
tellurium Documentation

Release 2.1.5

sys-bio

Jan 24, 2019

Contents

1	Installation and Front-ends	3
1.1	Installation Options	3
2	Quick Start	7
2.1	Simple Example	7
2.2	More Complex Example	8
3	Walkthrough	11
3.1	Notebook Walkthrough	11
3.2	Advanced Topics	21
3.3	Notebook Troubleshooting	24
3.4	Further Reading	24
3.5	IDE Walkthrough	27
3.6	Advanced Topics on Tellurium Spyder	31
3.7	Tellurium Spyder Troubleshooting	31
3.8	Further Reading	31
3.9	Additional Resources for Tellurium	32
3.10	Learning Python	32
4	Usage Examples	33
4.1	Basics	33
4.2	Models & Model Building	39
4.3	Simulation and Analysis Methods	44
4.4	SED-ML	47
4.5	COMBINE & Inline OMEX	54
4.6	Modeling Case Studies	71
4.7	Miscellaneous	88
5	Antimony Reference	89
5.1	Background	89
5.2	Change Log	90
5.3	Contents	90
5.4	Introduction & Basics	91
5.5	Examples	92
5.6	Simulating Models	98
5.7	Language Reference	102

6	Parameter Scan	123
6.1	Loading a Model	123
6.2	Class Methods	124
6.3	Example	125
6.4	Properties	129
6.5	SteadyStateScan	130
7	Tellurium Methods	133
7.1	Installing Packages	133
7.2	Utility Methods	134
7.3	Model Loading	140
7.4	Interconversion Utilities	142
7.5	Export Utilities	144
7.6	Stochastic Simulation	154
7.7	Math	158
7.8	ODE Extraction Methods	159
7.9	Plotting	159
7.10	Model Reset	176
7.11	jarnac Short-cuts	179
7.12	Test Models	180
7.13	Running external tools	182
7.14	Model Methods	183
8	API	187
9	Appendix	193
9.1	Source Code Repositories	193
9.2	License	193
9.3	Contact	194
9.4	Funding	194
9.5	Acknowledgments	194
10	Indices and tables	195

Model, simulate, and analyse biochemical systems using a single tool. To view the documentation on libRoadRunner, please go [here](#).

Contents:

Installation and Front-ends




1.1 Installation Options

Tellurium has several front-ends, and can also be installed as a collection of pip packages. We recommend a front-end for end-users who wish to use Tellurium for biological modeling, and the pip packages for developers of other software which uses or incorporates Tellurium.

1.1.1 Front-end 1: Tellurium Notebook

Tellurium's notebook front-end mixes code and narrative in a flowing, visual style. The Tellurium notebook will be familiar to users of Jupyter, Mathematica, and SAGE. However, unlike Jupyter, Tellurium notebook comes pre-packaged as an app for Windows, Mac, and Linux and does not require any command line installation. This front-end is based on the [interact project](#).

- Front-end: **Tellurium Notebook**

- Supported platforms:   
- Python version: 3.6, 64-bit
- **Download:** [here](#)

1.1.2 Front-end 2: Tellurium IDE

User who are more familiar with MATLAB may prefer Tellurium's IDE interface, which is based on popular programming tools (Visual Studio, etc.). This front-end is based on the [Spyder project](#). Due to stability issues, we recommend Mac users use the Tellurium notebook front-end instead.

- Front-end: **Tellurium IDE**
- Supported platforms:   (no Mac updates)

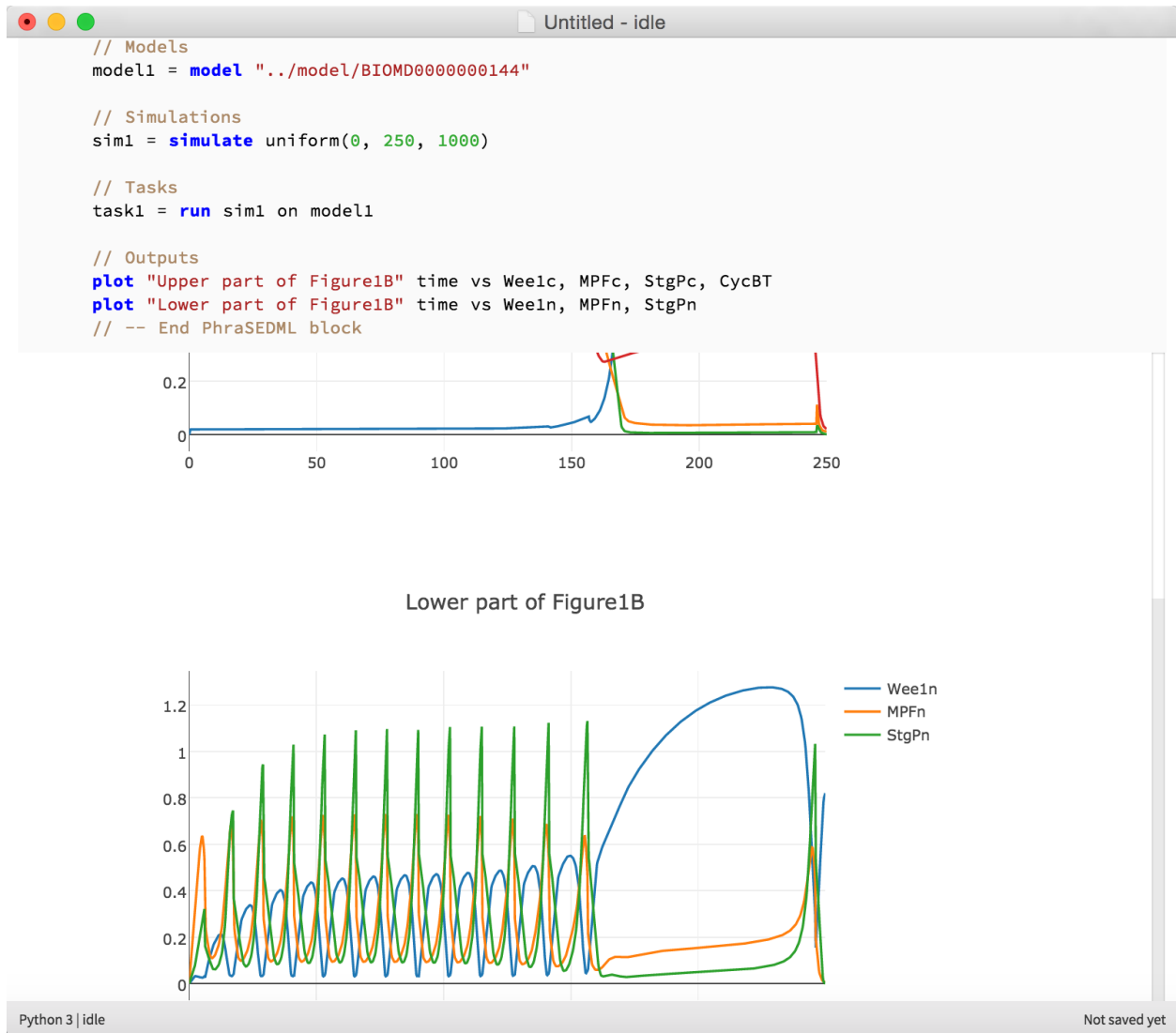


Fig. 1: Tellurium notebook offers an environment similar to Jupyter

- Python version: 2.7, 64-bit
- **Download:** [here](#)

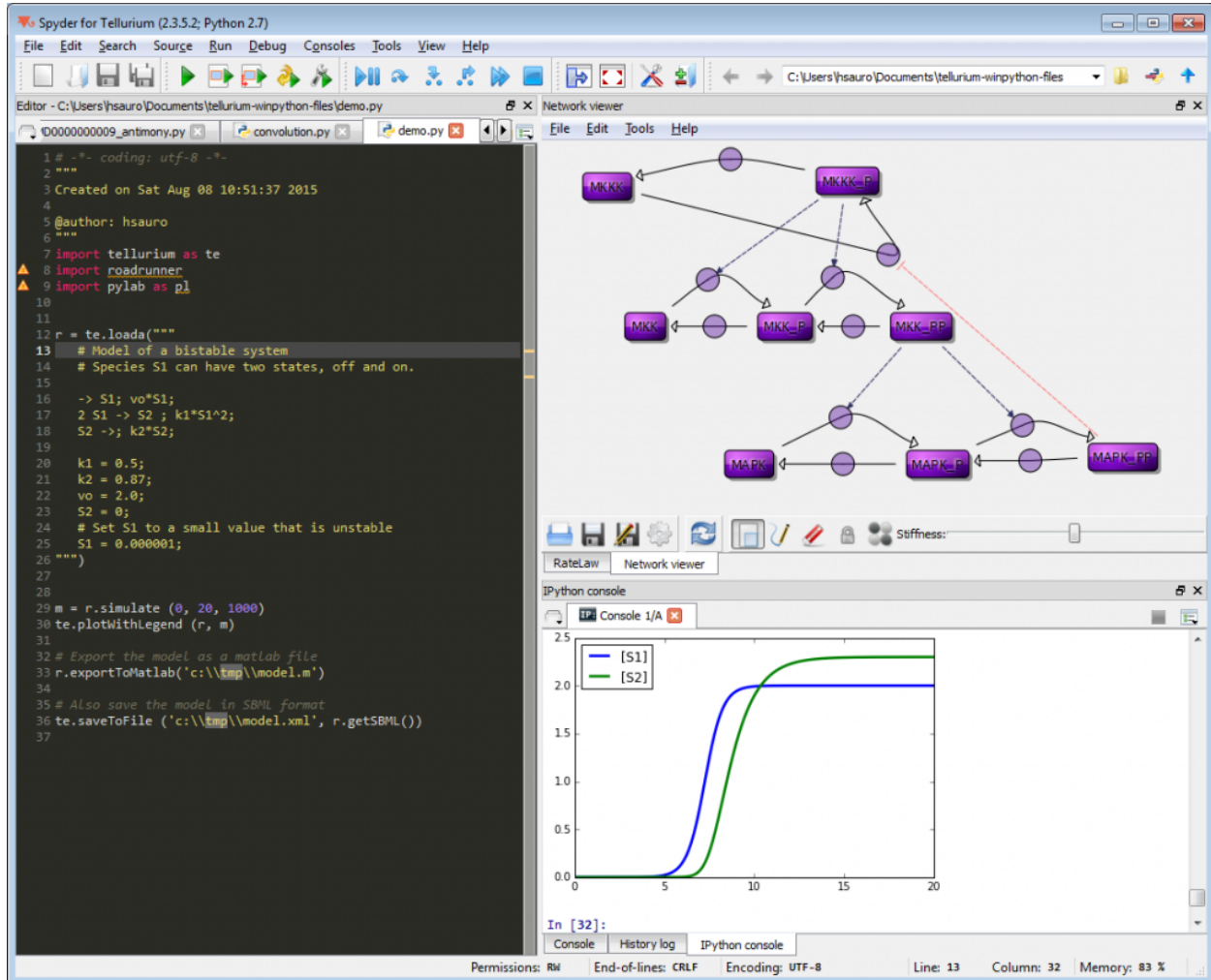


Fig. 2: Tellurium IDE features a programmer-centric interface similar to MATLAB

1.1.3 PyPI Packages

Tellurium can be installed using the command line tool `pip`.

- No front-end

- Supported platforms: 
- Python version: 2.7, 3.4, 3.5, 3.6

```
$ pip install tellurium
```

1.1.4 Supported Python Versions

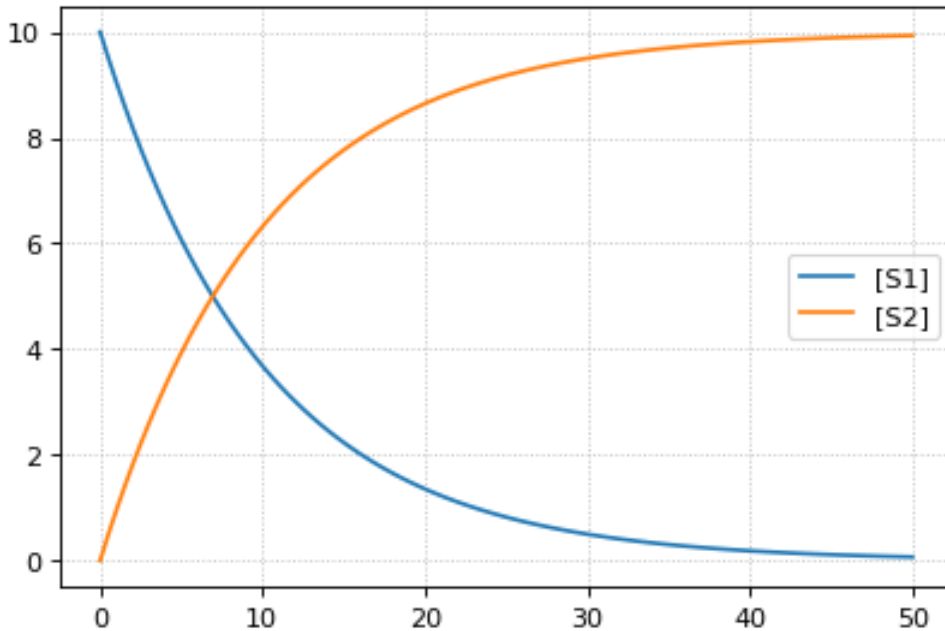
The Tellurium PyPI packages support 64-bit Python versions 2.7, 3.4, 3.5, and 3.6 for Windows, Mac, and Linux. The notebook viewer comes with Python 3.6 (64-bit) and the IDE comes with Python 2.7 (32-bit). If you need support for a Python version not already covered, please [file an issue](#).

To get started using Tellurium, download one of the [front-ends](#). Then, follow the examples below to get an idea of how to load and simulate models.

2.1 Simple Example

This shows how to set up a simple model in Tellurium and solve it as an ODE. Tellurium uses a human-readable representation of SBML models called Antimony. The Antimony code for this example contains a single reaction with associated kinetics. After creating the Antimony string, use the `loada` function to load it into the [RoadRunner](#) simulator. A [RoadRunner](#) instance has a method `simulate` that can be used to run timecourse simulations of a model, as shown below.

```
import tellurium as te
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
r.simulate(0, 50, 100)
r.plot()
```



2.2 More Complex Example

Tellurium can also handle stochastic models. This example shows how to select Tellurium's stochastic solver. The underlying simulation engine used by Tellurium implements a Gibson direct method for simulating this model.

```
import tellurium as te
import numpy as np

r = te.loada('''
    J1: S1 -> S2; k1*S1;
    J2: S2 -> S3; k2*S2 - k3*S3
    # J2_1: S2 -> S3; k2*S2
    # J2_2: S3 -> S2; k3*S3;
    J3: S3 -> S4; k4*S3;

    k1 = 0.1; k2 = 0.5; k3 = 0.5; k4 = 0.5;
    S1 = 100;
''')

# use a stochastic solver
r.integrator = 'gillespie'
r.integrator.seed = 1234
# selections specifies the output variables in a simulation
selections = ['time'] + r.getBoundarySpeciesIds() + r.getFloatingSpeciesIds()
r.integrator.variable_step_size = False

# run repeated simulation
Ncol = len(r.selections)
Nsim = 30
points = 101
s_sum = np.zeros(shape=[points, Ncol])
for k in range(Nsim):
```

(continues on next page)

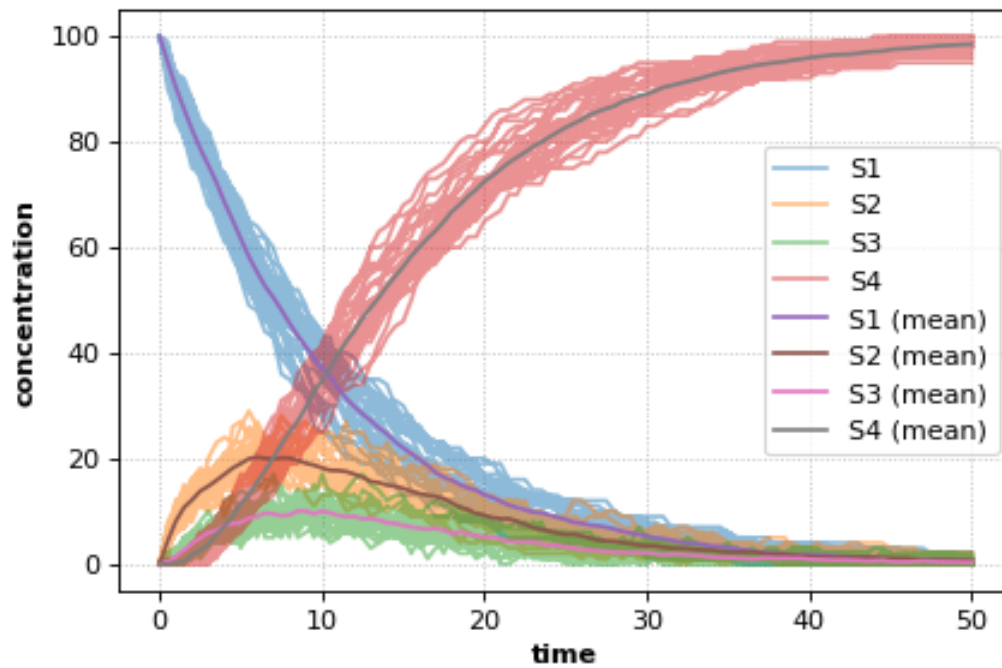
(continued from previous page)

```

r.resetToOrigin()
s = r.simulate(0, 50, points, selections=selections)
s_sum += s
# use show=False to add traces to the current plot
# instead of starting a new one, equivalent to MATLAB hold on
r.plot(s, alpha=0.5, show=False)

# add mean curve, legend, show everything and set labels, titels, ...
fig = te.plot(s[:,0], s_sum[:,1:]/Nsim, names=[x + ' (mean)' for x in selections[1:]],
→ title="Stochastic simulation", xtitle="time", ytitle="concentration")

```



3.1 Notebook Walkthrough

If you have not already done so, download and install the [Tellurium notebook front-end](#) for your platform (Windows, Mac, and Linux supported).

3.1.1 Basics

The notebook environment allows you to mix Python code, narrative, and exchangeable standards for models and simulations. When you first open the notebook, you will have a single Python cell. You can type Python code into this cell. To run the code:

- Press `shift+Enter`,
- Click the play button at the upper right of the cell, or
- Choose `Cell -> Run All` from the menu.

The output of running the cell will be shown beneath the cell.

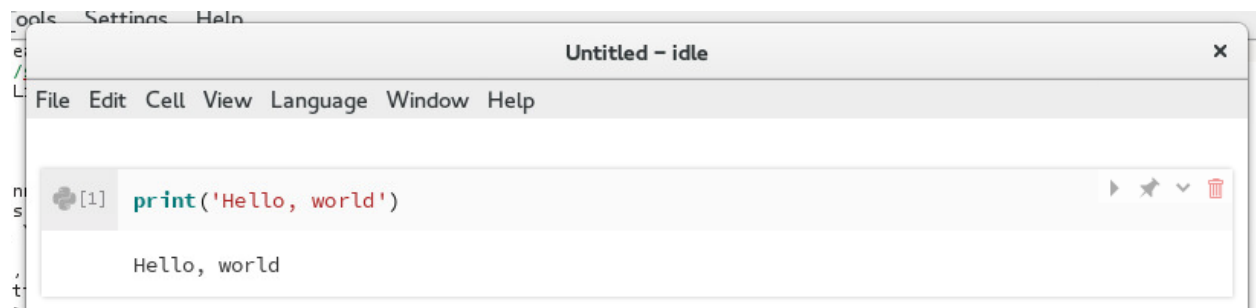


Fig. 1: Output of running a Python cell

3.1.2 Creating Cells

You can add new cells by moving your cursor past the last cell in the notebook. You should see a menu with three options: New, Import, and Merge. Choose *New* to create a new cell. You will see four choices:

- New Python Cell
- New Markdown Cell (for creating narrative)
- New Model Cell (for SBML/Antimony models)
- New OMEX Cell (for COMBINE archives)

For now, select *New Markdown Cell*. Markdown cells allow you to place formatted text and hyperlinks in your notebook. For a review of Markdown syntax, please see [this reference](#).

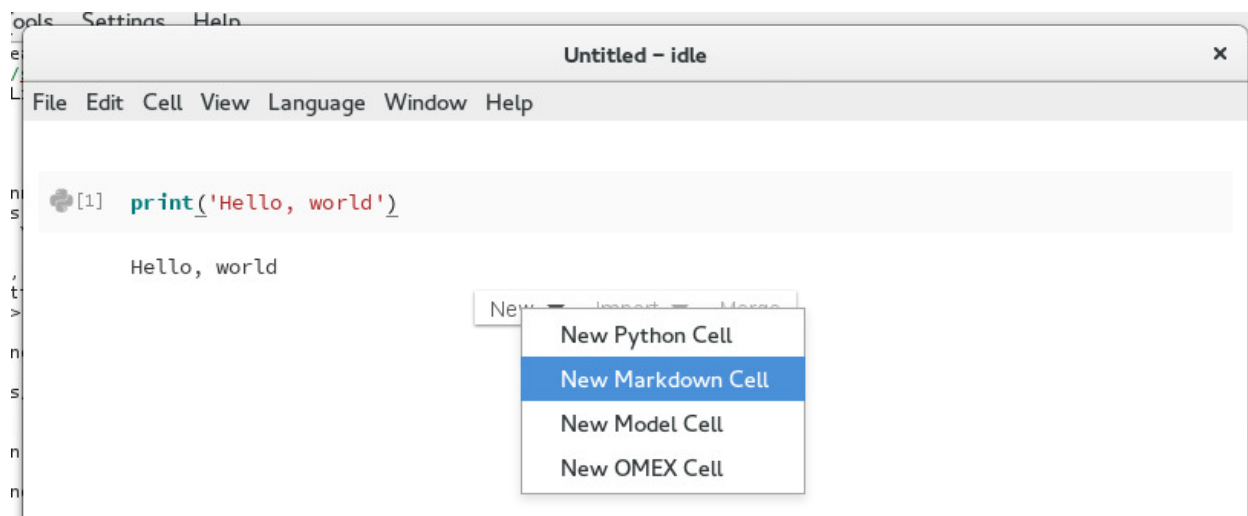


Fig. 2: Creating a new Markdown cell

3.1.3 SBML Cells

Unlike vanilla Jupyter, Tellurium allows you embed a human-readable representation of SBML directly in your notebook. To begin, first download the [SBML for the repressilator circuit from BioModels](#). Then, in the Tellurium notebook viewer, move your cursor past the last cell in the notebook and select *Import* -> *Import SBML...*

Navigate to the `BIOMD0000000012.xml` file that you downloaded and select it. A new cell will be created in the notebook with a human-readable representation of this model. The human-readable syntax is called Antimony, and you can find an [extensive reference on the syntax here](#). For now, just change the name of the model from `BIOMD0000000012` to `repressilator`.

