
                      dtype=int)
s.calculate_stoichiometry_matrix() 4
s.calculate_dependency_matrix() 5
```

In preparation for Tau leap simulation, the following extra procedures need to be called:

```
s.calculate_consumed() 1
s.calculate_species_hors() 2
```

In order to run the simulation three parameters are required, `petri_net`, `run_time` and `time_step`, which can each be set as follows:

```
t = sputnik_simulation.TauLeap() 1
t.petri_net = p 2
t.run_time = 40 3
t.time_step = 0.01 4
```

Now you are ready to run the simulation and visualise the output:

```
t.run_simulation() 1
results = t.simulation.data 2
plt.plot(results[0].times, results[0].markings) 3
plt.show() 4
```

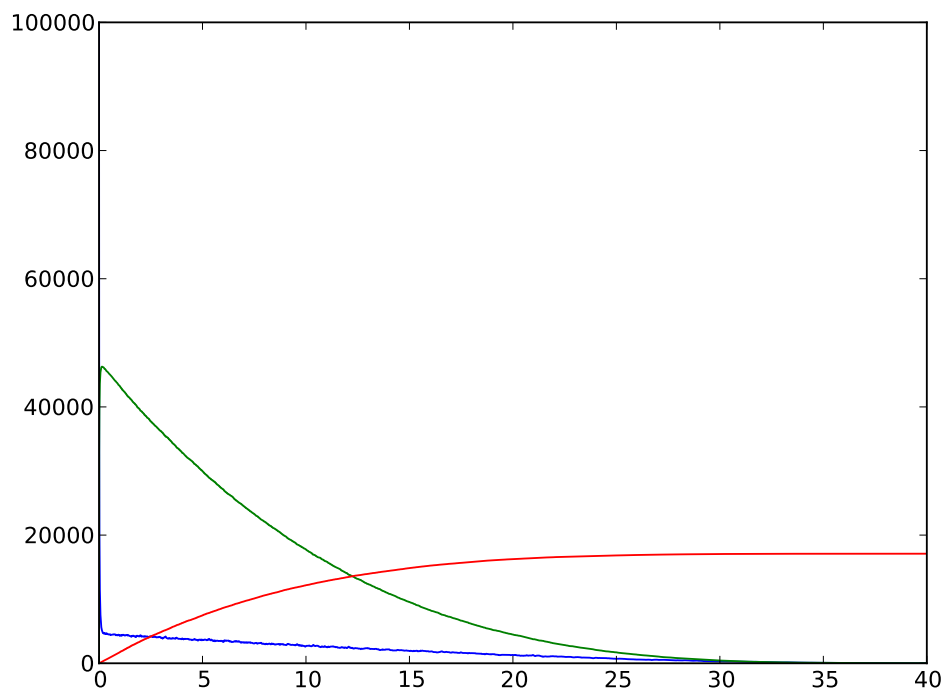


Figure 6.7: The results of Tau-leap simulation of the four-reaction system. The x -axis represents runtime and the y -axis shows place markings. Coloured lines are represent places as follows: *blue*: S1, *green*: S2, *red*: S3.

NOTE: for a more sophisticated way to plot diagrams, see section 5.8.

6.3 Example 3 : MAP kinase

The library contains two powerful layout algorithms to help the user make sense of Petri nets imported into the program. There are parameters within these algorithms that can be optimised for a given scenario, be it the requirements of a specific graph, or particular drawing area size. To demonstrate how these parameters affect the graph layout process, the MAP kinase model^[18] will be used, due to its high layout difficulty as a result of its non planar characteristic and high connectivity.