Now, run the cell. You should see confirmation that the model was correctly loaded and is available under the variable `repressilator` in Python.

After the SBML cell, create a Python cell with the following content:

```
repressilator.reset() # in case you run the cell again
repressilator.simulate(0,1000,1000) # simulate from time 0 to 1000 with 1000 points
repressilator.plot() # plot the simulation
```

After you run this cell, you should see the following simulation plot:

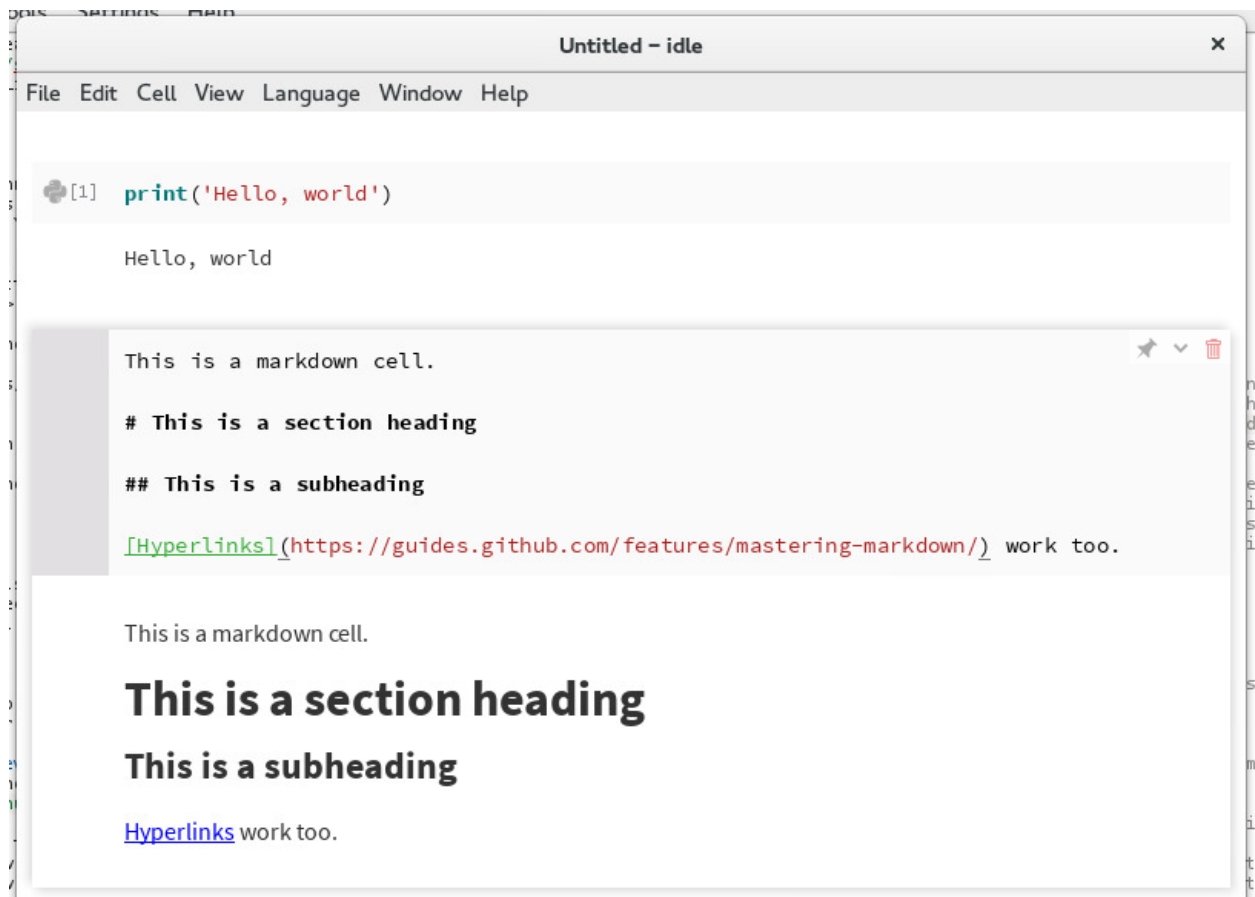


Fig. 3: Editing a Markdown cell

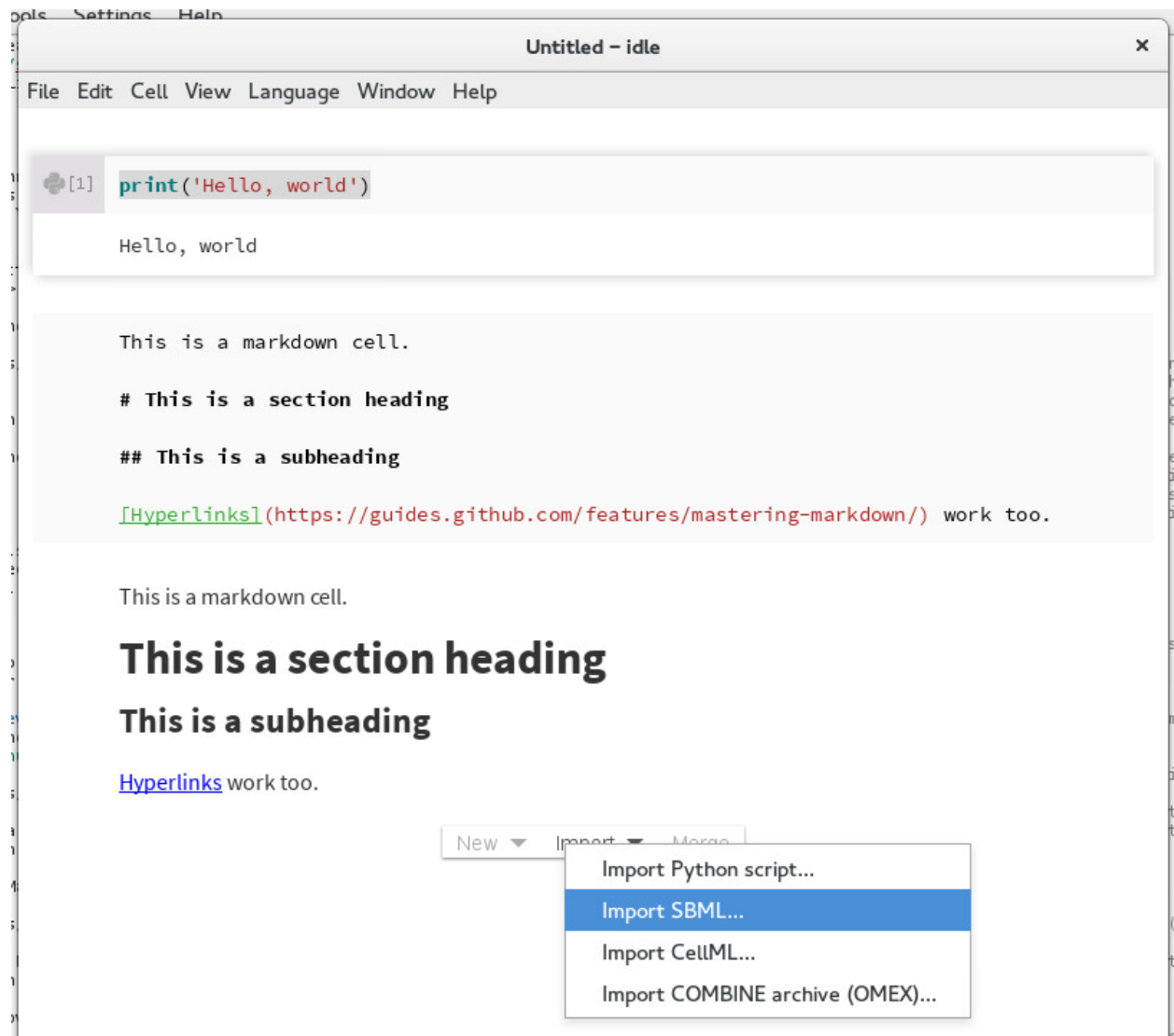


Fig. 4: Importing an SBML model



Fig. 5: Changing the name of the model

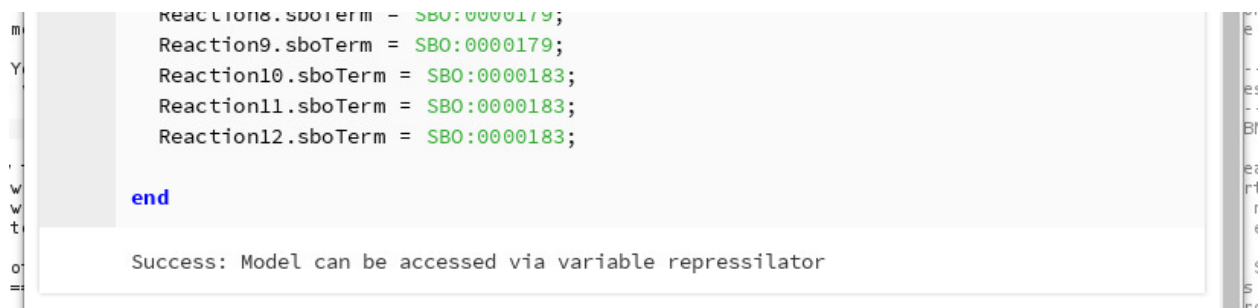


Fig. 6: Running the SBML cell

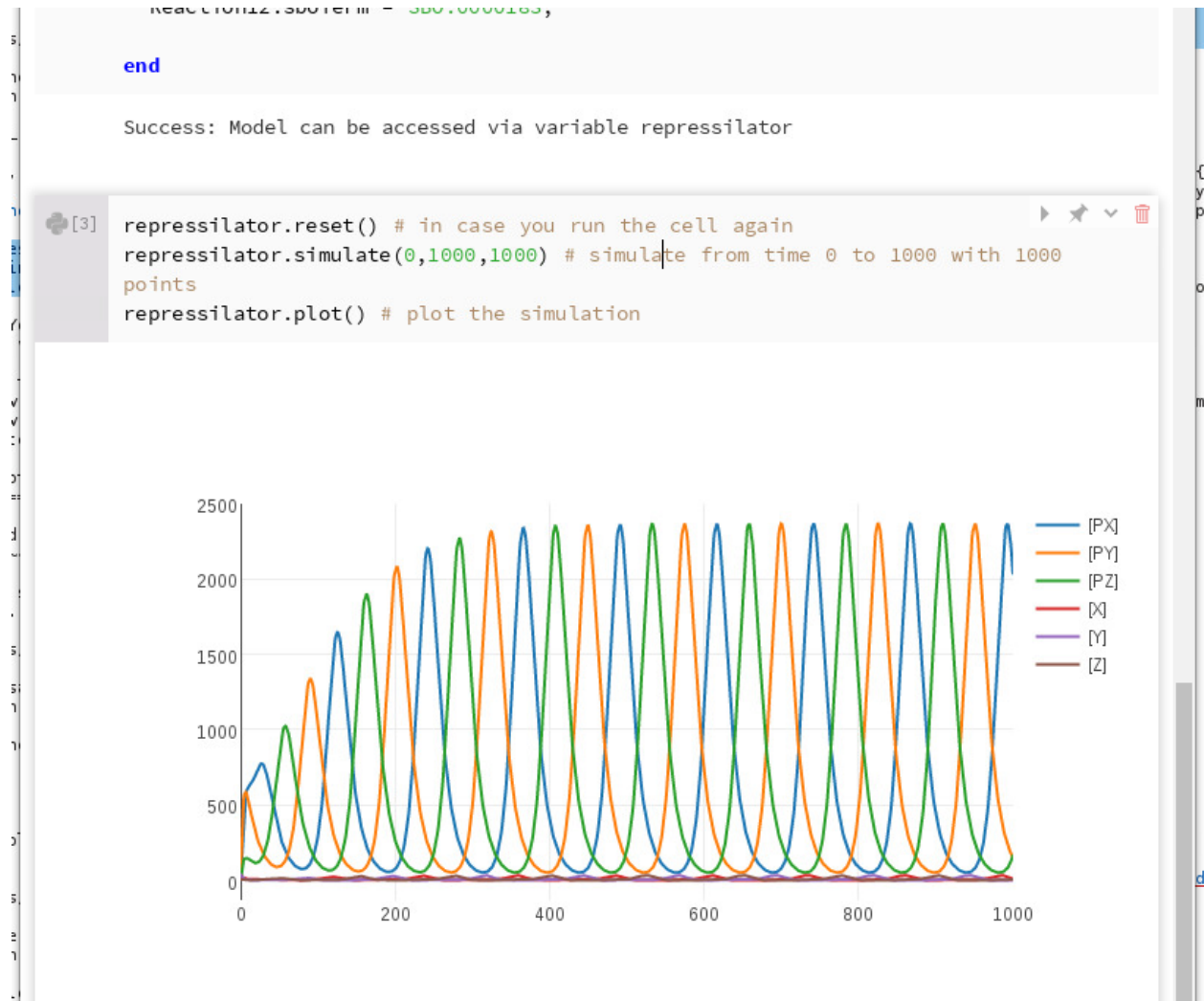


Fig. 7: Simulating the SBML model

The `repressilator` variable is actually an instance of the [RoadRunner](#) simulator. Please see the [official documentation for libRoadRunner](#) for an extensive list of methods and options that can be used with RoadRunner.

You can also use `ctrl+Space` to open the auto-completion menu for a variable defined in a previous cell. This also goes for variables such as `repressilator` defined in SBML cells.

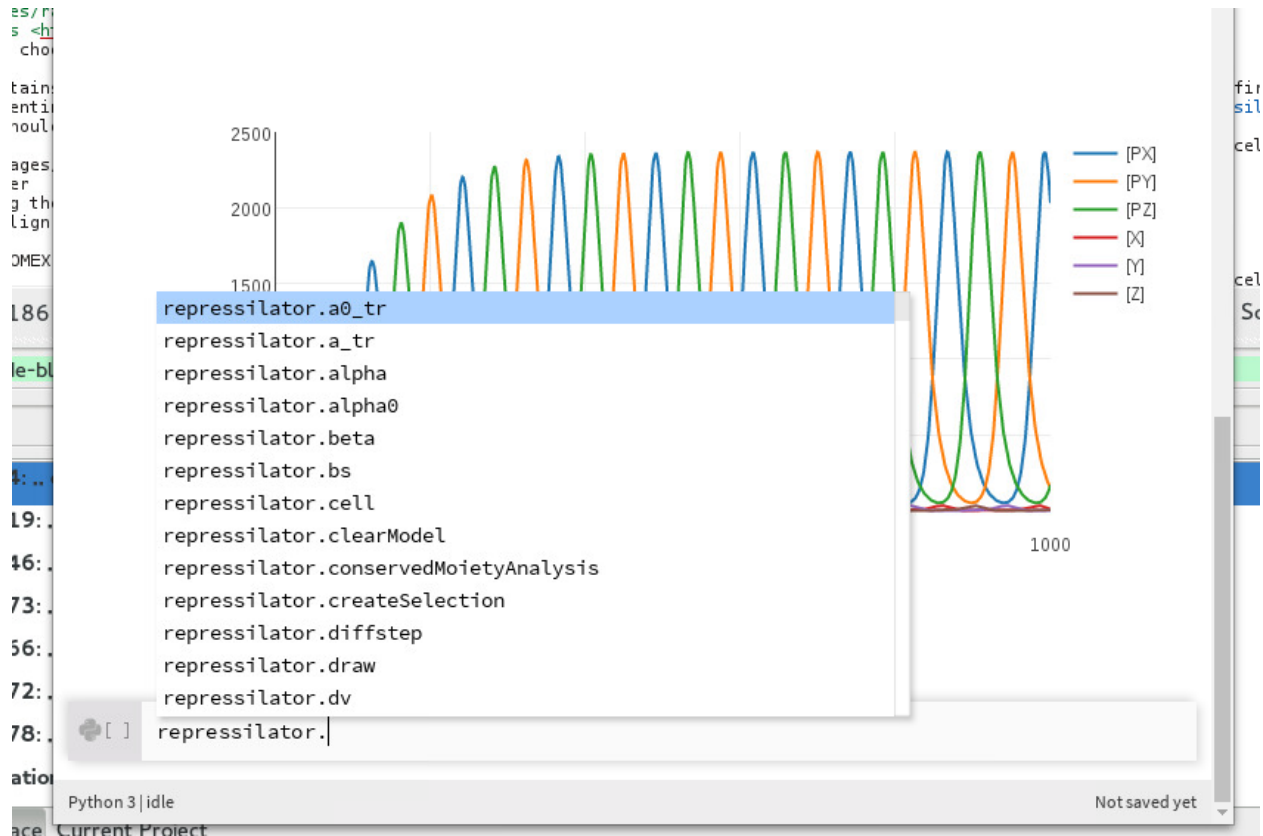


Fig. 8: Showing all auto-completions for the `repressilator` RoadRunner instance

3.1.4 COMBINE Archive Cells

Another name for COMBINE archives is the Open Modeling and EXchange format (OMEX), which shows up in various Tellurium menus and functions. COMBINE archives are containers for various community standards in systems biology. They can contain [SBML](#), [SED-ML](#), [CellML](#), and [NeuroML](#). Tellurium supports importing COMBINE archives containing SBML and SED-ML.

To begin, download [this COMBINE archive](#) example (originally from the [SED-ML Web Tools](#)). In the Tellurium notebook viewer, move the mouse past the last cell until the Cell Creator Bar appears and choose `Import -> Import COMBINE archive (OMEX) ...`

This archive contains an SBML model and a SED-ML simulation. The simulation has a forcing function (representing external input to the system) in the form of a pulse. After running this cell, you should see the following output:

As a demo of Tellurium's COMBINE archive editing functionality, we can change the duration of the pulse. Change the following line:

```
task1 = repeat task0 for local.index in uniform(0, 10, 100), local.current = index ->
    piecewise(8, index < 1, 0.1, (index >= 4) && (index < 6), 8), model1.J0_v0 =
    current : current
```

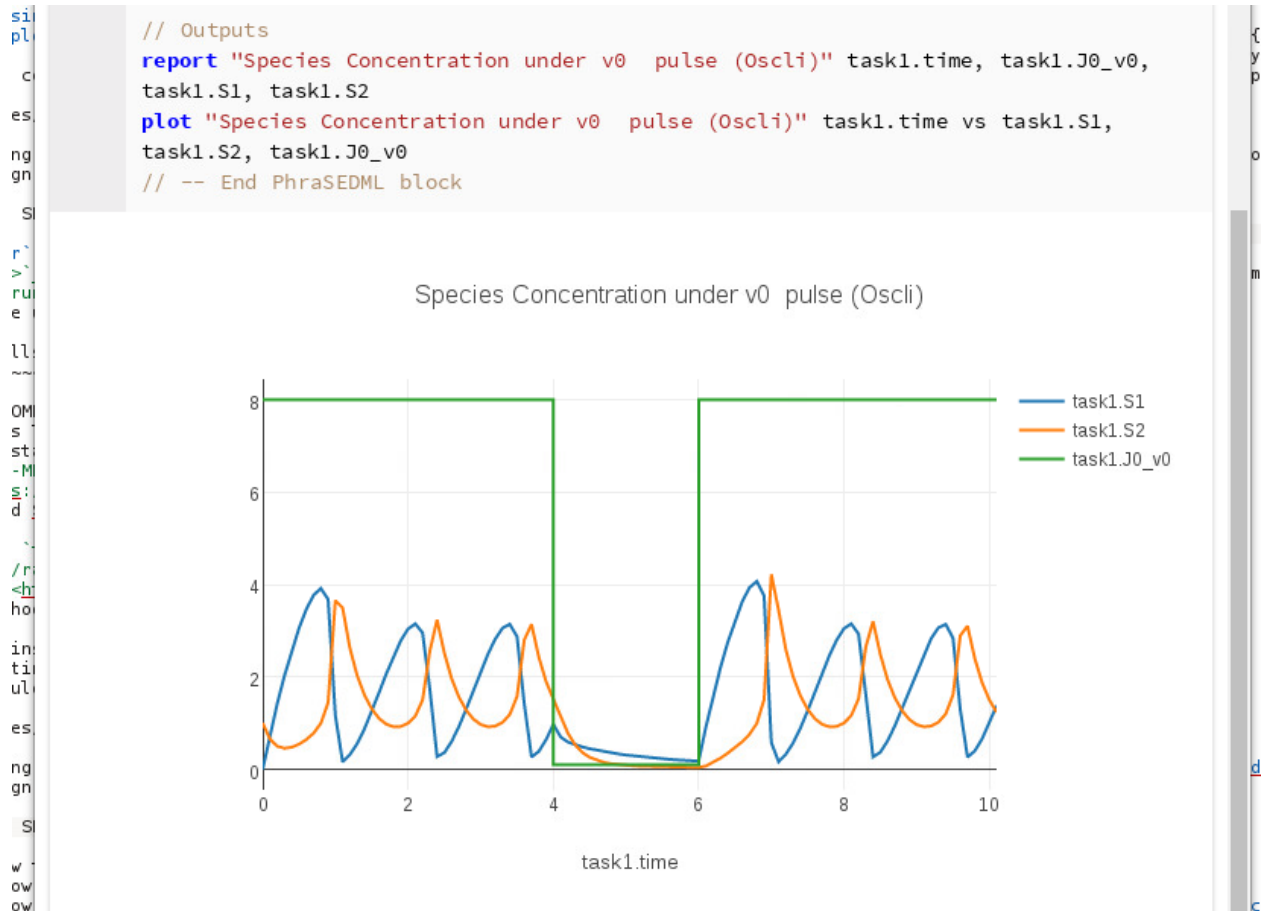


Fig. 9: Running the OMEX cell

To:

```
task1 = repeat task0 for local.index in uniform(0, 10, 100), local.current = index ->
  ↳ piecewise(8, index < 1, 0.1, (index >= 4) && (index < 10), 8), model1.J0_v0 =
  ↳ current : current
```