When a new Petri net is loaded to the program it is automatically laid out using the spectral algorithm with default settings. An example of the MAP kinase system under default layout conditions is shown in Figure 6.8.

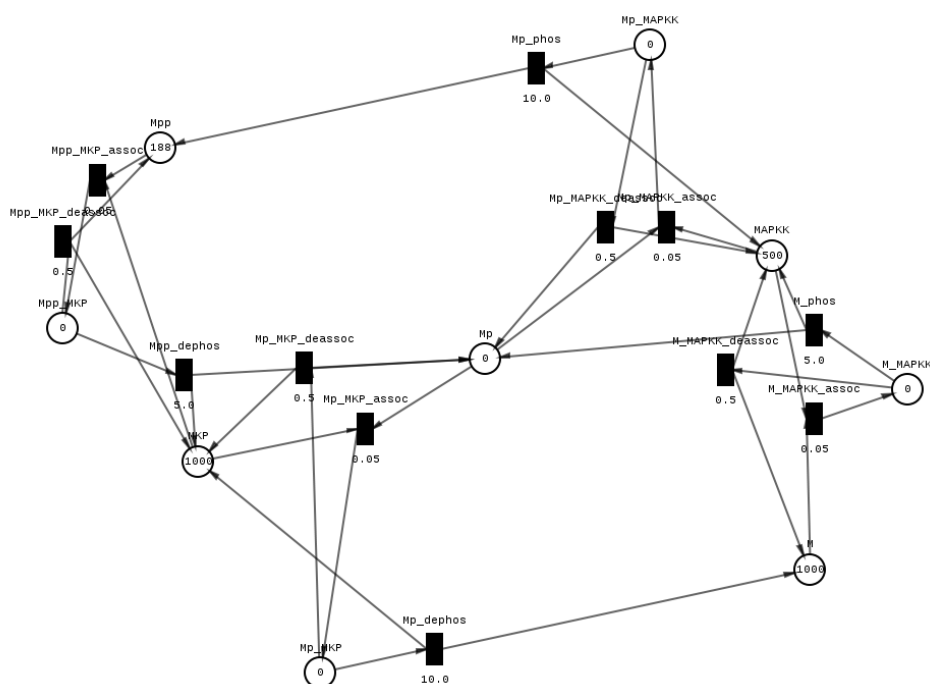


Figure 6.8: The default layout of the MAP kinase system shown here is not ideal, there is some clustering of objects and this has caused some of the object labels to overlap.

By decreasing the border size to provide more space for the layout of objects and increasing the size of the displacement radius to detect more clusters, you can generate a nicer automatic layout. The result of refining the layout parameters for the MAP kinase Petri net is shown in figure 6.9.

7. Extending the library

Sputnik was planned and designed using the object oriented paradigm (Figure 7.1). The *PetriNet* class is the heart of the framework, used to encapsulate all the other components from each other. This means that these individual components may be extended separately, without worrying about how the rest of the program will be affected, so long as the interaction with the central *PetriNet* class remains the same. In the rest of this section, advice is offered on how to extend aspects of the library.

7.1 Extending the GUI

Sputnik uses the Model-View-Controller (MVC) architecture, which allows the integration of new components into the GUI without major changes to the whole system^[19]. It consists of three main components: model, view and controller^[19,20]. *Views* form visual aspects of the GUI and *controllers* manage