In other words, `index < 6` was changed to `index < 10`. Run the cell:

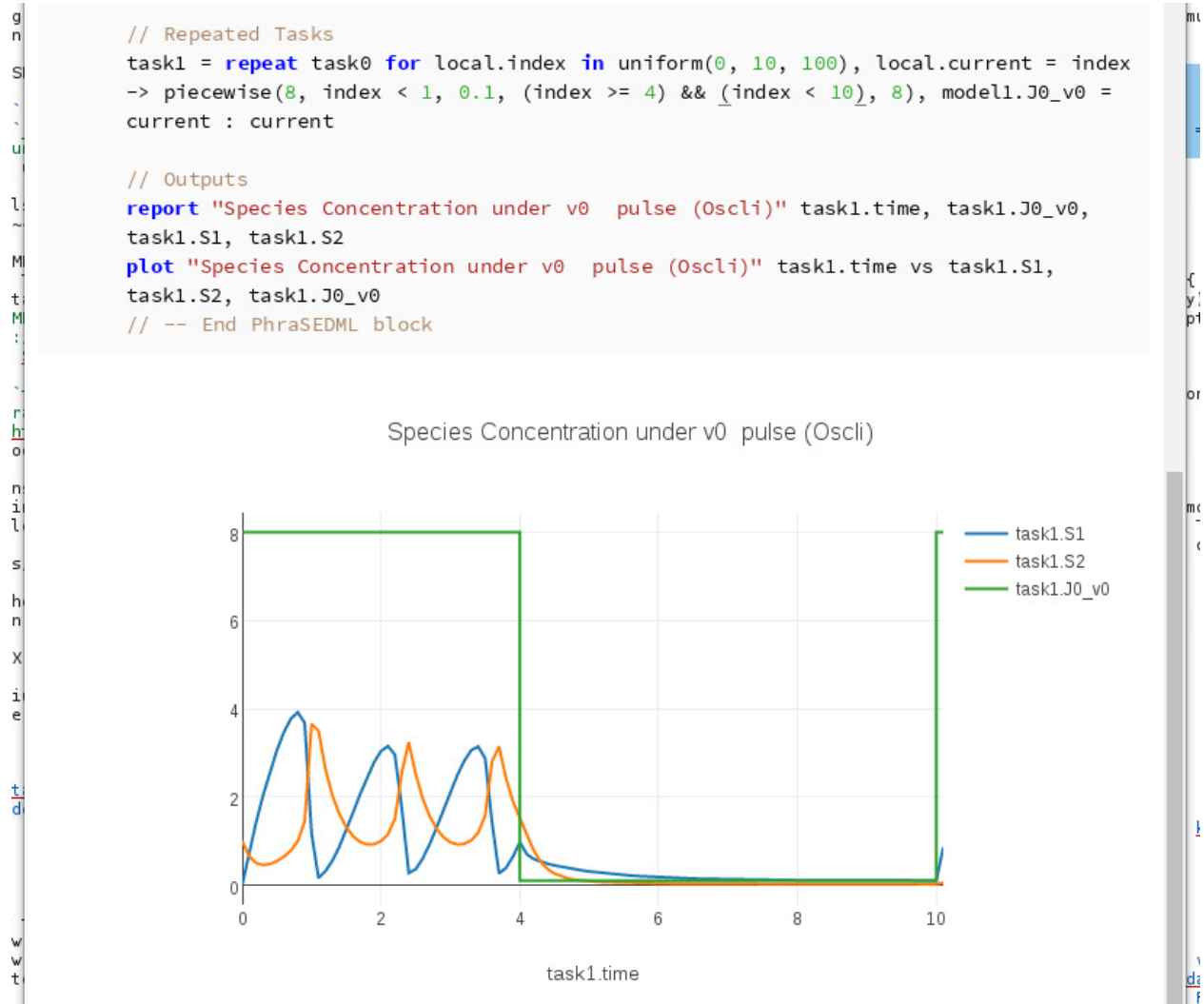


Fig. 10: Editing the OMEX cell

You can re-export this cell to a COMBINE archive by clicking the diskette icon in the upper right:

3.1.5 Find/Replace in Notebook Cells

To search for text in a notebook cell, use `ctrl+F`. To search for whole-words only, use `/\bmyword\b` where `myword` is the word you want to search for.

To search and replace, use `ctrl+shift+R`. For example, to replace `myvar` but not `myvar2` (i.e. whole-word search & replace) in the code below, press `ctrl+shift+R`, enter `/\bmyvar\b/` for the search field and `newvar` for the replace field. The result is that all instances of `myvar` are replaced, but not `myvar2`:

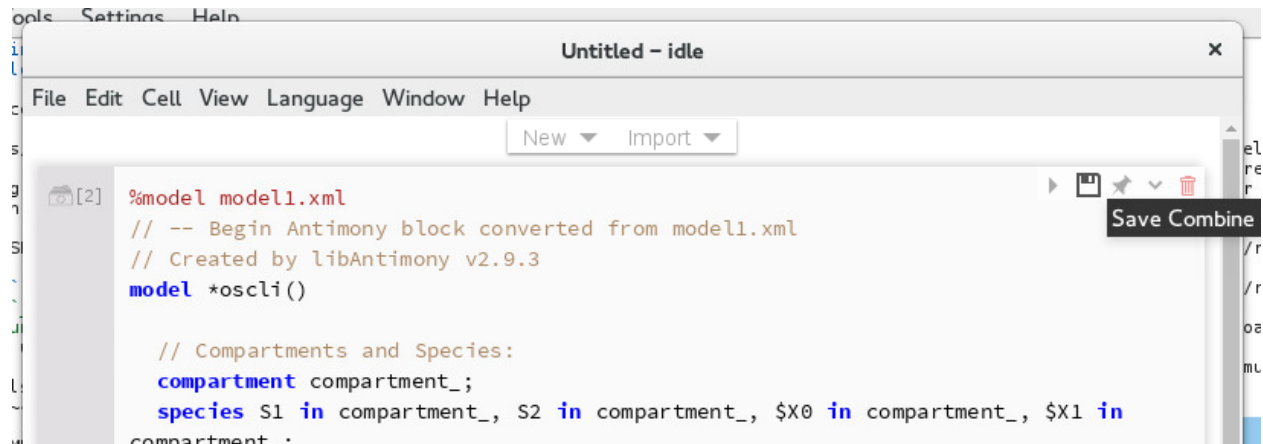


Fig. 11: Exporting the COMBINE archive

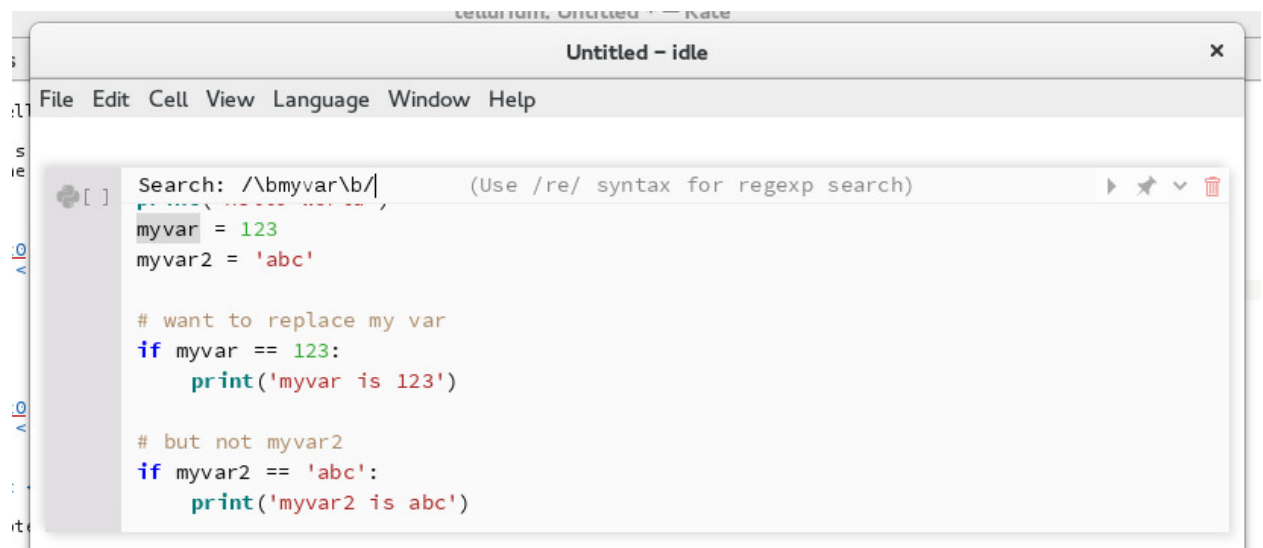


Fig. 12: Searching for whole words

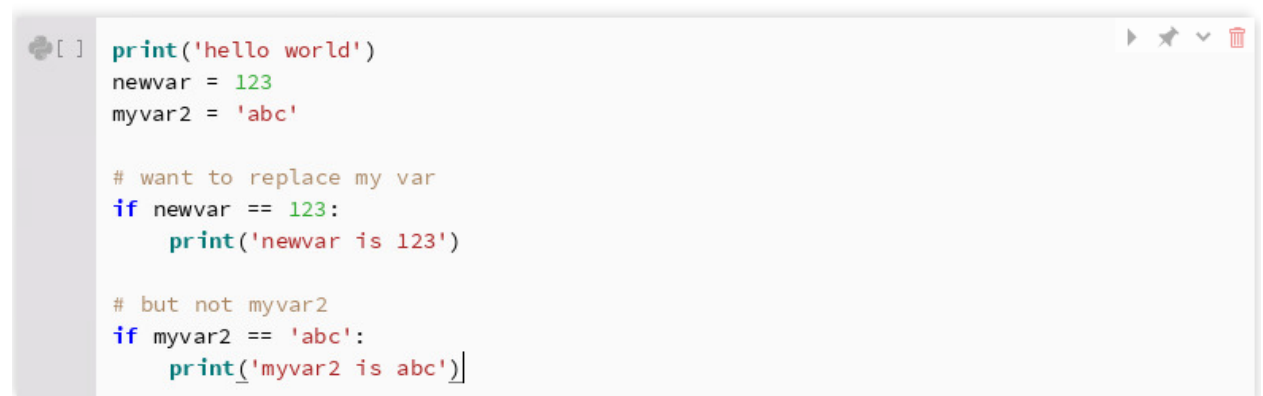


Fig. 13: Search & replace demo with whole words

3.1.6 Example Notebooks

Tellurium comes with many example notebooks showing how to use its various features. To access these notebooks, use the `File -> Open Example Notebook` menu. Tellurium comes with five example notebooks:

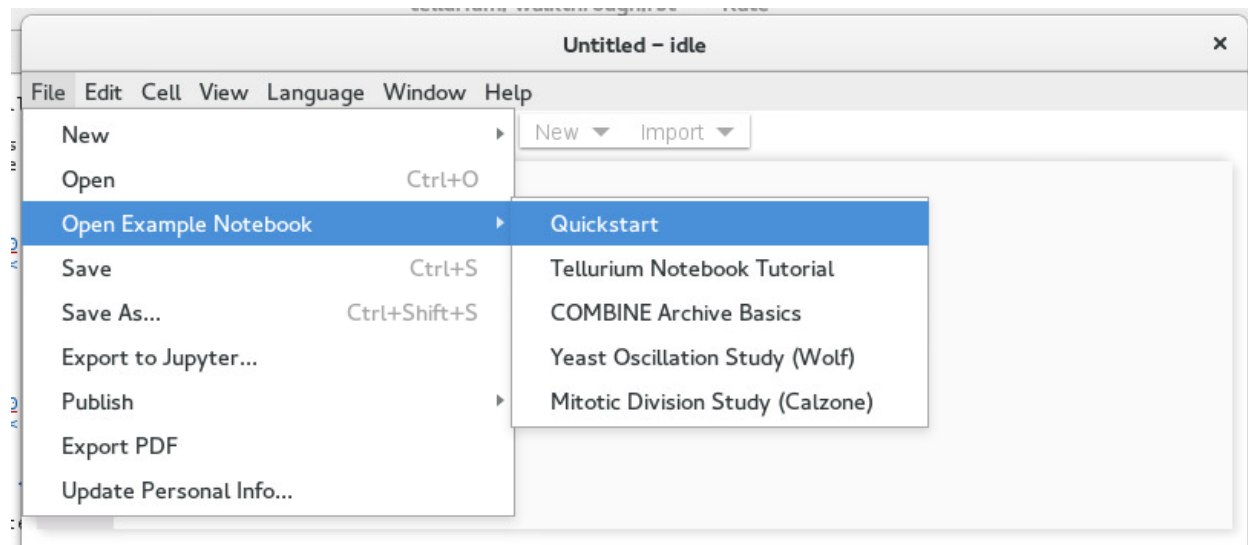


Fig. 14: Opening example notebooks

The **Quickstart** notebook contains the [Quickstart](#) example from this documentation, using the SBML cells of the Tellurium notebook viewer.

3.1.7 Exporting to Jupyter

Tellurium notebooks can contain special cell types such as the SBML or OMEX cells described above. These notebooks cannot be properly read by Jupyter. However, you can export these notebooks to Jupyter by choosing `File -> Export to Jupyter...` from the menu. You will notice that the exported notebooks contain special cell magics such as `%%crn` and `%%omex`. To run these notebooks in Jupyter, install the `temagics` package in addition to `tellurium` using `pip`.

3.2 Advanced Topics

3.2.1 Using Other Jupyter Kernels / Languages

A built-in Python 3 kernel is provided with the notebook app. However, there are cases where this is not enough. Tellurium owes its existence in part to great free / open-source projects like [nteract](#). We recommend anyone interest in a general-purpose notebook environment [consider nteract instead](#).

Nevertheless, sometimes using a kernel other than the built-in Python 3 kernel is necessary. Starting with version 2.0.14, Tellurium supports automated discovery of other Jupyter kernels, such as different Python versions and distributions (e.g. Anaconda) and other languages (the Tellurium packages are not available in other languages). The following example shows how to use an R kernel with Tellurium.

- First, follow the installation instructions for the [IRkernel](#) (see also). These instructions use R 3.3.0. The following procedure for installing the IRkernel works for us:

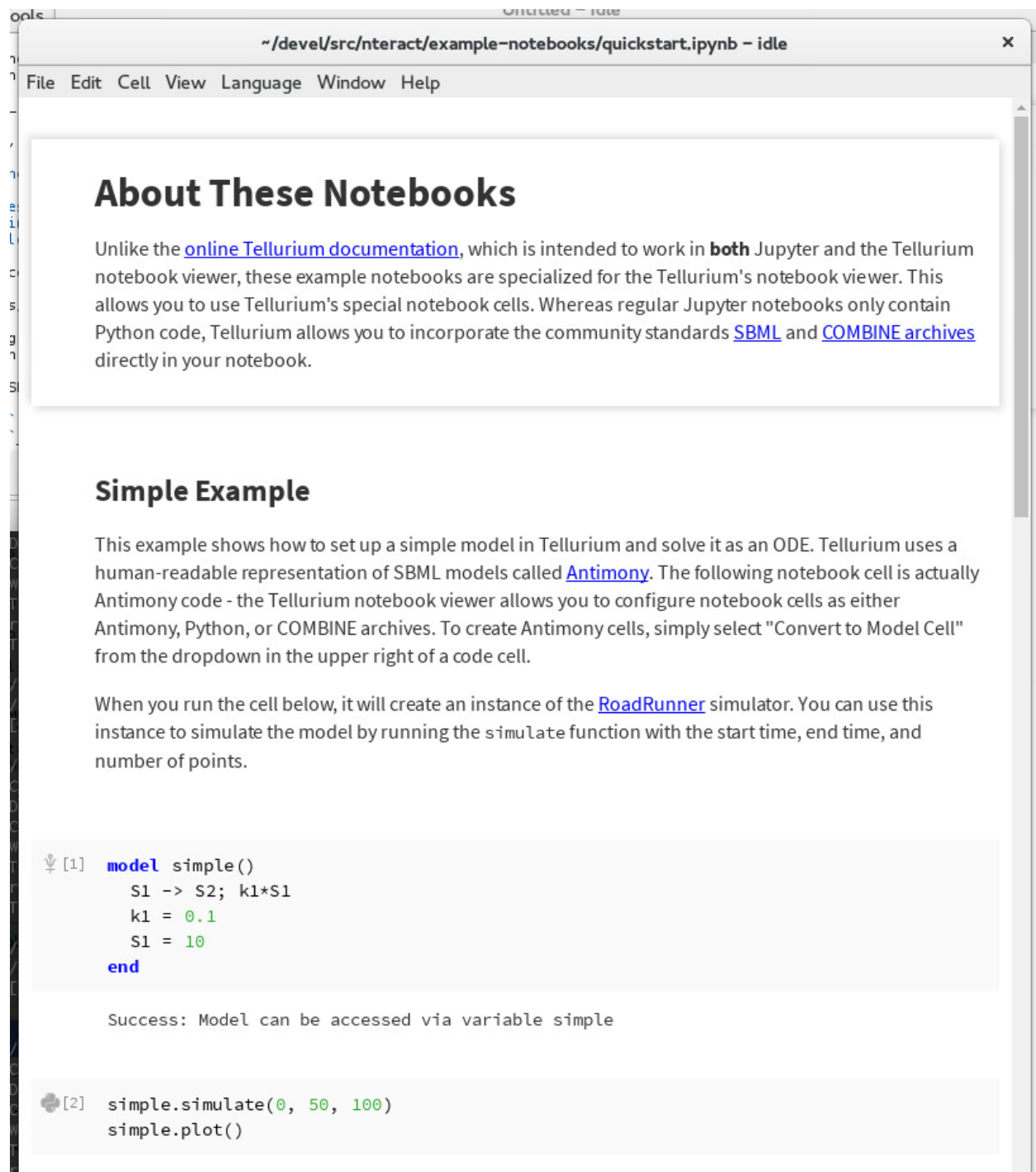


Fig. 15: Quickstart example notebook

```
install.packages('devtools')
install.packages(c('repr', 'IRdisplay', 'evaluate', 'crayon', 'pbdZMQ', 'devtools',
  ↪ 'uuid', 'digest'))
devtools::install_github('IRkernel/IRkernel')
```

- Make sure the IRkernel is registered:

```
IRkernel::installspec()
```

- Start the Tellurium notebook app. Under the Language menu, select Find Kernels... A pop-up with a Scan button should appear. Click the Scan button. The results of the scan show all the kernels available to Tellurium. The built-in Python 3 and Node.js kernels are always available. Additional kernels appear based on installed Jupyter kernels. If you don't see a Jupyter kernel you want here, make sure you have correctly installed the kernel (each has its own set of instructions). If the kernel still does not show up, make sure it is a true Jupyter kernel. Older IPython-based kernels (i.e. kernels which install under ~/.ipython instead of ~/.jupyter) cannot be discovered by Tellurium.

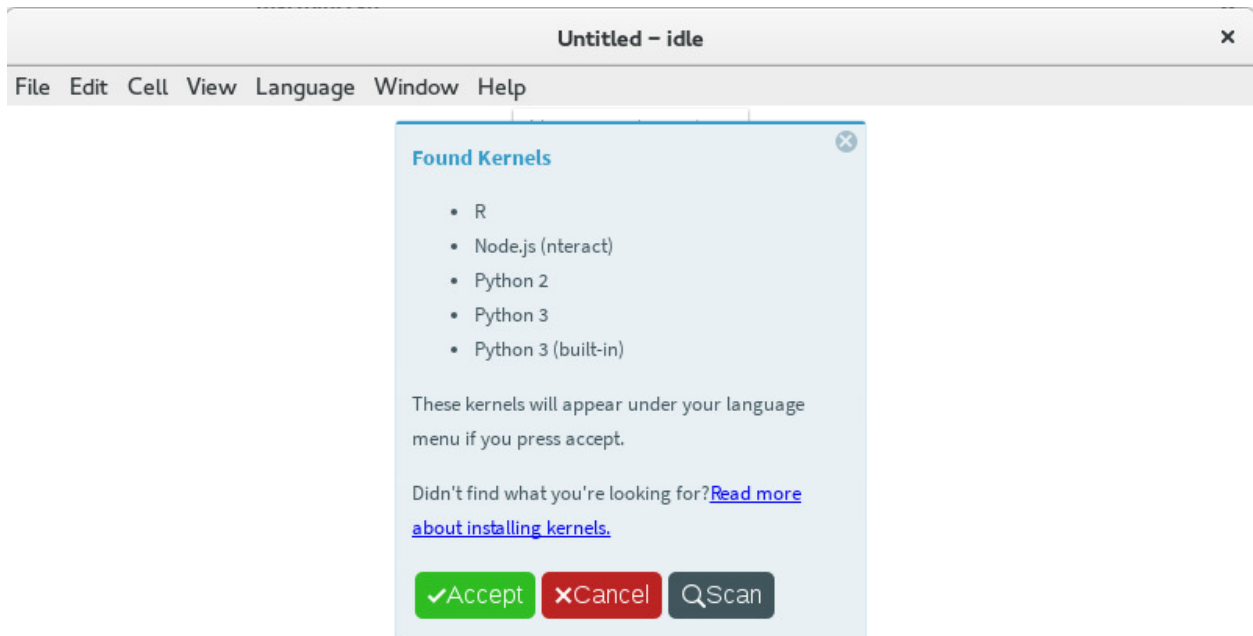


Fig. 16: Displaying the available kernels after a scan

- Sometimes the path to a kernel's executable can be displayed by hovering over the kernel's name. The R kernel you installed should appear in the list. Click the accept button to cause the Language menu to be updated with the new kernel choices.
- By selecting Language -> R, you can cause the notebook to switch to the IRkernel. All code cells will be interpreted as R (SBML and OMEX cells will not work).

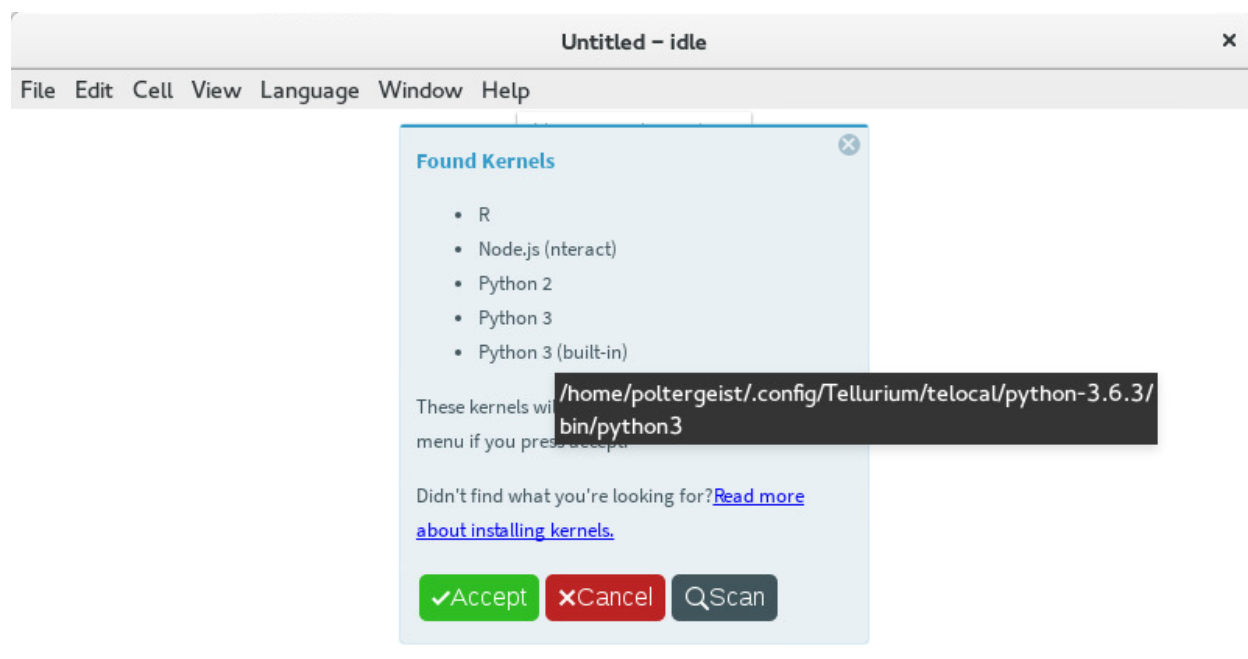


Fig. 17: Hover over the name of a kernel to display its path

3.3 Notebook Troubleshooting

3.3.1 Problem: Cannot Load Kernel

The notebook viewer ships with a Python3 kernel, which causes problems when trying to open a notebook saved (e.g. by Jupyter) with Python2.

3.3.2 Solution

In such a case, simply replace the kernel by choosing Language -> Python 3 from the menu.

3.3.3 Problem: Saving the Notebook Takes Forever

3.3.4 Solution

When highly detailed / numerous plots are present, Plotly is known to slow down notebook saving. In such cases, you can switch to matplotlib instead of Plotly by calling `import tellurium as te; te.setDefaultPlottingEngine('matplotlib')` at the beginning of your notebook. When using matplotlib for plotting, the long save times do not occur.

Alternatively, choose Language -> Restart and Clear All Cells to save the notebook without output.

3.4 Further Reading

- Tellurium notebook is based on the [nteract](#) app.

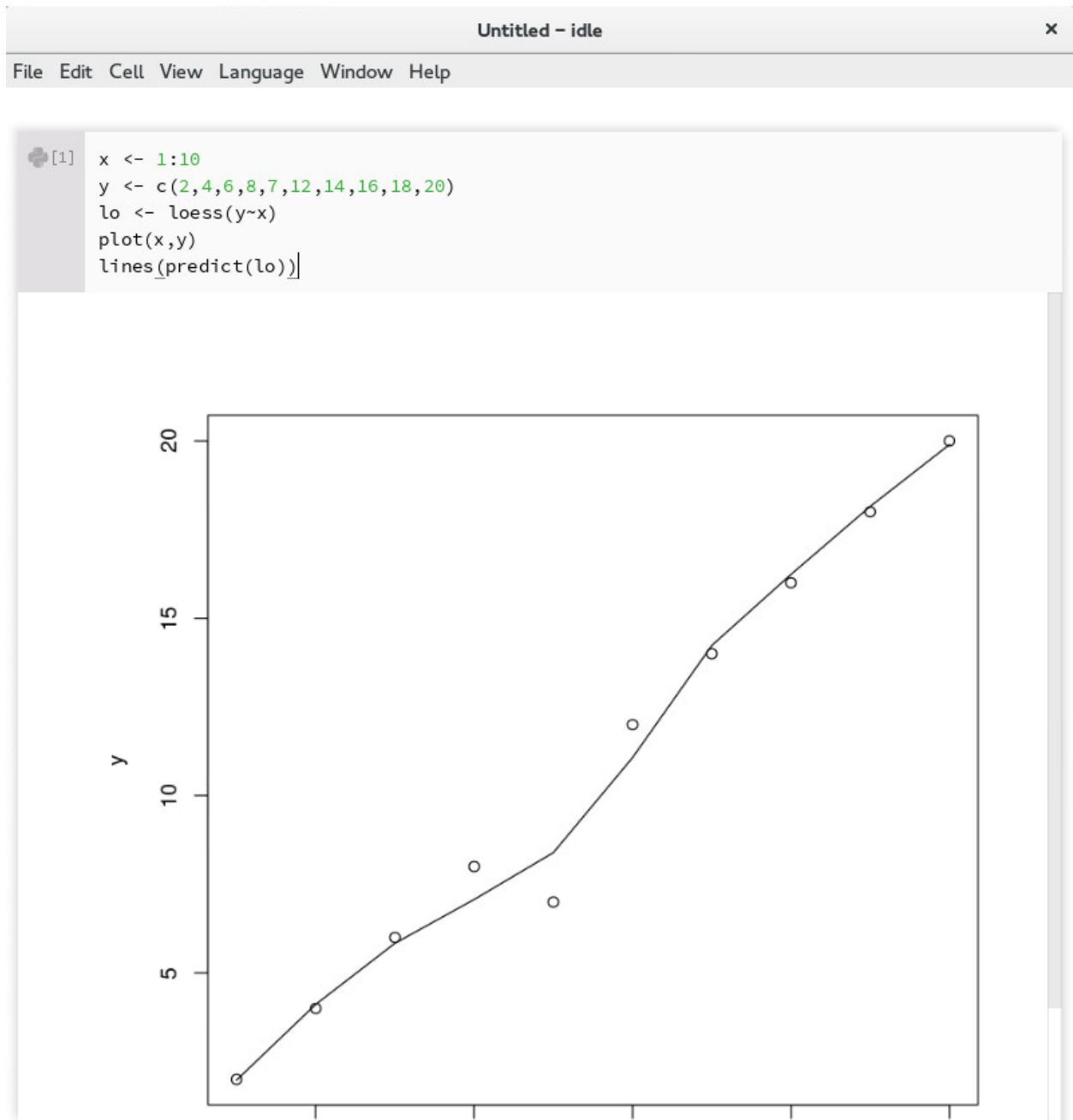


Fig. 18: Demo of running an R kernel in Tellurium

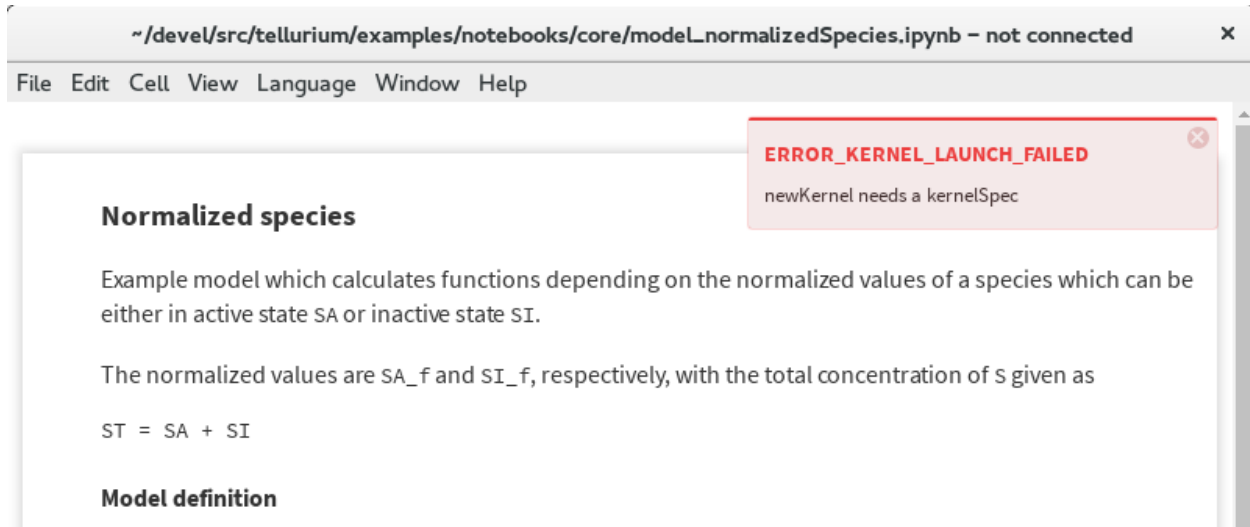


Fig. 19: Error message when kernel cannot be loaded

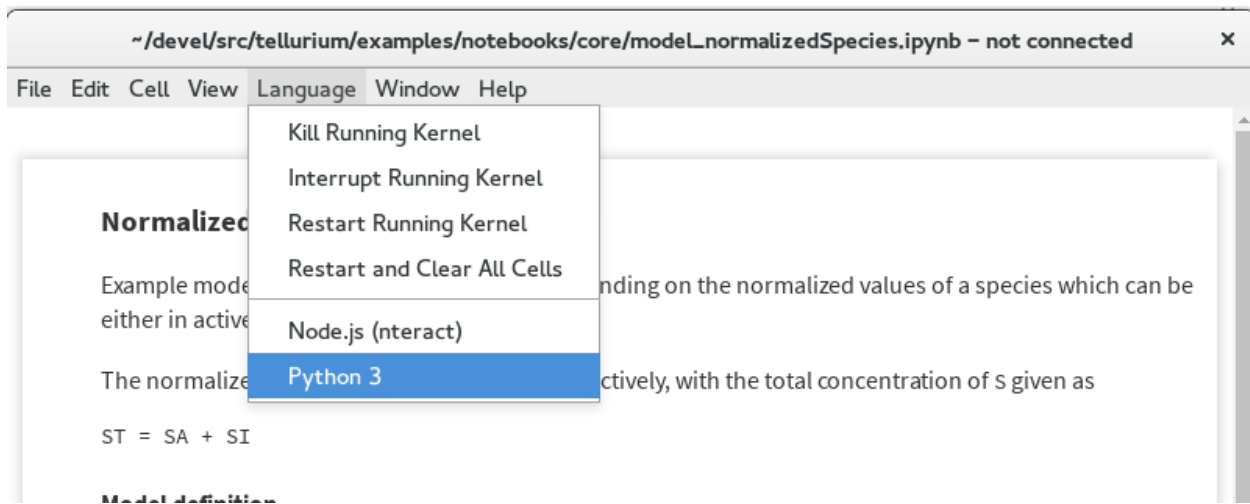


Fig. 20: Fix for kernel loading problem

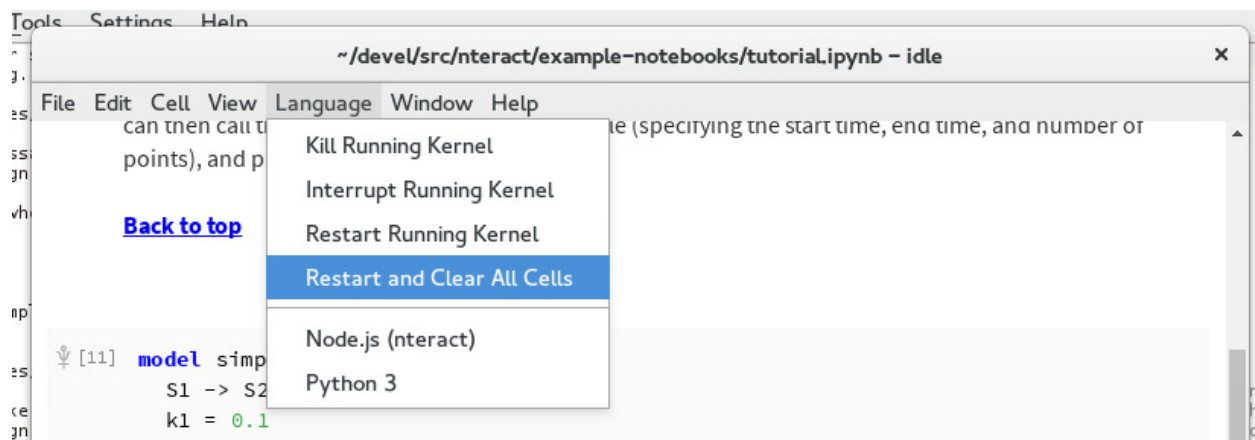


Fig. 21: Reset and clear all cells

- Jupyter.

3.5 IDE Walkthrough

If you have not already done so, download and install the [Tellurium Spyder IDE front-end](#) for your platform (only for Windows, legacy versions supported Mac).

3.5.1 Basics

Tellurium Spyder is based on Spyder IDE, a popular open-source integrated development environment for Python. Tellurium Spyder offers experience akin to MATLAB, allowing you to view, edit, and execute Python scripts through dedicated editor and console windows. Additionally, Tellurium Spyder comes with various tools to help you code. When you first open Tellurium Spyder, you will be greeted by an editor pane with example script, an IPython console, and a Help pane. You can execute the script in the editor pane directly on the IPython console by:

- Pressing the green arrow
- Pressing F5
- Pressing Run -> Run

The example script contains an oscillation model. When you run the script, a plot will appear in the IPython console as the output.

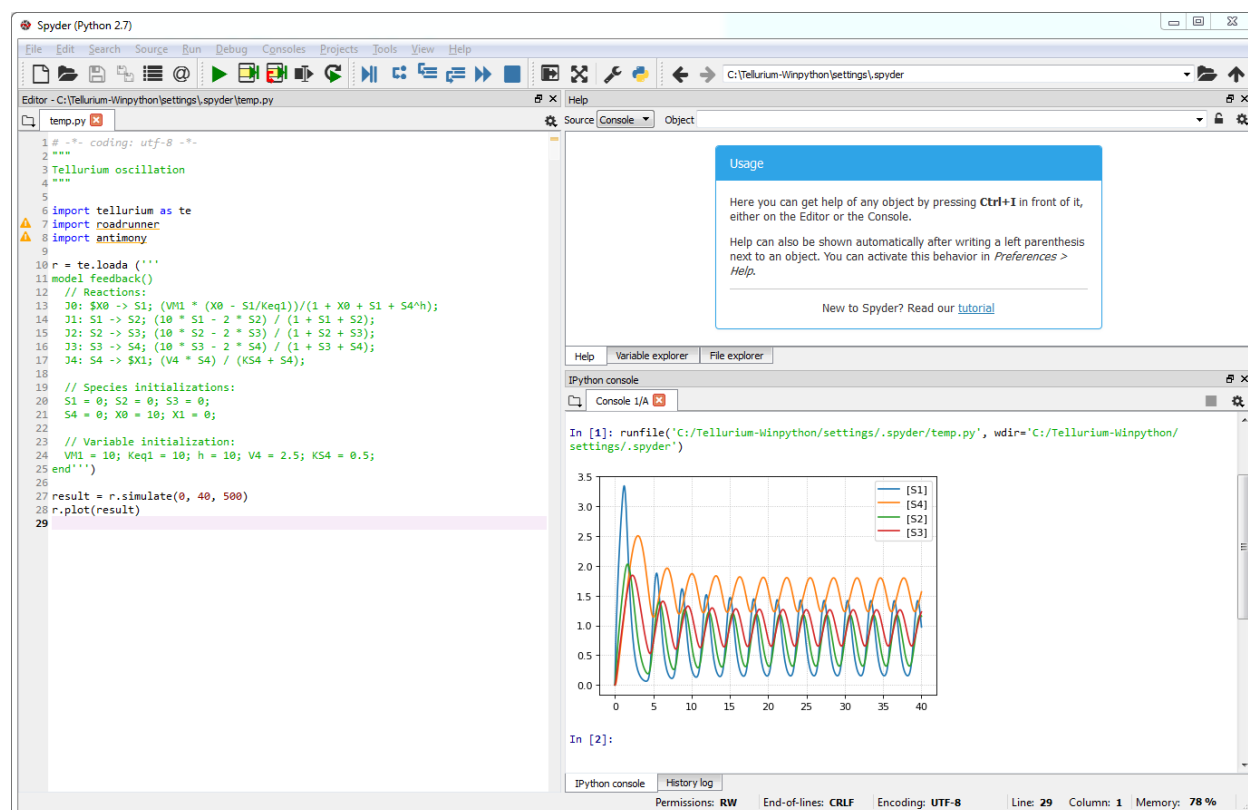


Fig. 22: Output of running the example script

For more information on how to build and simulate a model, check [Quick Start](#) and [libRoadRunner Tutorial](#).

3.5.2 Creating and Running Cells

Similar to Jupyter notebook, Spyder IDE allows you to create cells. To create a cell, simply put `%%` in the script. Each `%%` will signal generation of a new cell. To run a cell, press `shift+enter` while in focus of a cell. If you want to run only part of a script, you can do it by drag-selecting the part and pressing `F9`.

3.5.3 Importing Files

Tellurium Spyder comes with few plugins to help you import SBML, SED-ML, and COMBINE archives. Under File menu, press `Open SBML file` to open an SBML file and automatically translate it into Antimony string in a new editor tab. To import SED-ML or COMBINE archive, go to `File -> Import`. You can import SED-ML or COMBINE archive using either `phraSED-ML` notation or raw Python output. Tellurium understands `phraSED-ML` notation.

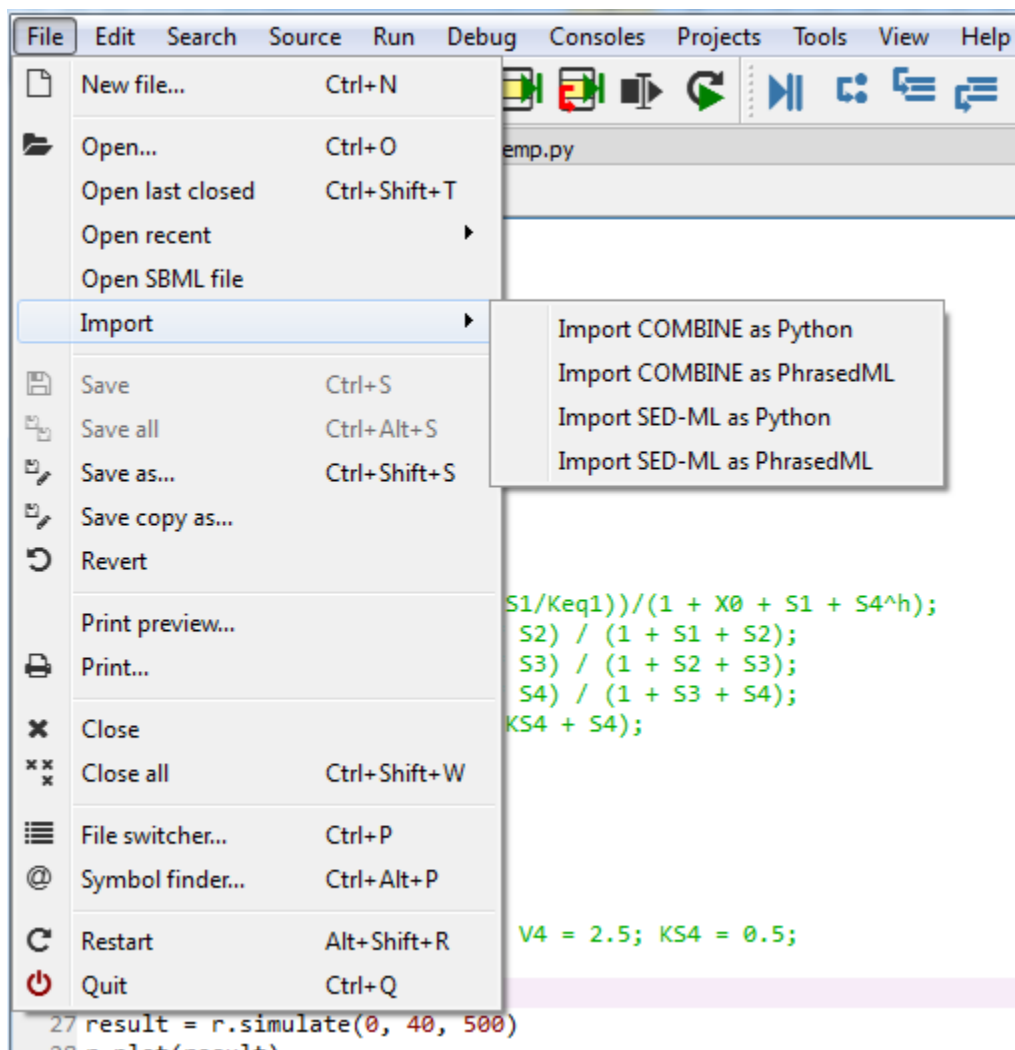


Fig. 23: Importing SBML, SED-ML, and COMBINE archives

3.5.4 RateLaw Plugin

Tellurium Spyder comes with RateLaw plugin. RateLaw plugin contains a list of various rate laws and allows users to insert rate laws directly to the editor. Simply put, it is a dictionary of rate laws so that you don't have to memorize it. To use it, go to Tools -> Rate Law Library. You can then choose a rate law, fill in the parameters if you wish, and press Insert Rate Law.