user interactions with views. Each view and controller is registered with a *model*. In **Sputnik**, there is only one model, and it manages all the interactions between the views and controllers. The views and controllers are *observers* of the model. If there is a data change in a view or controller object, that object notifies the model of the change, and the model in turn notifies its observers of the same changes, synchronising the current state of the program. Figure 7.2 shows an overview of the MVC architecture.

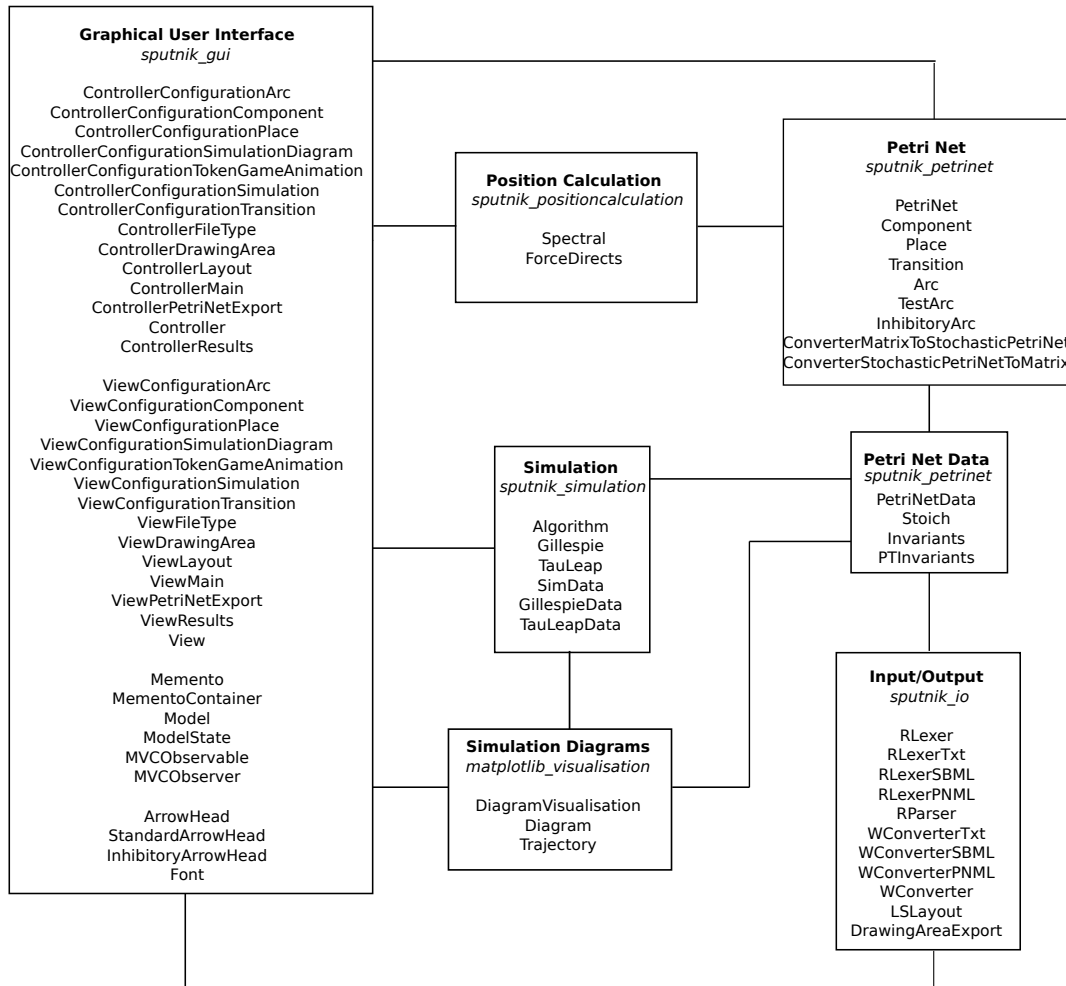


Figure 7.1: Simplified class diagram illustrating the overall architecture of **Sputnik**.
Key: **group of classes**, *module name*, other class names.

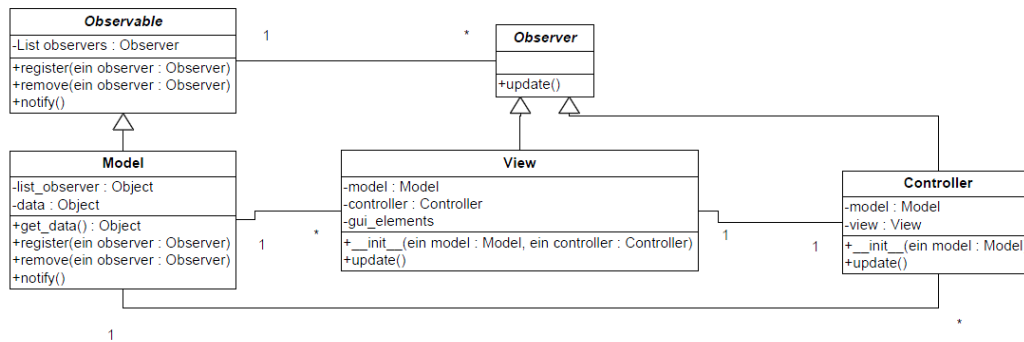


Figure 7.2: A general overview of the MVC architecture^[19]. Views inherit from a general **View** class, which itself inherits from the **MVCObserver** class. A *controller* handles user interactions with the view. Both views and controllers are registered at their model, which inherits from **MVCObservable**. The model carries out the notification protocol, which informs its registered observers of data changes.

The main reasons for choosing the MVC architecture were its extensibility, flexibility and re-usability^[20]. Another important factor was its integrated notification protocol, which is used to synchronise the views and controllers during runtime. Extensions to the GUI should abide by MVC practices. The following code presents a sample extension with a new view and controller:

```

import gtk
import pygtk
import sputnik_gui

class ViewExtension(sputnik_gui.View):

    def __init__(self):
        ## call constructor of parent class
        sputnik_gui.View.__init__(self)

    def __init__(self, model = None, controller = None):
        ## call constructor of parent class
        sputnik_gui.View.__init__(self, model, controller)

    ## common methods which are used for displaying and notification purposes
    def show(self):
        pass
    def update_component(self, key):
        pass
    def update_output(self):
        pass
    def undo(self):
        pass
    def update(self):
        pass
    def reset(self):
        pass

class ControllerExtension(sputnik_gui.Controller):

    def __init__(self):
        ## call constructor of parent class
        sputnik_gui.Controller.__init__(self)

    def __init__(self, model = None, view = None):
        ## call constructor of parent class
        sputnik_gui.Controller.__init__(self, model, view)

    ## common methods which are used for displaying and notification purposes
    def show(self):
        pass
    def update_component(self, key):
        pass
    def update_output(self):
        pass
    def undo(self):
        pass
    def update(self):
        pass
    def reset(self):
        pass

```

7.2 Adding additional language support

It is simple to extend the range of file formats accepted by **Sputnik**. To do this, a single lexing class should be added (which could optionally be based on any of the existing classes `RLexerTxt`, `RLexerSBML` or `RLexerPNML`). The only requirement of a novel lexing class is that it returns a list of token objects, defined in the `Token` class, so that they can be correctly parsed through `RParser`. Accepted components of a token list are detailed below (Table 7.1). It is also possible to instantiate a *PetriNetData* object directly, bypassing tokenisation and parsing; this method will, however, also avoid existing error-handling and input constraints so should be used with caution.

Token.label	Token.value
'p'	<code>np.array</code> of place names
't'	<code>np.array</code> of transition names
'r'	<code>np.array</code> of rate constants
'm'	<code>np.array</code> of initial markings
'c'	<code>np.array</code> of capacities
'pre'	<code>np.matrix</code> for pre
'post'	<code>np.matrix</code> for post
'test'	<code>np.matrix</code> for test arcs
'inhib'	<code>np.matrix</code> for inhibitory arcs

Table 7.1: All possible components of a Token list. `np` refers to the `numpy` module.

Existing error checking methods can be invoked on any newly-defined lexer class by inheriting from class `RLexer` and calling the function `self.check()`, passing the completed token list as an argument. Alternatively, in-built error checking can be avoided by omitting this function call.

A new lexing class could be incorporated into the GUI frontend by editing the `controller_main.py` file. First, the new module should be imported in the file header. Then the `open_file` function should be edited to initialise the new lexer upon the opening of a file with the required extension. For example:

```

if '.yourfileextension' in f.name: 1
    lexer = newModule.newLexerClass() 2

```

In order to save to a novel format, a converter class can be written which extracts data from a *PetriNetData* object and writes to a file format of choice. This functionality can then be added to the GUI as above, but instead editing the `open_file` function.

7.3 Adding additional simulation algorithms

To add an additional simulation algorithm, simply create a new subclass of the abstract base class `Algorithm`. This new class should contain a method that runs the simulation, and a class variable, `algorithm`, whose value is the same as the name of the simulation method. The simulation method should return data in a format consistent with that of the existing algorithms, and store it in a new subclass of the abstract base class `SimData`.

References

- [1] CA Petri. Kommunikation mit automaten. *Bonn: Institut fur Instrumentelle Mathematik, Schriften des IIM*, 3, 1962.
- [2] R. David and H. Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer-Verlag Berlin, 2005.
- [3] M. A. Marsan. Stochastic petri nets: An elementary introduction. In *In Advances in Petri Nets*, pages 1–29. Springer, 1989.
- [4] D. Wilkinson. *Stochastic Modelling for Systems Biology*. Chapman & Hall / CRC Mathematical & Computational Biology, 2006.
- [5] J. W. Pinney, D. R. Westhead, and G. A. McConkey. Petri net representations in systems biology. *Biochemical Society Transactions*, 31:1513–1515, 2003.
- [6] D. Gilbert, M. Heiner, and S. Lehrack. A unifying framework for modelling and analysing biochemical pathways using petri nets. Technical report, Brandenburg University of Technology at Cottbus, 2007.
- [7] P.J.E. Goss and J. Peccoud. Quantitative modeling of stochastic systems in molecular biology by using stochastic petri nets. *Proceedings of the National Academy of Sciences of the United States of America*, 1998.
- [8] T. Toni. *Approximate Bayesian computation for parameter inference and model selection in systems biology*. PhD thesis, Imperial College London, 2010.
- [9] D. T. Gillespie. Exact stochastic simulation of coupled checmical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, December 1977.
- [10] Y. Cao, D. T. Gillespie, and L. R. Petzold. Efficient step size selection for the tau-leaping simulation method. *J Chem Phys*, 124(4):044109, Jan 2006.

- [11] A. Finney and M. Hucka. Systems biology markup language: Level 2 and beyond. *Biochem Soc Trans*, 31(Pt 6):1472–1473, Dec 2003.
- [12] Pnml.org. Pnml grammar, version 2009, 2009.
- [13] M. B. Elowitz and S. Leibler. A synthetic oscillatory network of transcriptional regulators. *Nature*, 403:335–338, 2000.
- [14] Creative Commons Attribution-ShareAlike 3.0 Unported Licence. <http://creativecommons.org/licenses/by-sa/3.0/>.
- [15] Open Source Icons. <http://openiconlibrary.sourceforge.net/>.
- [16] PyDoc Python v2.7.2 Documentation. <http://docs.python.org/library/pydoc.html>.
- [17] T. Tian and B. Burrage. Binomial leap methods for simulating stochastic chemical kinetics. *Journal of Chemical Physics*, 121:10356–10364, 2004.
- [18] T Toni, Y Ozaki, P Kirk, S Kuroda, and MPH Stumpf. Elucidating phosphorylation dynamics of the erk map kinase (in preparation).
- [19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern - Orientierte Software Architektur: Ein Pattern - System*. Addison-Wesley, 1998.
- [20] E. Gamma E and J. Vlissides R. Helm, R. Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison - Wesley, 1 edition, 1994.