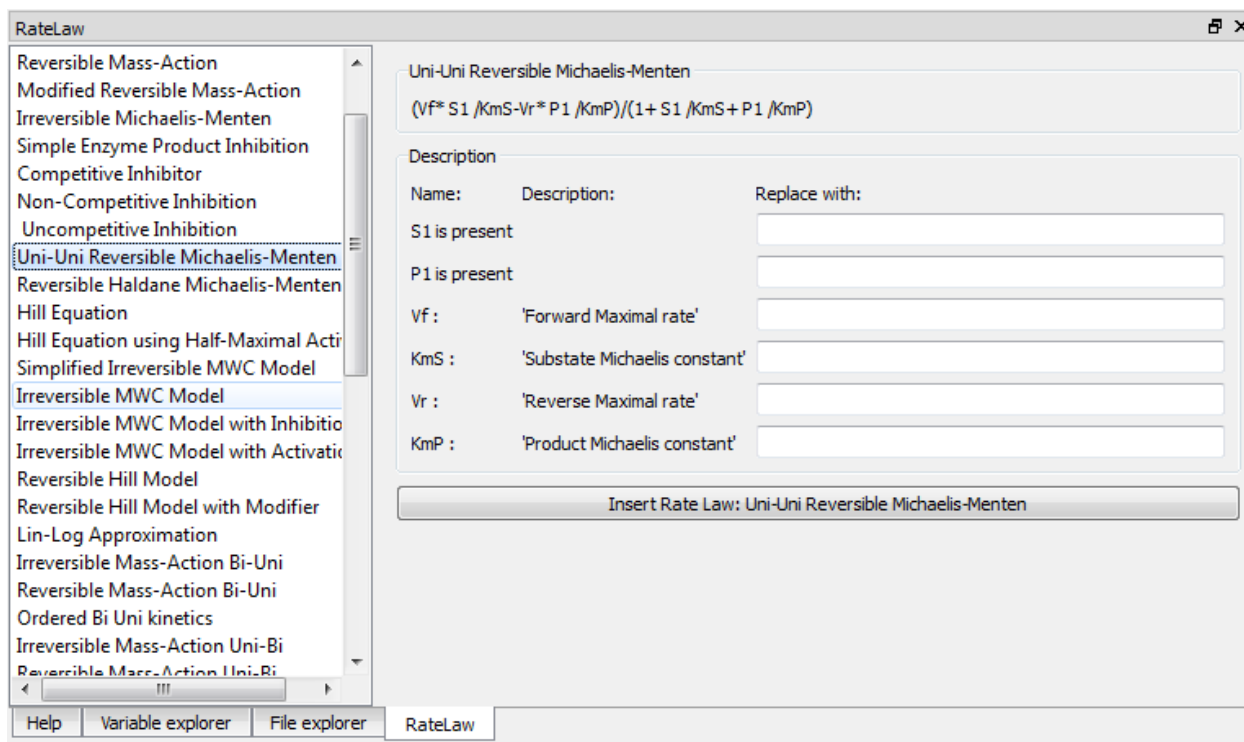


Fig. 24: RateLaw Plugin

3.5.5 Spyder Tips

Both the editor and console window support tab completion. If you are looking for certain functions under a module or a class, simply press `tab`.

Spyder IDE also supports various ways to check the documentations. If you wish to know more about a function, press `ctrl+i` while the cursor is next to the function to pull up the documentation in the Help window or execute `help(<function>)` in IPython console. You can also use `<function>?` to know more about the object itself. IPython offers various IPython-specific magic commands. Check it out by running `?` without any arguments in the IPython console.

Sometimes, IPython console might crash. Sometimes, you might want to restart the console to free up some memory. Yes, these things happen unfortunately. To restart the console, simply press `ctrl+.` in the console or right click -> Restart kernel. While doing so, you will notice that you can open another IPython console as well.

The screenshot displays the tellurium documentation interface. At the top, a blue header bar contains the text `getFloatingSpeciesIds`. Below this, a light gray box contains the following information:

- Definition :** `getFloatingSpeciesIds()`
- Type :** Method of `ExtendedRoadRunner` instance

Below the gray box, the text `ExecutableModel.getFloatingSpeciesIds()` is shown, followed by the description: "Return a list of floating species sbml ids."

At the bottom of the interface, there is a tabbed bar with three tabs: "Help", "Variable explorer", and "File explorer". The "Help" tab is currently selected. Below the tabs is a section labeled "IPython console". Inside the console, the following text is visible:

```
In [2]: help(r.getFloatingSpeciesIds)
Help on method getFloatingSpeciesIds in module tellurium.tellurium:

getFloatingSpeciesIds(self) method of tellurium.roadrunner.extended_roadrunner.ExtendedRoadRunner
instance
    ExecutableModel.getFloatingSpeciesIds()

    Return a list of floating species sbml ids.

In [3]:
```

Fig. 25: Pulling documentation in the Help window or through IPython console.

3.6 Advanced Topics on Tellurium Spyder

3.6.1 Running Jupyter Notebook

Tellurium Spyder comes with Jupyter Notebook by default. To run it, go to Start Menu -> Tellurium Winpython -> Launch Jupyter Notebook or go to Tellurium Spyder installation directory and run `Jupyter Notebook.exe`.

3.6.2 Running Command Prompt for Tellurium Spyder

Sometimes, you might want to run a Windows command prompt with the Python that comes with Tellurium Spyder as the default Python distribution. This can be useful if you wish to install additional Python packages with more control. To do so, go to Start Menu -> Tellurium Winpython -> WinPython Command Prompt or go to Tellurium Spyder installation directory and run `WinPython Command Prompt.exe`.

3.6.3 Running JupyterLab

Tellurium Spyder comes with JupyterLab by default. To run it, go to Start Menu -> Tellurium Winpython -> Launch JupyterLab or go to Tellurium Spyder installation directory and run `Jupyter Lab.exe`. For more information on JupyterLab, go to [Official JupyterLab Documentation](#).

3.7 Tellurium Spyder Troubleshooting

3.7.1 Problem: IPython Console Crashed

When this happens, IPython will automatically recover most of the times. If it does not, manually restart the IPython console using `ctrl+.` or right click -> Restart kernel.

3.7.2 Problem: Cannot Open Tellruim Spyder

When Spyder IDE crashes, it will automatically try to recover on the next execution. However, if this does not happen, manually run the reset script. To do so, go to Start Menu -> Tellurium Winpython -> Reset Spyder or go to Tellurium Spyder installation directory and run `Spyder reset.exe`. If Spyder still does not open, we suggest you to clean re-install Tellurium Spyder.

3.8 Further Reading

- [Official Spyder documentation](#)
-

3.9 Additional Resources for Tellurium

- For general questions or to request help, please post to the [Tellurium-discuss](#) mailing list.
- [Herbert Sauro's modeling textbook](#), which uses Tellurium
- [YouTube video tutorials](#) (made prior to Tellurium notebook).

3.10 Learning Python

- [Google's Python class](#).
- Official tutorial for [Python 2](#) and [Python 3](#).

CHAPTER 4

Usage Examples

All tellurium examples are available as interactive [Tellurium](#) or [Jupyter](#) notebooks.

To run the examples, clone the git repository:

```
git clone https://github.com/sys-bio/tellurium.git
```

and use the [Tellurium notebook viewer](#) or [Jupyter](#) to open any notebook in the `tellurium/examples/notebooks/core` directory.

Tellurium Spyder comes with these examples under Tellurium Spyder installation directory. Look for folder called `tellurium-winpython-examples`.

4.1 Basics

4.1.1 Model Loading

To load models use any the following functions. Each function takes a model with the corresponding format and converts it to a [RoadRunner](#) simulator instance.

- `te.loadAntimony(te.loada)`: Load an Antimony model.
- `te.loadSBML`: Load an SBML model.
- `te.loadCellML`: Load a CellML model (this passes the model through Antimony and converts it to SBML, may be lossy).

```
import tellurium as te
te.setDefaultPlottingEngine('matplotlib')

model = """
model test
    compartment C1;
    C1 = 1.0;
```

(continues on next page)

(continued from previous page)

```
species S1, S2;

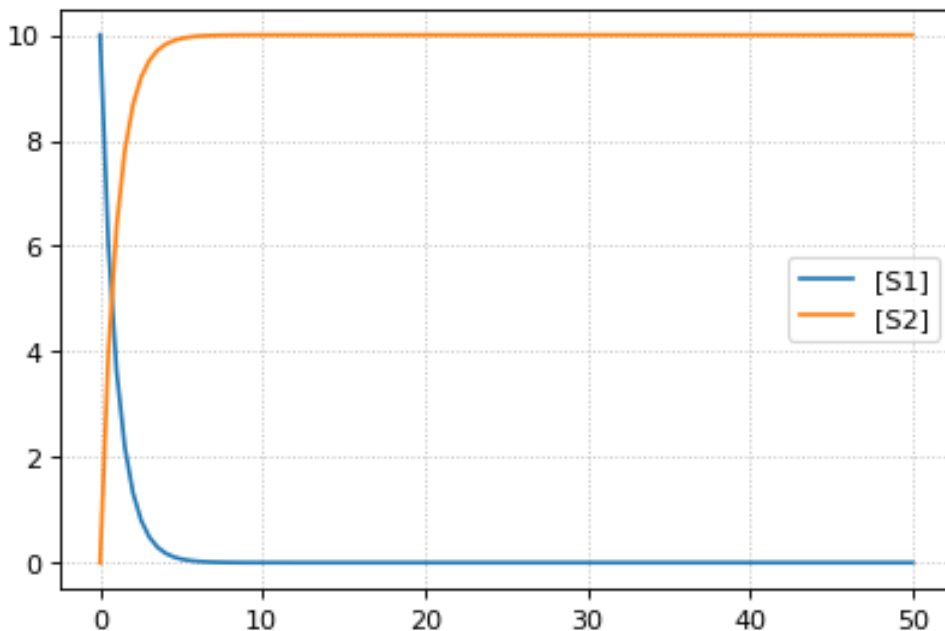
S1 = 10.0;
S2 = 0.0;
S1 in C1; S2 in C1;
J1: S1 -> S2; k1*S1;

k1 = 1.0;
end
"""
# load models
r = te.loada(model)
```

4.1.2 Running Simulations

Simulating a model in roadrunner is as simple as calling the `simulate` function on the RoadRunner instance `r`. The `simulate` accepts three positional arguments: start time, end time, and number of points. The `simulate` function also accepts the keyword arguments `selections`, which is a list of variables to include in the output, and `steps`, which is the number of integration time steps and can be specified instead of number of points.

```
# simulate from 0 to 50 with 100 steps
r.simulate(0, 50, 100)
# plot the simulation
r.plot()
```



4.1.3 Integrator and Integrator Settings

To set the integrator use `r.setIntegrator(<integrator-name>)` or `r.integrator = <integrator-name>`. RoadRunner supports 'cvoid', 'gillespie', and 'rk4' for the integrator

name. CVODE uses adaptive stepping internally, regardless of whether the output is gridded or not. The size of these internal steps is controlled by the tolerances, both absolute and relative.

To set integrator settings use `r.integrator.<setting-name> = <value>` or `r.integrator.setValue(<setting-name>, <value>)`. Here are some important settings for the `cvode` integrator:

- `variable_step_size`: Adaptive step-size integration (True/False).
- `stiff`: Stiff solver for CVODE only (True/False). Enabled by default.
- `absolute_tolerance`: Absolute numerical tolerance for integrator internal stepping.
- `relative_tolerance`: Relative numerical tolerance for integrator internal stepping.

Settings for the `gillespie` integrator:

- `seed`: The RNG seed for the Gillespie method. You can set this before running a simulation, or leave it alone for a different seed each time. Simulations initialized with the same seed will have the same results.

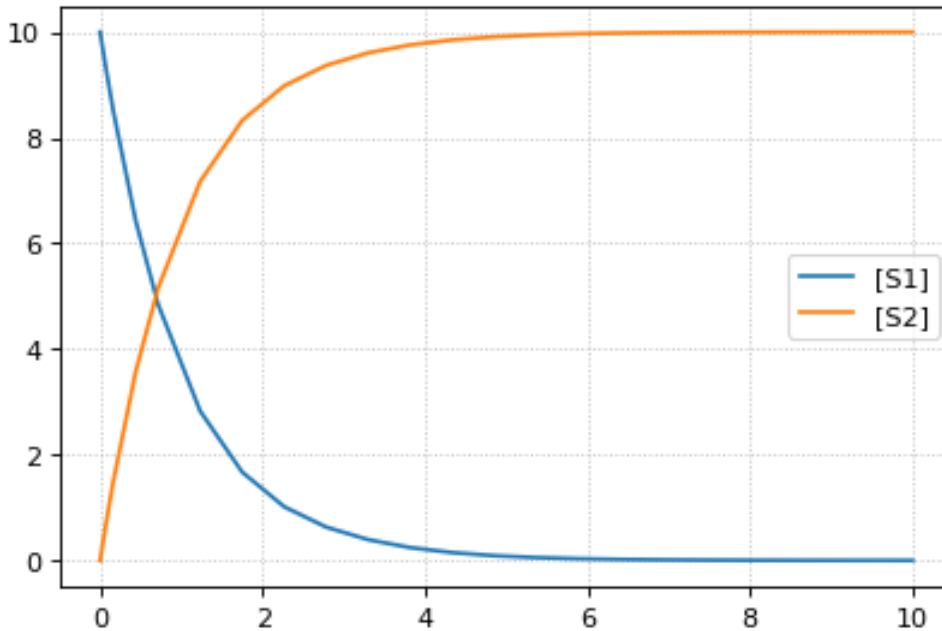
```
# what is the current integrator?
print('The current integrator is:')
print(r.integrator)

# enable variable stepping
r.integrator.variable_step_size = True
# adjust the tolerances (can set directly or via setValue)
r.integrator.absolute_tolerance = 1e-3 # set directly via property
r.integrator.setValue('relative_tolerance', 1e-1) # set via a call to setValue

# run a simulation, stop after reaching or passing time 10
r.reset()
results = r.simulate(0, 10)
r.plot()

# print the time values from the simulation
print('Time values:')
print(results[:,0])
```

```
The current integrator is:
< roadrunner.Integrator() >
  name: cvode
  settings:
    relative_tolerance: 0.000001
    absolute_tolerance: 0.000000000001
    stiff: true
    maximum_bdf_order: 5
    maximum_adams_order: 12
    maximum_num_steps: 20000
    maximum_time_step: 0
    minimum_time_step: 0
    initial_time_step: 0
    multiple_steps: false
    variable_step_size: false
```



Time values:

```
[0.00000000e+00 3.43225906e-07 3.43260229e-03 3.77551929e-02
 7.20777836e-02 1.60810095e-01 4.37546265e-01 7.14282434e-01
 1.23145372e+00 1.74862501e+00 2.26579629e+00 2.78296758e+00
 3.30013887e+00 3.81731015e+00 4.33448144e+00 4.85165273e+00
 5.36882401e+00 5.88599530e+00 6.40316659e+00 6.92033787e+00
 7.43750916e+00 7.95468045e+00 8.47185173e+00 9.25832855e+00
 1.00000000e+01]
```

```
# set integrator to Gillespie solver
r.setIntegrator('gillespie')
# identical ways to set integrator
r.setIntegrator('rk4')
r.integrator = 'rk4'
# set back to ccode (the default)
r.setIntegrator('ccode')

# set integrator settings
r.integrator.setValue('variable_step_size', False)
r.integrator.setValue('stiff', True)

# print integrator settings
print(r.integrator)
```

```
< roadrunner.Integrator() >
name: ccode
settings:
  relative_tolerance: 0.1
  absolute_tolerance: 0.001
  stiff: true
  maximum_bdf_order: 5
  maximum_adams_order: 12
  maximum_num_steps: 20000
```

(continues on next page)

(continued from previous page)

```

maximum_time_step: 0
minimum_time_step: 0
initial_time_step: 0
    multiple_steps: false
variable_step_size: false

```

4.1.4 Simulation options

The `RoadRunner.simulate` method is responsible for running simulations using the current integrator. It accepts the following arguments:

- `start`: Start time.
- `end`: End time.
- `points`: Number of points in solution (exclusive with steps, do not pass both). If the output is gridded, the points will be evenly spaced in time. If not, the simulation will stop when it reaches the `end` time or the number of points, whichever happens first.
- `steps`: Number of steps in solution (exclusive with points, do not pass both).

```

# simulate from 0 to 6 with 6 points in the result
r.reset()
# pass args explicitly via keywords
res1 = r.simulate(start=0, end=10, points=6)
print(res1)
r.reset()
# use positional args to pass start, end, num. points
res2 = r.simulate(0, 10, 6)
print(res2)

```

```

time,      [S1],      [S2]
[[ 0,        10,        0],
 [ 2,    1.23775,  8.76225],
 [ 4,    0.253289,  9.74671],
 [ 6,    0.0444091,  9.95559],
 [ 8,    0.00950381,  9.9905],
 [ 10, 0.00207671,  9.99792]]

time,      [S1],      [S2]
[[ 0,        10,        0],
 [ 2,    1.23775,  8.76225],
 [ 4,    0.253289,  9.74671],
 [ 6,    0.0444091,  9.95559],
 [ 8,    0.00950381,  9.9905],
 [ 10, 0.00207671,  9.99792]]

```

4.1.5 Selections

The selections list can be used to set which state variables will appear in the output array. By default, it includes all SBML species and the `time` variable. Selections can be given as an argument to `r.simulate`.

```

print('Floating species in model:')
print(r.getFloatingSpeciesIds())

```

(continues on next page)

(continued from previous page)

```
# provide selections to simulate
print(r.simulate(0,10,6, selections=r.getFloatingSpeciesIds()))
r.resetAll()
# try different selections
print(r.simulate(0,10,6, selections=['time','J1']))
```

```
Floating species in model:
['S1', 'S2']

      S1,      S2
[[ 0.00207671, 9.99792],
 [ 0.000295112, 9.9997],
 [-0.000234598, 10.0002],
 [-0.000203385, 10.0002],
 [-9.474e-05, 10.0001],
 [-3.43429e-05, 10]]

      time,      J1
[[ 0, 10],
 [ 2, 1.23775],
 [ 4, 0.253289],
 [ 6, 0.0444091],
 [ 8, 0.00950381],
 [10, 0.00207671]]
```

4.1.6 Reset model variables

To reset the model's state variables use the `r.reset()` and `r.reset(SelectionRecord.*)` functions. If you have made modifications to parameter values, use the `r.resetAll()` function to reset parameters to their initial values when the model was loaded.

```
# show the current values
for s in ['S1', 'S2']:
    print('r.{s} == {}'.format(s, r[s]))
# reset initial concentrations
r.reset()
print('reset')
# S1 and S2 have now again the initial values
for s in ['S1', 'S2']:
    print('r.{s} == {}'.format(s, r[s]))
# change a parameter value
print('r.k1 before = {}'.format(r.k1))
r.k1 = 0.1
print('r.k1 after = {}'.format(r.k1))
# reset parameters
r.resetAll()
print('r.k1 after resetAll = {}'.format(r.k1))
```

```
r.S1 == 0.0020767122285295023
r.S2 == 9.997923287771478
reset
r.S1 == 10.0
r.S2 == 0.0
r.k1 before = 1.0
```

(continues on next page)

(continued from previous page)

```
r.k1 after = 0.1
r.k1 after resetAll = 1.0
```

4.2 Models & Model Building

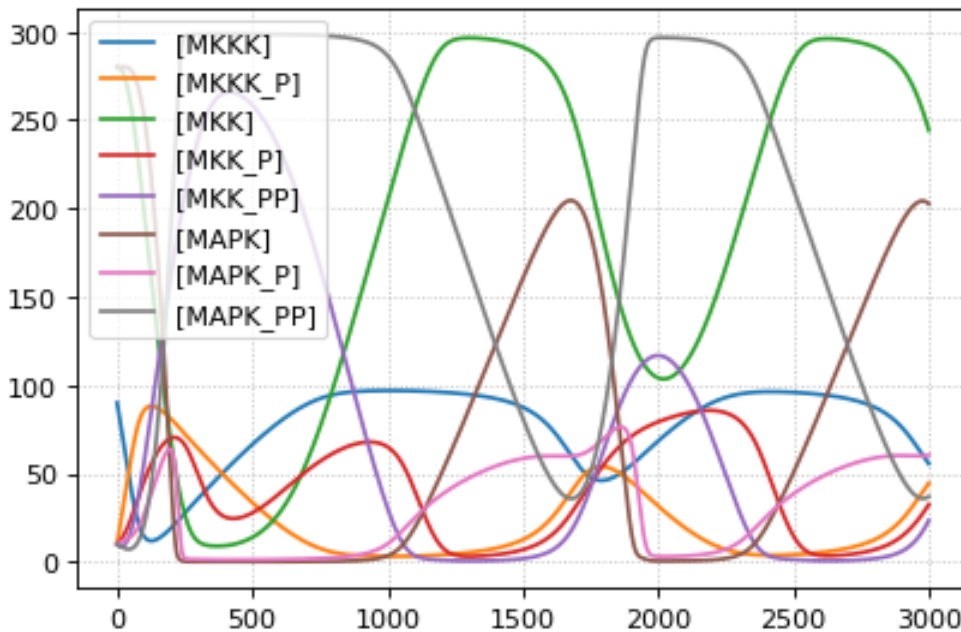
In this section, various types of models and different ways to building models are shown.

4.2.1 Model loading from BioModels

Models can be easily retrieved from BioModels using the `loadSBMLModel` function in Tellurium. This example uses a download URL to directly load `BIOMD0000000010`.

```
import tellurium as te

# Load model from biomodels (may not work with https).
r = te.loadSBMLModel("https://www.ebi.ac.uk/biomodels-main/download?
↳mid=BIOMD0000000010")
result = r.simulate(0, 3000, 5000)
r.plot(result)
```



4.2.2 Non-unit stoichiometries

The following example demonstrates how to create a model with non-unit stoichiometries.

```
import tellurium as te

r = te.loada('')
```

(continues on next page)

(continued from previous page)

```

model pathway()
  S1 + S2 -> 2 S3; k1*S1*S2
  3 S3 -> 4 S4 + 6 S5; k2*S3^3
  k1 = 0.1;k2 = 0.1;
end
'''
print(r.getCurrentAntimony())

```

```

// Created by libAntimony v2.9.4
model *pathway()

// Compartments and Species:
species S1, S2, S3, S4, S5;

// Reactions:
_J0: S1 + S2 -> 2 S3; k1*S1*S2;
_J1: 3 S3 -> 4 S4 + 6 S5; k2*S3^3;

// Species initializations:
S1 = 0;
S2 = 0;
S3 = 0;
S4 = 0;
S5 = 0;

// Variable initializations:
k1 = 0.1;
k2 = 0.1;

// Other declarations:
const k1, k2;
end

```

4.2.3 Consecutive UniUni reactions using first-order mass-action kinetics

Model creation and simulation of a simple irreversible chain of reactions $S1 \rightarrow S2 \rightarrow S3 \rightarrow S4$.

```

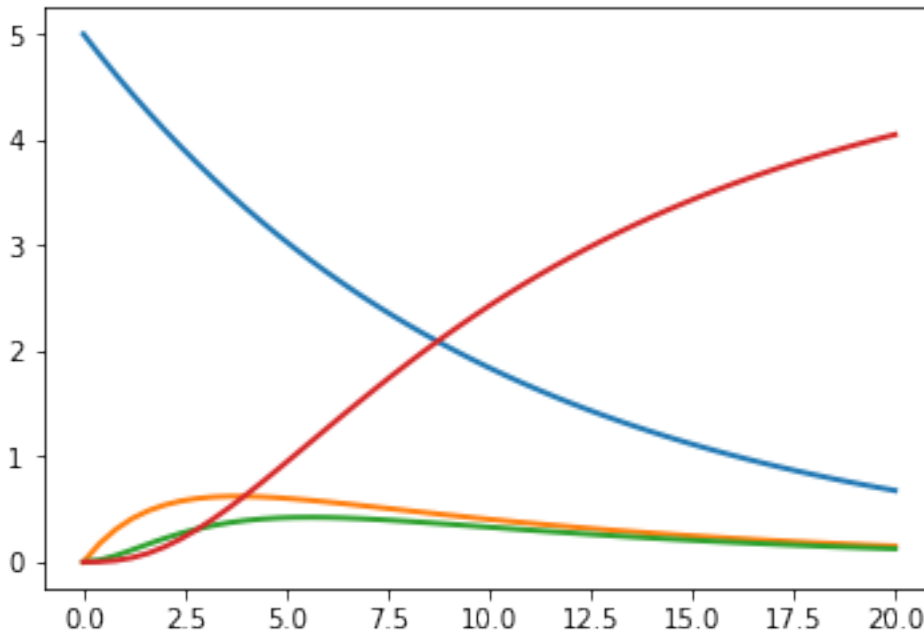
import tellurium as te

r = te.loada('''
model pathway()
  S1 -> S2; k1*S1
  S2 -> S3; k2*S2
  S3 -> S4; k3*S3

  # Initialize values
  S1 = 5; S2 = 0; S3 = 0; S4 = 0;
  k1 = 0.1; k2 = 0.55; k3 = 0.76
end
''')

result = r.simulate(0, 20, 51)
te.plotArray(result);

```



4.2.4 Generate different wave forms

Example for how to create different wave form functions in tellurium.

```
import tellurium as te
from roadrunner import Config

# We do not want CONSERVED MOIETIES set to true in this case
Config.setValue(Config.LOADSBMLOPTIONS_CONSERVED_MOIETIES, False)

# Generating different waveforms
model = '''
    model waveforms()
        # All waves have the following amplitude and period
        amplitude = 1
        period = 10

        # These events set the 'UpDown' variable to 1 or 0 according to the period.
        UpDown=0
        at sin(2*pi*time/period) > 0, t0=false: UpDown = 1
        at sin(2*pi*time/period) <= 0, t0=false: UpDown = 0

        # Simple Sine wave with y displaced by 3
        SineWave := amplitude/2*sin(2*pi*time/period) + 3

        # Square wave with y displaced by 1.5
        SquareWave := amplitude*UpDown + 1.5

        # Triangle waveform with given period and y displaced by 1
        TriangleWave = 1
        TriangleWave' = amplitude*2*(UpDown - 0.5)/period

        # Saw tooth wave form with given period
```

(continues on next page)

(continued from previous page)

```

SawTooth = amplitude/2
SawTooth' = amplitude/period
at UpDown==0: SawTooth = 0

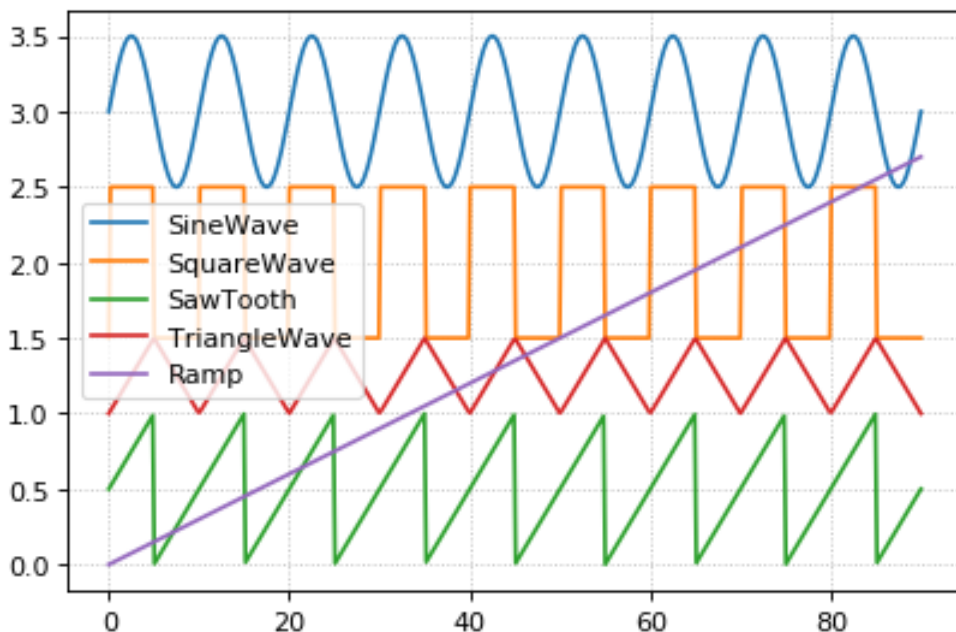
# Simple ramp
Ramp := 0.03*time
end
'''

r = te.loada(model)

r.timeCourseSelections = ['time', 'SineWave', 'SquareWave', 'SawTooth', 'TriangleWave', 'Ramp']
result = r.simulate(0, 90, 500)
r.plot(result)

# reset to default config
Config.setValue(Config.LOADSBMOPTIONS_CONSERVED_MOIETIES, False)

```



4.2.5 Normalized species

Example model which calculates functions depending on the normalized values of a species which can be either in active state SA or inactive state SI.

The normalized values are SA_f and SI_f, respectively, with the total concentration of S given as

$$ST = SA + SI$$

Model definition

The model is defined using Tellurium and Antimony. The identical equations could be typed directly in COPASI.

The created model is exported as SBML which than can be used in COPASI.

```
import tellurium as te
r = te.loada("""
    model normalized_species()

        # conversion between active (SA) and inactive (SI)
        J1: SA -> SI; k1*SA - k2*SI;
        k1 = 0.1; k2 = 0.02;

        # species
        species SA, SI, ST;
        SA = 10.0; SI = 0.0;
        const ST := SA + SI;

        SA is "active state S";
        SI is "inactive state S";
        ST is "total state S";

        # normalized species calculated via assignment rules
        species SA_f, SI_f;
        SA_f := SA/ST;
        SI_f := SI/ST;

        SA_f is "normalized active state S";
        SI_f is "normalized inactive state S";

        # parameters for your function
        P = 0.1;
        tau = 10.0;
        nA = 1.0;
        nI = 2.0;
        kA = 0.1;
        kI = 0.2;
        # now just use the normalized species in some math
        F := ( (1-(SI_f^nI)/(kI^nI+SI_f^nI)*(kI^nI+1) ) * ( (SA_f^nA)/(kA^nA+SA_f^nA)*(kA^
↪nA+1) ) -P)*tau;

    end
""")
# print(r.getAntimony())

# Store the SBML for COPASI
import os
import tempfile
temp_dir = tempfile.mkdtemp()
file_path = os.path.join(temp_dir, 'normalizedSpecies.xml')
r.exportToSBML(file_path)
```

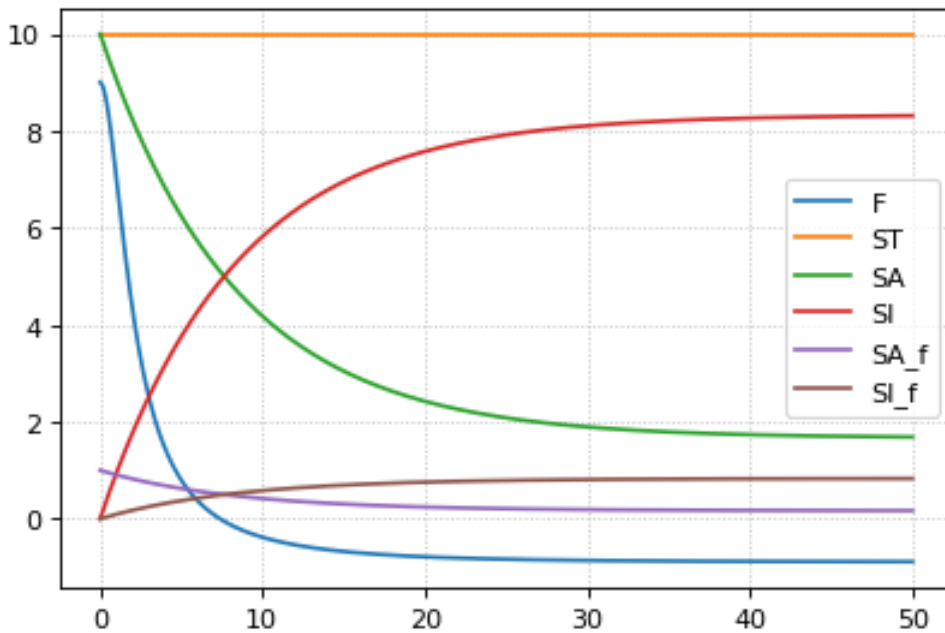
Model simulation

We perform a simple model simulation to demonstrate the main features using roadrunner: - normalized values SA_f and SI_f are normalized in $[0, 1]$ - the normalized values have same dynamics like SA and SI - the normalized values can be used to calculate some dependent function, here F

```

r.reset()
# select the variables of interest in output
r.selections = ['time', 'F'] + r.getBoundarySpeciesIds() \
               + r.getFloatingSpeciesIds()
# simulate from 0 to 50 with 1001 points
s = r.simulate(0,50,1001)
# plot the results
r.plot(s);

```



4.3 Simulation and Analysis Methods

In this section, different ways to simulate and analyse a model is shown.

Stochastic simulations can be run by changing the current integrator type to 'gillespie' or by using the `r.gillespie` function.

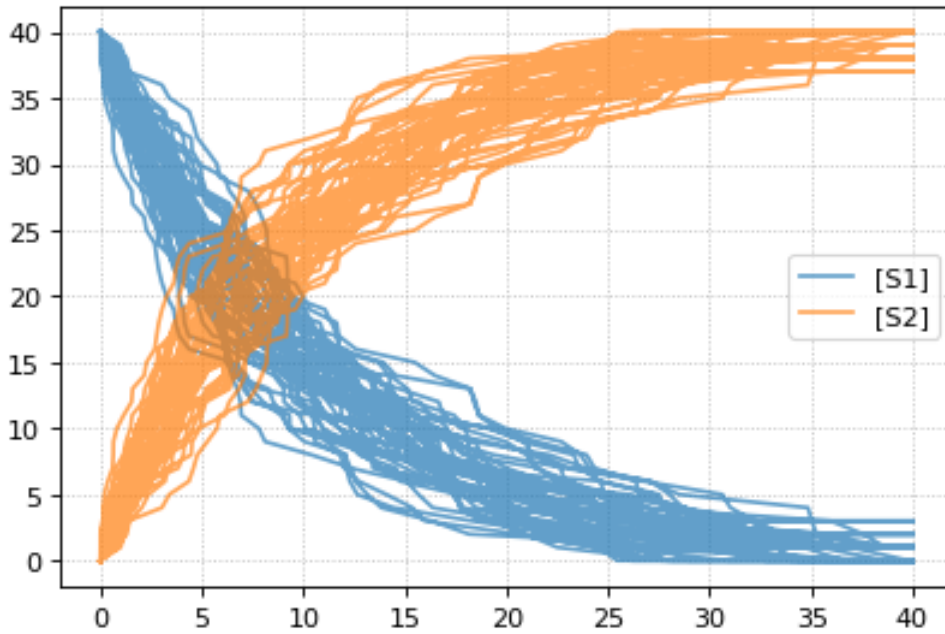
```

import tellurium as te
import numpy as np

r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 40')
r.integrator = 'gillespie'
r.integrator.seed = 1234

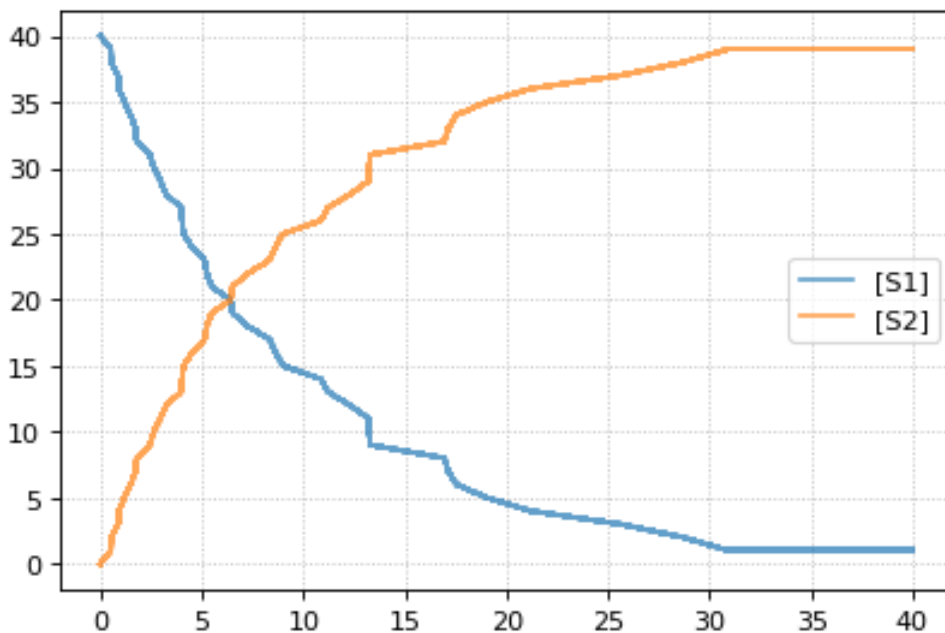
results = []
for k in range(1, 50):
    r.reset()
    s = r.simulate(0, 40)
    results.append(s)
    r.plot(s, show=False, alpha=0.7)
te.show()

```

Setting the identical seed for all repeats results in identical traces in each simulation.

```
results = []
for k in range(1, 20):
    r.reset()
    r.setSeed(123456)
    s = r.simulate(0, 40)
    results.append(s)
    r.plot(s, show=False, loc=None, color='black', alpha=0.7)
te.show()
```



You can combine two timecourse simulations and change e.g. parameter values in between each simulation. The

`gillespie` method simulates up to the given end time 10, after which you can make arbitrary changes to the model, then simulate again.

When using the `r.plot` function, you can pass the parameter `labels`, which controls the names that will be used in the figure legend, and `tag`, which ensures that traces with the same tag will be drawn with the same color (each species within each trace will be plotted in its own color, but these colors will match trace to trace).

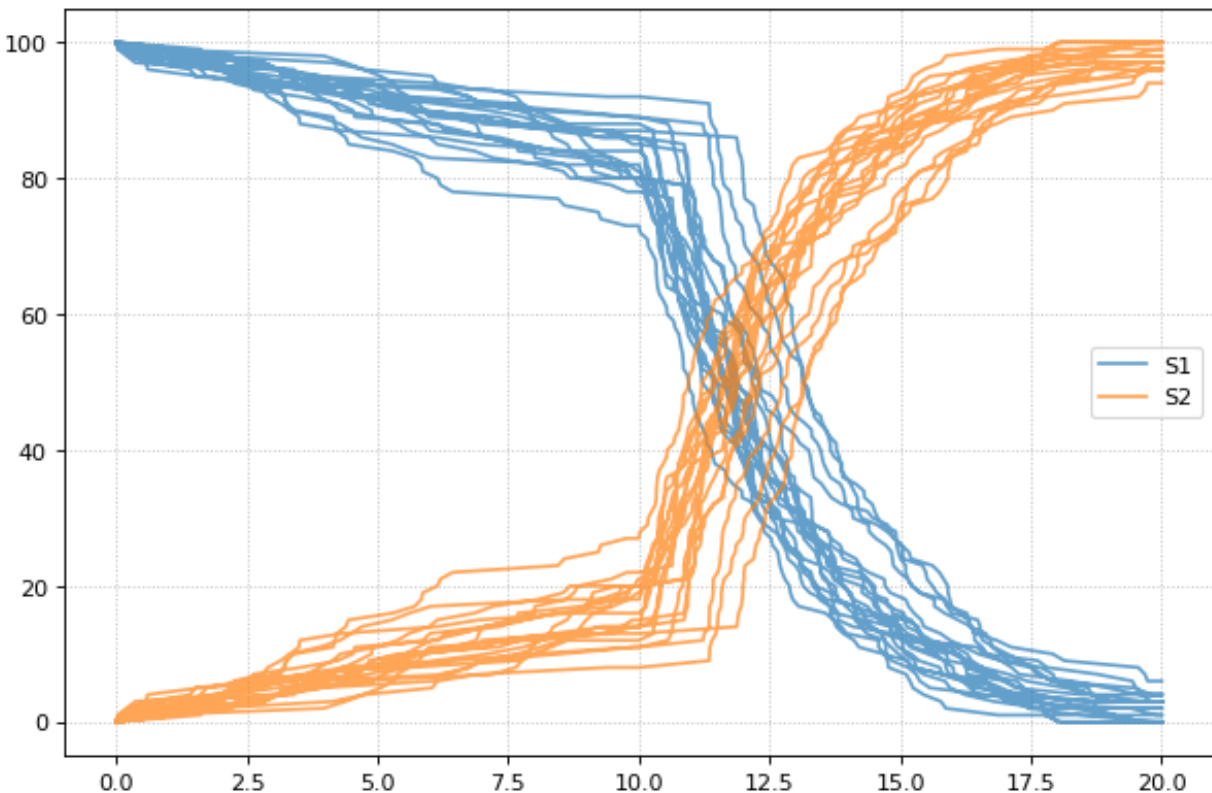
```
import tellurium as te

r = te.loada('S1 -> S2; k1*S1; k1 = 0.02; S1 = 100')
r.setSeed(1234)
for k in range(1, 20):
    r.resetToOrigin()
    res1 = r.gillespie(0, 10)
    r.plot(res1, show=False) # plot first half of data

    # change in parameter after the first half of the simulation
    # We could have also used an Event in the antimony model,
    # which are described further in the Antimony Reference section
    r.k1 = r.k1*20
    res2 = r.gillespie(10, 20)

    r.plot(res2, show=False) # plot second half of data

te.show()
```



4.4 SED-ML

Tellurium exchangeability via the simulation experiment description markup language [SED-ML](#) and [COMBINE archives](#) (.omex files). These formats are designed to allow modeling software to exchange models and simulations. Whereas SBML encodes models, SED-ML encodes simulations, including the solver (e.g. deterministic or stochastic), the type of simulation (timecourse or steady state), and various parameters (start/end time, ODE solver tolerances, etc.).

4.4.1 Working with SED-ML

SED-ML describes how to run a set of simulations on a model encoded in SBML or CellML through specifying tasks, algorithm parameters, and post-processing. SED-ML has a limited vocabulary of simulation types (timecourse and steady state) is not designed to replace scripting with Python or other general-purpose languages. Instead, SED-ML is designed to provide a rudimentary way to reproduce the dynamics of a model across different tools. This process would otherwise require human intervention and becomes very laborious when thousands of models are involved.

The basic elements of a SED-ML document are:

- **Models**, which reference external SBML/CellML files or other previously defined models within the same SED-ML document,
- **Simulations**, which reference specific numerical solvers from the [KiSAO ontology](#),
- **Tasks**, which apply a simulation to a model, and
- **Outputs**, which can be plots or reports.

Models in SED-ML essentially create instances of SBML/CellML models, and each instance can have different parameters.

Tellurium's approach to handling SED-ML is to first convert the SED-ML document to a Python script, which contains all the Tellurium-specific function calls to run all tasks described in the SED-ML. For authoring SED-ML, Tellurium uses PhraSEDML, a human-readable analog of SED-ML. Example syntax is shown below.

SED-ML files are not very useful in isolation. Since SED-ML always references external SBML and CellML files, software which supports exchanging SED-ML files should use COMBINE archives, which package all related standards-encoded files together.

Reproducible computational biology experiments with SED-ML - The Simulation Experiment Description Markup Language. Waltemath D., Adams R., Bergmann F.T., Hucka M., Kolpakov F., Miller A.K., Moraru I.I., Nickerson D., Snoep J.L., Le Novère, N. BMC Systems Biology 2011, 5:198 (<http://www.pubmed.org/22172142>)

Creating a SED-ML file

This example shows how to use PhraSEDML to author SED-ML files. Whenever a PhraSEDML script references an external model, you should use `phrasedml.setReferencedSBML` to ensure that the PhraSEDML script can be properly converted into a SED-ML file.

```
import tellurium as te
te.setDefaultPlottingEngine('matplotlib')
import phrasedml

antimony_str = '''
model myModel
  S1 -> S2; k1*S1
```

(continues on next page)

(continued from previous page)

```

    S1 = 10; S2 = 0
    k1 = 1
end
'''

phrasedml_str = '''
    model1 = model "myModel"
    sim1 = simulate uniform(0, 5, 100)
    task1 = run sim1 on model1
    plot "Figure 1" time vs S1, S2
'''

# create the sedml xml string from the phrasedml
sbml_str = te.antimonyToSBML(antimony_str)
phrasedml.setReferencedSBML("myModel", sbml_str)

sedml_str = phrasedml.convertString(phrasedml_str)
if sedml_str == None:
    print(phrasedml.getLastPhrasedError())
print(sedml_str)

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by phraSED-ML version v1.0.9 with libSBML version 5.15.0. -->
<sedML xmlns="http://sed-ml.org/sed-ml/level1/version2" level="1" version="2">
  <listOfSimulations>
    <uniformTimeCourse id="sim1" initialTime="0" outputStartTime="0" outputEndTime="5
→ " numberOfPoints="100">
      <algorithm kisaoID="KISAO:0000019"/>
    </uniformTimeCourse>
  </listOfSimulations>
  <listOfModels>
    <model id="model1" language="urn:sedml:language:sbml:level-3.version-1" source=
→ "myModel"/>
  </listOfModels>
  <listOfTasks>
    <task id="task1" modelReference="model1" simulationReference="sim1"/>
  </listOfTasks>
  <listOfDataGenerators>
    <dataGenerator id="plot_0_0_0" name="time">
      <listOfVariables>
        <variable id="time" symbol="urn:sedml:symbol:time" taskReference="task1"/>
      </listOfVariables>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <ci> time </ci>
      </math>
    </dataGenerator>
    <dataGenerator id="plot_0_0_1" name="S1">
      <listOfVariables>
        <variable id="S1" target="/sbml:sbml/sbml:model/sbml:listOfSpecies/
→ sbml:species[@id='S1']" taskReference="task1" modelReference="model1"/>
      </listOfVariables>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <ci> S1 </ci>
      </math>
    </dataGenerator>
    <dataGenerator id="plot_0_1_1" name="S2">

```

(continues on next page)

(continued from previous page)

```

    <listOfVariables>
      <variable id="S2" target="/sbml:sbml/sbml:model/sbml:listOfSpecies/
↳sbml:species[@id='S2']" taskReference="task1" modelReference="model1"/>
    </listOfVariables>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <ci> S2 </ci>
    </math>
  </dataGenerator>
</listOfDataGenerators>
<listOfOutputs>
  <plot2D id="plot_0" name="Figure 1">
    <listOfCurves>
      <curve id="plot_0__plot_0_0_0__plot_0_0_1" logX="false" logY="false"
↳xDataReference="plot_0_0_0" yDataReference="plot_0_0_1"/>
      <curve id="plot_0__plot_0_0_0__plot_0_1_1" logX="false" logY="false"
↳xDataReference="plot_0_0_0" yDataReference="plot_0_1_1"/>
    </listOfCurves>
  </plot2D>
</listOfOutputs>
</sedML>

```

4.4.2 Reading / Executing SED-ML

After converting PhraSEDML to SED-ML, you can call `te.executeSEDML` to use Tellurium to execute all simulations in the SED-ML. This example also shows how to use `libSEDML` (used by Tellurium and PhraSEDML internally) for reading SED-ML files.

```

import tempfile, os, shutil

workingDir = tempfile.mkdtemp(suffix="_sedml")

sbml_file = os.path.join(workingDir, 'myModel')
sedml_file = os.path.join(workingDir, 'sed_main.xml')

with open(sbml_file, 'wb') as f:
    f.write(sbml_str.encode('utf-8'))
    f.flush()
    print('SBML written to temporary file')

with open(sedml_file, 'wb') as f:
    f.write(sedml_str.encode('utf-8'))
    f.flush()
    print('SED-ML written to temporary file')

# For technical reasons, any software which uses libSEDML
# must provide a custom build - Tellurium uses tersedml
import tersedml as libsedml
sedml_doc = libsedml.readSedML(sedml_file)
n_errors = sedml_doc.getErrorLog().getNumFailsWithSeverity(libsedml.LIBSEDML_SEV_
↳ERROR)
print('Read SED-ML file, number of errors: {}'.format(n_errors))
if n_errors > 0:
    print(sedml_doc.getErrorLog().toString())

# execute SED-ML using Tellurium

```

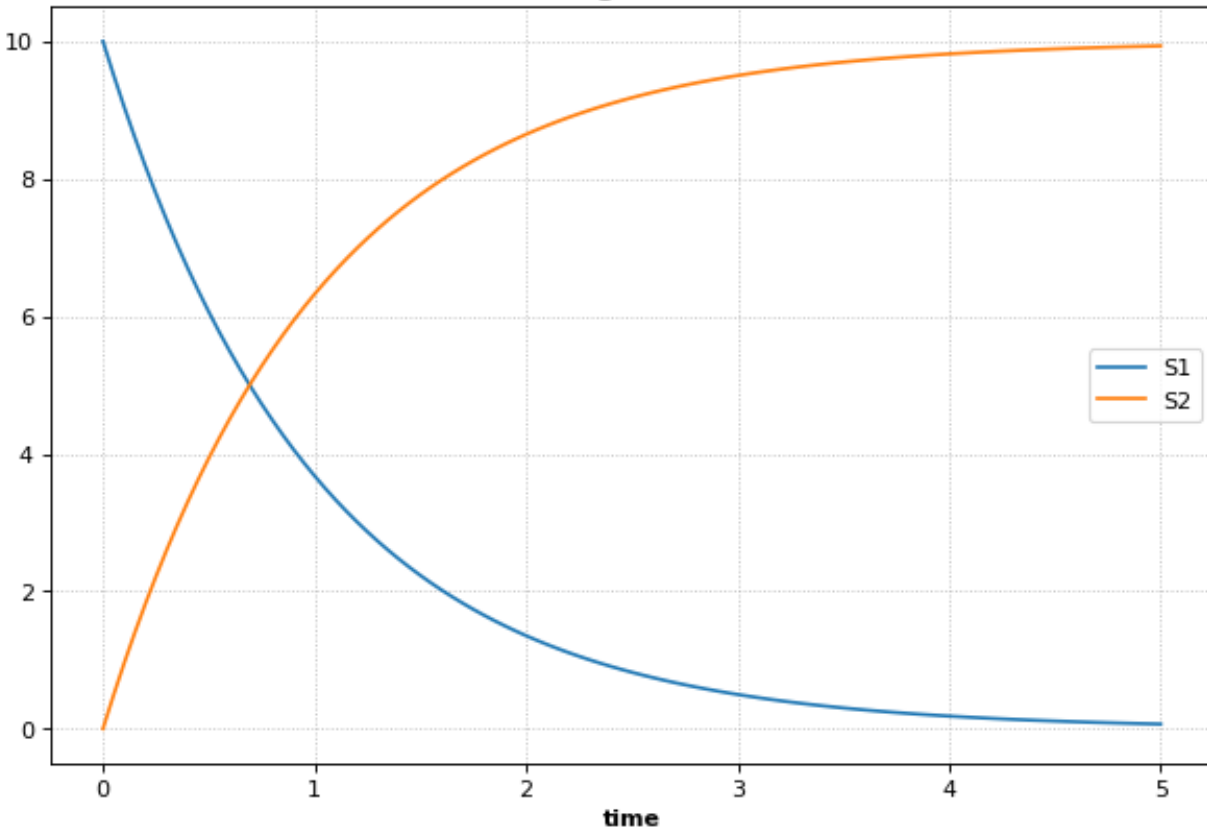
(continues on next page)

(continued from previous page)

```
te.executeSEDML(sedml_str, workingDir=workingDir)

# clean up
#shutil.rmtree(workingDir)
```

```
SBML written to temporary file
SED-ML written to temporary file
Read SED-ML file, number of errors: 0
```

Figure 1

4.4.3 SED-ML L1V2 specification example

This example uses the celebrated [repressilator model](#) to demonstrate how to 1) download a model from the [BioModels database](#), 2) create a PhraSEDML string to simulate the model, 3) convert the PhraSEDML to SED-ML, and 4) use Tellurium to execute the resulting SED-ML.

This and other examples here are the [SED-ML reference specification](#) (Introduction section).

```
import tellurium as te, tellurium.temiriam as temiriam
te.setDefaultPlottingEngine('matplotlib')
import phrasedml

# Get SBML from URN and set for phrasedml
urn = "urn:miriam:biomodels.db:BIOMD0000000012"
sbml_str = temiriam.getSBMLFromBiomodelsURN(urn=urn)
```

(continues on next page)

(continued from previous page)

```

phrasedml.setReferencedSBML('BIOMD0000000012', sbml_str)

# <SBML species>
#   PX - LacI protein
#   PY - TetR protein
#   PZ - cI protein
#   X - LacI mRNA
#   Y - TetR mRNA
#   Z - cI mRNA

# <SBML parameters>
#   ps_a - tps_active: Transcription from free promotor in transcripts per second and
#   ↪promotor
#   ps_0 - tps_repr: Transcription from fully repressed promotor in transcripts per
#   ↪second and promotor

phrasedml_str = """
    model1 = model "{}"
    model2 = model model1 with ps_0=1.3E-5, ps_a=0.013
    sim1 = simulate uniform(0, 1000, 1000)
    task1 = run sim1 on model1
    task2 = run sim1 on model2

    # A simple timecourse simulation
    plot "Figure 1.1 Timecourse of repressilator" task1.time vs task1.PX, task1.PZ,
    ↪task1.PY

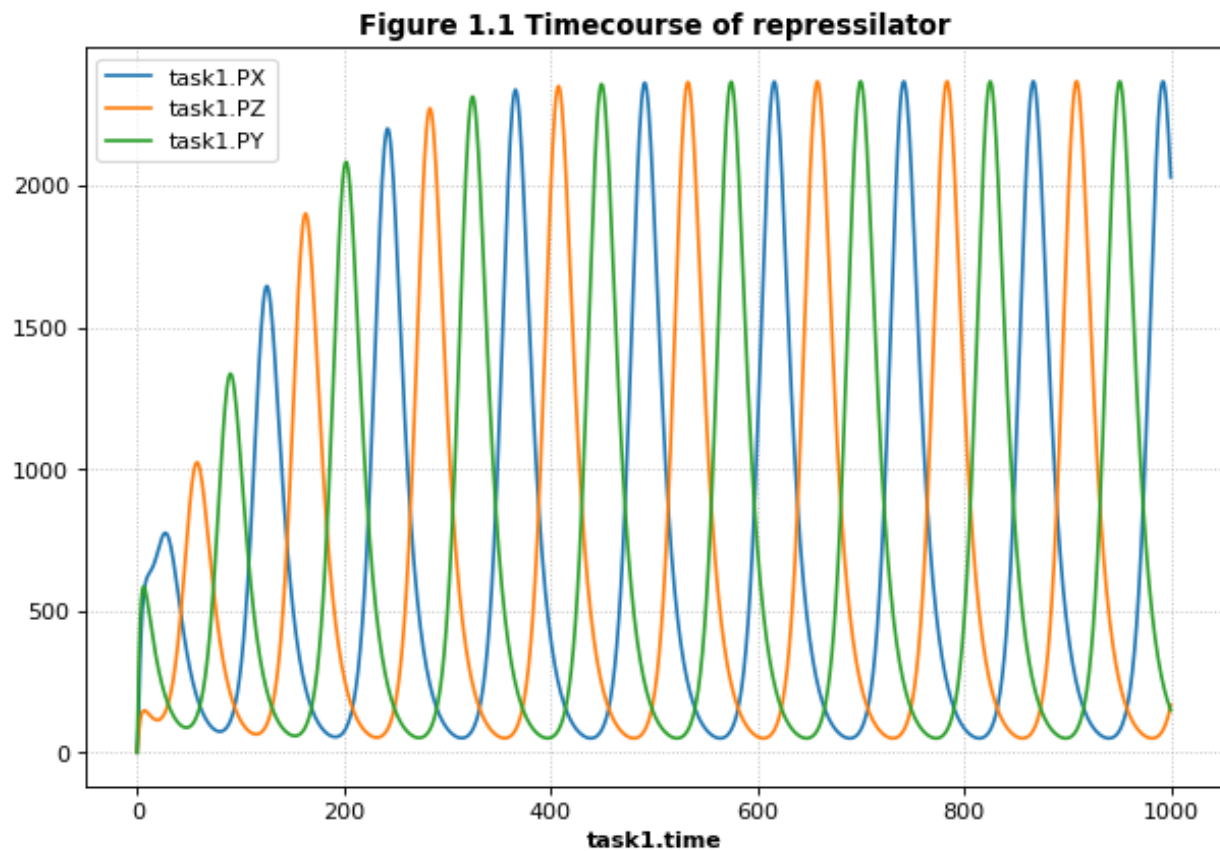
    # Applying preprocessing
    plot "Figure 1.2 Timecourse after pre-processing" task2.time vs task2.PX, task2.
    ↪PZ, task2.PY

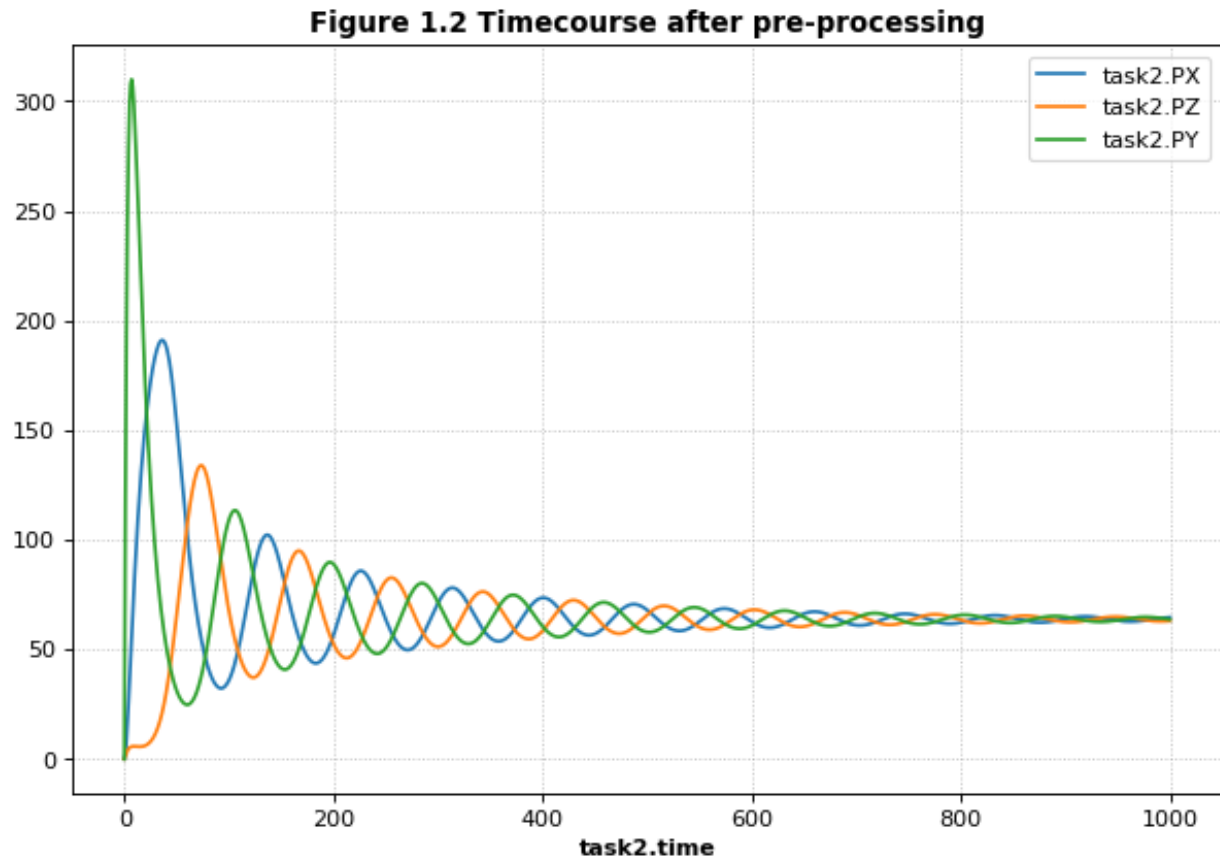
    # Applying postprocessing
    plot "Figure 1.3 Timecourse after post-processing" task1.PX/max(task1.PX) vs
    ↪task1.PZ/max(task1.PZ), \
                                                    task1.PY/max(task1.PY) vs
    ↪task1.PX/max(task1.PX), \
                                                    task1.PZ/max(task1.PZ) vs
    ↪task1.PY/max(task1.PY)
    """.format('BIOMD0000000012')

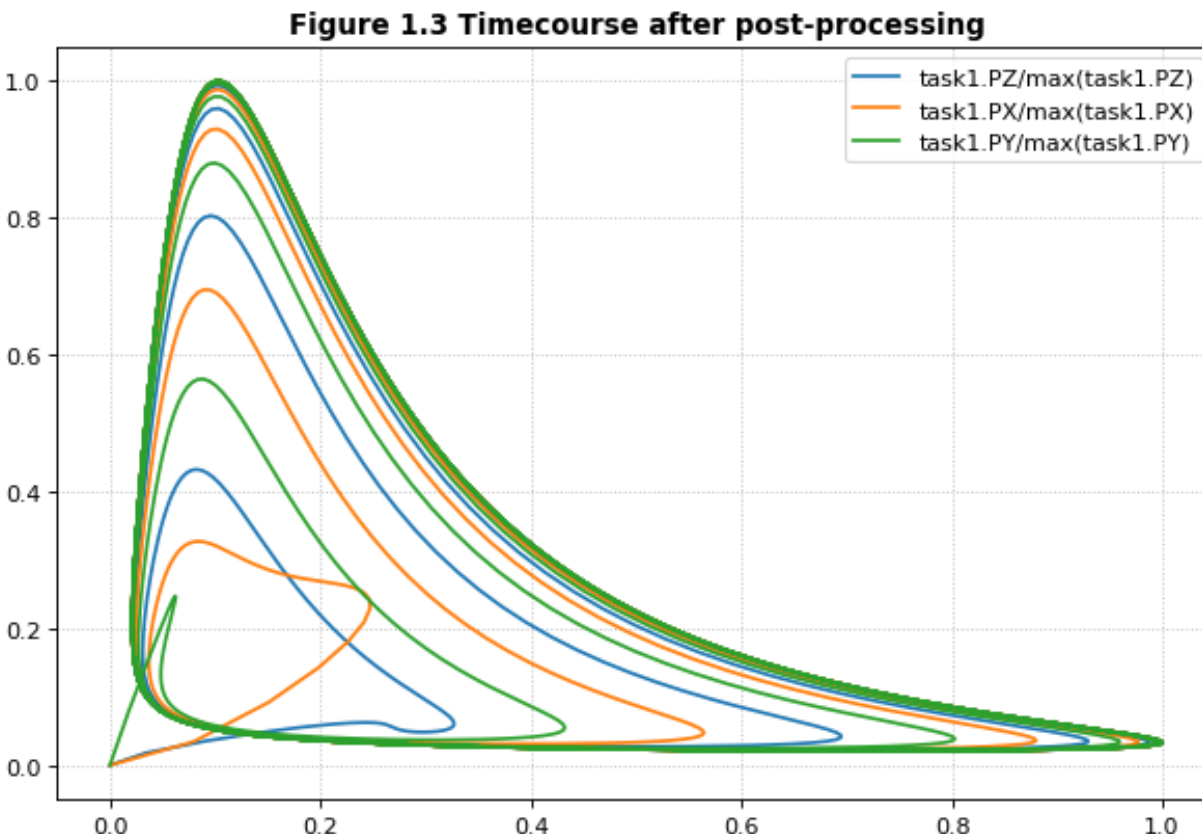
# convert to SED-ML
sedml_str = phrasedml.convertString(phrasedml_str)
if sedml_str == None:
    raise RuntimeError(phrasedml.getLastErrorMessage())

# Run the SED-ML file with results written in workingDir
import tempfile, shutil, os
workingDir = tempfile.mkdtemp(suffix="_sedml")
# write out SBML
with open(os.path.join(workingDir, 'BIOMD0000000012'), 'wb') as f:
    f.write(sbml_str.encode('utf-8'))
te.executeSEDML(sedml_str, workingDir=workingDir)
shutil.rmtree(workingDir)

```







4.5 COMBINE & Inline OMEX

COMBINE archives package related standards such as SBML models and SED-ML simulations together so that they can be easily exchanged between software tools. Tellurium provides the *inline OMEX* format for editing the contents of COMBINE archives in a human-readable format. You can use the function `convertCombineArchive` to convert a COMBINE archive on disk to an inline OMEX string, and the function `executeInlineOmex` to execute the inline OMEX string. Examples below.

```
tellurium.convertCombineArchive(location)
```

Read a COMBINE archive and convert its contents to an inline Omex.

Parameters `location` – Filesystem path to the archive.

```
tellurium.executeInlineOmex(inline_omex, comp=False)
```

Execute inline phrasedml and antimony.

Parameters `inline_omex` – String containing inline phrasedml and antimony.

```
tellurium.exportInlineOmex(inline_omex, export_location)
```

Export an inline OMEX string to a COMBINE archive.

Parameters

- `inline_omex` – String containing inline OMEX describing models and simulations.
- `export_location` – Filepath of Combine archive to create.

```
tellurium.extractFileFromCombineArchive(archive_path, entry_location)
```

Extract a single file from a COMBINE archive and return it as a string.

4.5.1 Inline OMEX and COMBINE archives

Tellurium provides a way to easily edit the contents of COMBINE archives in a human-readable format called inline OMEX. To create a COMBINE archive, simply create a string containing all models (in Antimony format) and all simulations (in PhraSEDML format). Tellurium will transparently convert the Antimony to SBML and PhraSEDML to SED-ML, then execute the resulting SED-ML. The following example will work in either Jupyter or the [Tellurium notebook viewer](#). The Tellurium notebook viewer allows you to create specialized cells for inline OMEX, which contain correct syntax-highlighting for the format.

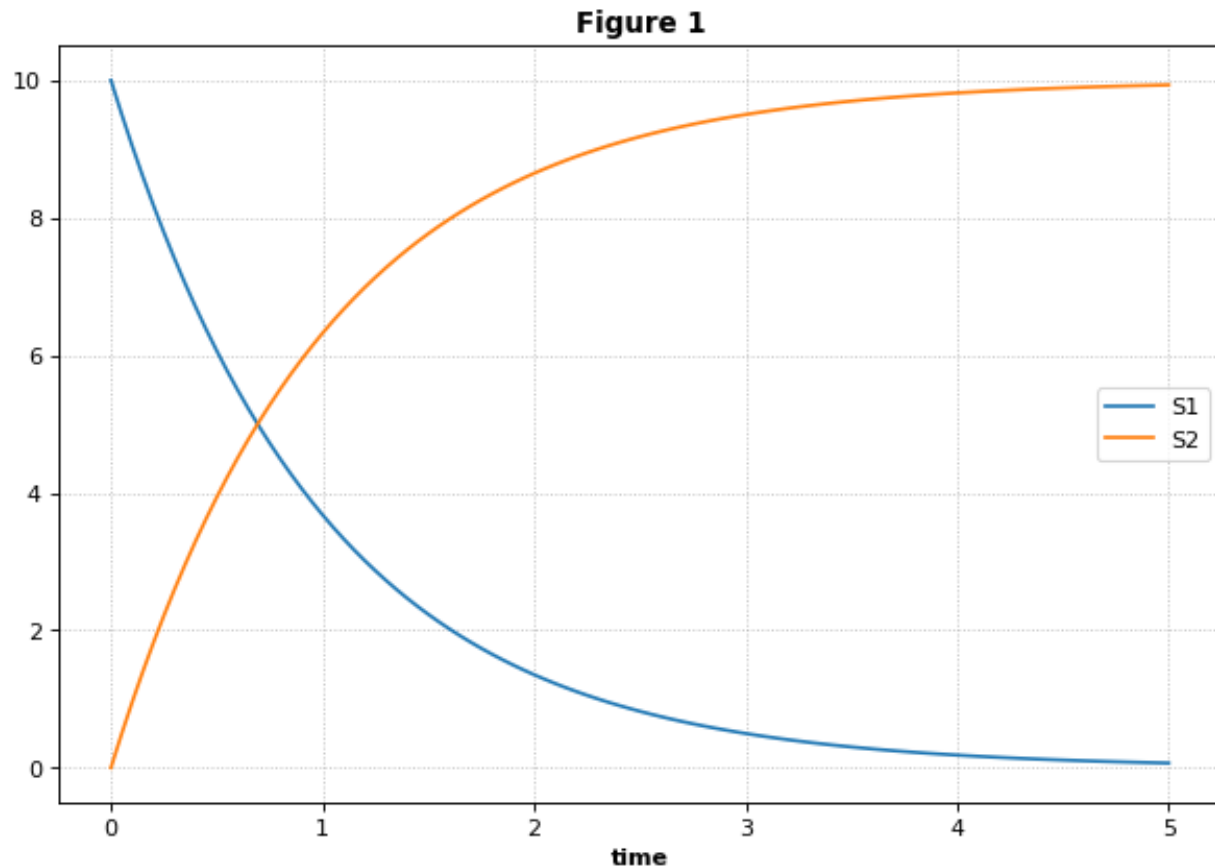
```
import tellurium as te, tempfile, os
te.setDefaultPlottingEngine('matplotlib')

antimony_str = '''
model myModel
  S1 -> S2; k1*S1
  S1 = 10; S2 = 0
  k1 = 1
end
'''

phrasedml_str = '''
modell = model "myModel"
sim1 = simulate uniform(0, 5, 100)
task1 = run sim1 on modell
plot "Figure 1" time vs S1, S2
'''

# create an inline OMEX (inline representation of a COMBINE archive)
# from the antimony and phrasedml strings
inline_omex = '\n'.join([antimony_str, phrasedml_str])

# execute the inline OMEX
te.executeInlineOmex(inline_omex)
# export to a COMBINE archive
workingDir = tempfile.mkdtemp(suffix="_omex")
te.exportInlineOmex(inline_omex, os.path.join(workingDir, 'archive.omex'))
```



4.5.2 Forcing Functions

A common task in modeling is to represent the influence of an external, time-varying input on the system. In SED-ML, this can be accomplished using a repeated task to run a simulation for a short amount of time and update the forcing function between simulations. In the example, the forcing function is a pulse represented with a `piecewise` directive, but it can be any arbitrarily complex time-varying function, as shown in the second example.

```
import tellurium as te

antimony_str = '''
// Created by libAntimony v2.9
model *oneStep()

// Compartments and Species:
compartment compartment_;
species S1 in compartment_, S2 in compartment_, $X0 in compartment_, $X1 in_
↪compartment_;
species $X2 in compartment_;

// Reactions:
J0: $X0 => S1; J0_v0;
J1: S1 => $X1; J1_k3*S1;
J2: S1 => S2; (J2_k1*S1 - J2_k_1*S2)*(1 + J2_c*S2^J2_q);
J3: S2 => $X2; J3_k2*S2;
```

(continues on next page)

(continued from previous page)

```
// Species initializations:
S1 = 0;
S2 = 1;
X0 = 1;
X1 = 0;
X2 = 0;

// Compartment initializations:
compartment_ = 1;

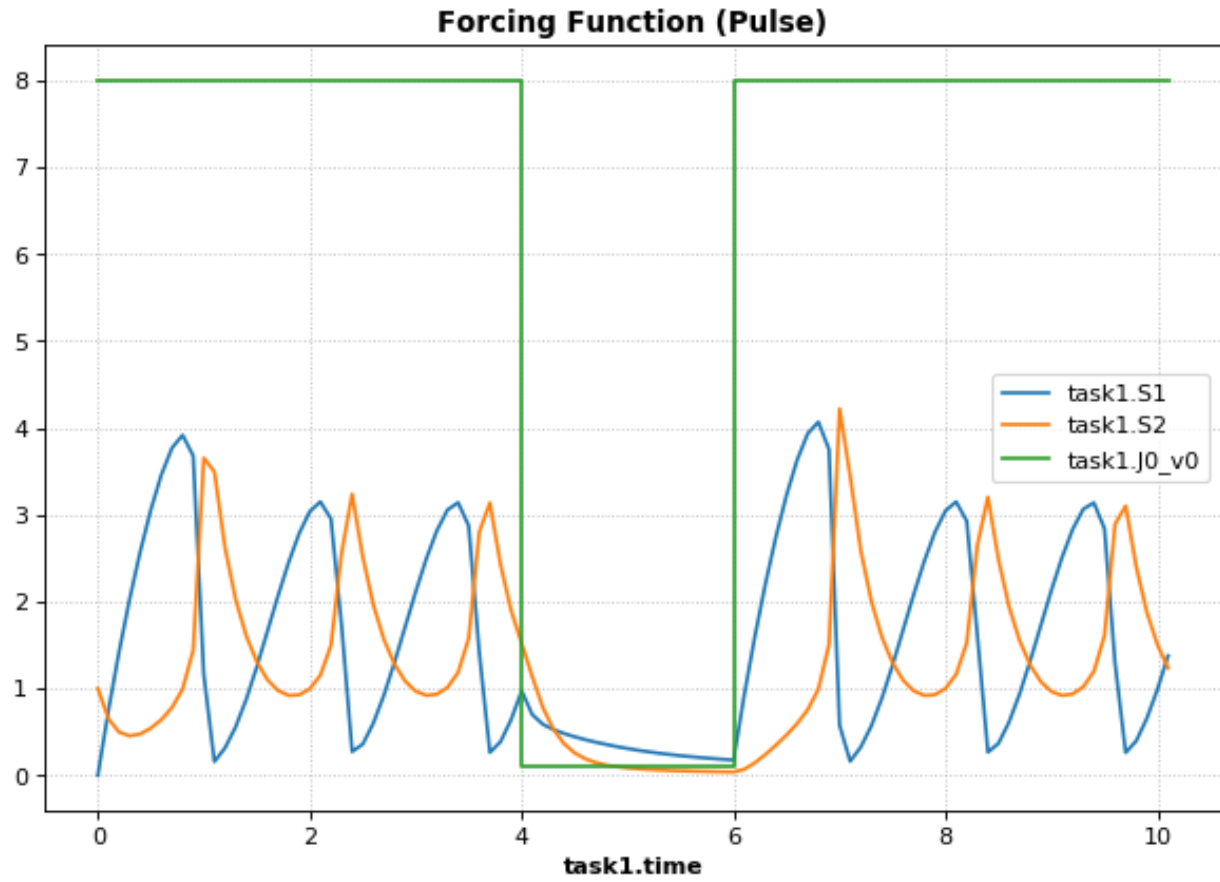
// Variable initializations:
J0_v0 = 8;
J1_k3 = 0;
J2_k1 = 1;
J2_k_1 = 0;
J2_c = 1;
J2_q = 3;
J3_k2 = 5;

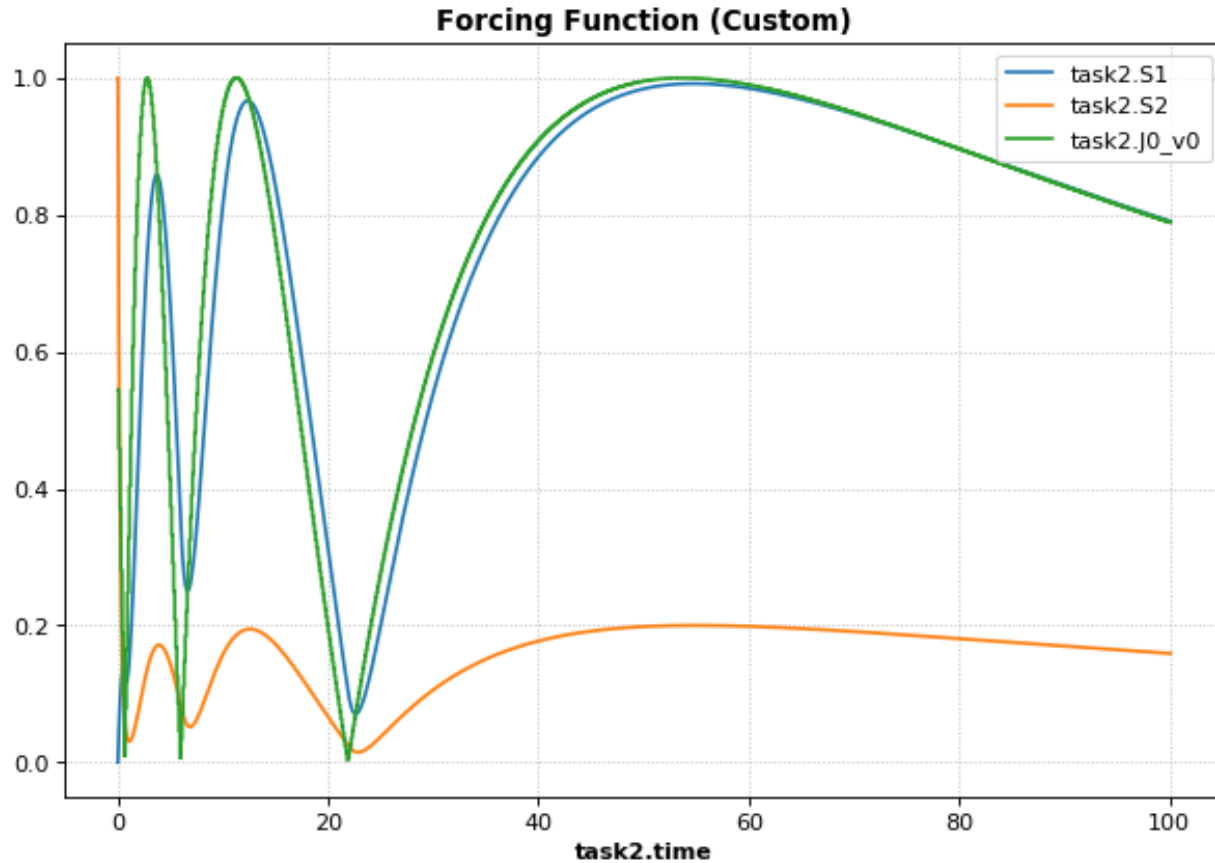
// Other declarations:
const compartment_, J0_v0, J1_k3, J2_k1, J2_k_1, J2_c, J2_q, J3_k2;
end
'''

phrasedml_str = '''
modell = model "oneStep"
stepper = simulate onestep(0.1)
task0 = run stepper on modell
task1 = repeat task0 for local.x in uniform(0, 10, 100), J0_v0 = piecewise(8, x<4, 0.
↳ 1, 4<=x<6, 8)
task2 = repeat task0 for local.index in uniform(0, 10, 1000), local.current = index ->
↳ abs(sin(1 / (0.1 * index + 0.1))), modell.J0_v0 = current : current
plot "Forcing Function (Pulse)" task1.time vs task1.S1, task1.S2, task1.J0_v0
plot "Forcing Function (Custom)" task2.time vs task2.S1, task2.S2, task2.J0_v0
'''

# create the inline OMEX string
inline_omex = '\n'.join([antimony_str, phrasedml_str])

# export to a COMBINE archive
workingDir = tempfile.mkdtemp(suffix="_omex")
archive_name = os.path.join(workingDir, 'archive.omex')
te.exportInlineOmex(inline_omex, archive_name)
# convert the COMBINE archive back into an
# inline OMEX (transparently) and execute it
te.convertAndExecuteCombineArchive(archive_name)
```





4.5.3 1d Parameter Scan

This example shows how to perform a one-dimensional parameter scan using Antimony/PhraSEDML and convert the study to a COMBINE archive. The example uses a PhraSEDML repeated task `task1` to run a timecourse simulation `task0` on a model for different values of the parameter `J0_v0`.

```
import tellurium as te

antimony_str = '''
// Created by libAntimony v2.9
model *parameterScan1D()

// Compartments and Species:
compartment compartment_;
species S1 in compartment_, S2 in compartment_, $X0 in compartment_, $X1 in_
↪compartment_;
species $X2 in compartment_;

// Reactions:
J0: $X0 => S1; J0_v0;
J1: S1 => $X1; J1_k3*S1;
J2: S1 => S2; (J2_k1*S1 - J2_k1*S2)*(1 + J2_c*S2^J2_q);
J3: S2 => $X2; J3_k2*S2;

// Species initializations:
```

(continues on next page)

(continued from previous page)

```
S1 = 0;
S2 = 1;
X0 = 1;
X1 = 0;
X2 = 0;

// Compartment initializations:
compartment_ = 1;

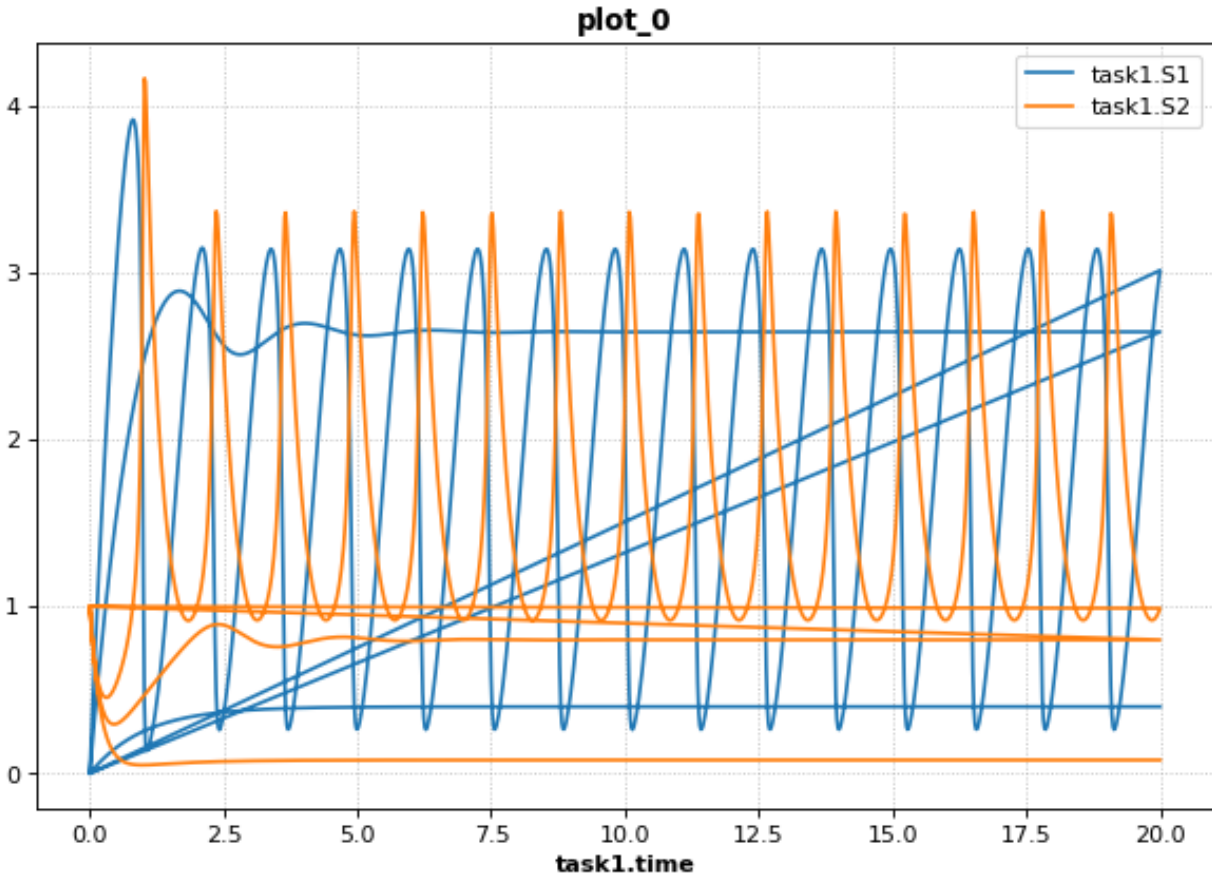
// Variable initializations:
J0_v0 = 8;
J1_k3 = 0;
J2_k1 = 1;
J2_k_1 = 0;
J2_c = 1;
J2_q = 3;
J3_k2 = 5;

// Other declarations:
const compartment_, J0_v0, J1_k3, J2_k1, J2_k_1, J2_c, J2_q, J3_k2;
end
'''

phrasedml_str = '''
modell = model "parameterScan1D"
timecourse1 = simulate uniform(0, 20, 1000)
task0 = run timecourse1 on modell
task1 = repeat task0 for J0_v0 in [8, 4, 0.4], reset=true
plot task1.time vs task1.S1, task1.S2
'''

# create the inline OMEX string
inline_omex = '\n'.join([antimony_str, phrasedml_str])

# execute the inline OMEX
te.executeInlineOmex(inline_omex)
```

4.5.4 2d Parameter Scan

There are multiple ways to specify the set of values that should be swept over. This example uses two repeated tasks instead of one. It sweeps through a discrete set of values for the parameter `J1_KK2`, and then sweeps through a uniform range for another parameter `J4_KK5`.

```
import tellurium as te

antimony_str = '''
// Created by libAntimony v2.9
model *parameterScan2D()

// Compartments and Species:
compartment compartment_;
species MKKK in compartment_, MKKK_P in compartment_, MKK in compartment_;
species MKK_P in compartment_, MKK_PP in compartment_, MAPK in compartment_;
species MAPK_P in compartment_, MAPK_PP in compartment_;

// Reactions:
J0: MKKK => MKKK_P; (J0_V1*MKKK)/((1 + (MAPK_PP/J0_Ki)^J0_n)*(J0_K1 + MKKK));
J1: MKKK_P => MKKK; (J1_V2*MKKK_P)/(J1_KK2 + MKKK_P);
J2: MKK => MKK_P; (J2_k3*MKKK_P*MKK)/(J2_KK3 + MKK);
J3: MKK_P => MKK_PP; (J3_k4*MKKK_P*MKK_P)/(J3_KK4 + MKK_P);
J4: MKK_PP => MKK_P; (J4_V5*MKK_PP)/(J4_KK5 + MKK_PP);
J5: MKK_P => MKK; (J5_V6*MKK_P)/(J5_KK6 + MKK_P);
```

(continues on next page)

(continued from previous page)

```

J6: MAPK => MAPK_P; (J6_k7*MKK_PP*MAPK)/(J6_KK7 + MAPK);
J7: MAPK_P => MAPK_PP; (J7_k8*MKK_PP*MAPK_P)/(J7_KK8 + MAPK_P);
J8: MAPK_PP => MAPK_P; (J8_V9*MAPK_PP)/(J8_KK9 + MAPK_PP);
J9: MAPK_P => MAPK; (J9_V10*MAPK_P)/(J9_KK10 + MAPK_P);

// Species initializations:
MKKK = 90;
MKKK_P = 10;
MKK = 280;
MKK_P = 10;
MKK_PP = 10;
MAPK = 280;
MAPK_P = 10;
MAPK_PP = 10;

// Compartment initializations:
compartment_ = 1;

// Variable initializations:
J0_V1 = 2.5;
J0_Ki = 9;
J0_n = 1;
J0_K1 = 10;
J1_V2 = 0.25;
J1_KK2 = 8;
J2_k3 = 0.025;
J2_KK3 = 15;
J3_k4 = 0.025;
J3_KK4 = 15;
J4_V5 = 0.75;
J4_KK5 = 15;
J5_V6 = 0.75;
J5_KK6 = 15;
J6_k7 = 0.025;
J6_KK7 = 15;
J7_k8 = 0.025;
J7_KK8 = 15;
J8_V9 = 0.5;
J8_KK9 = 15;
J9_V10 = 0.5;
J9_KK10 = 15;

// Other declarations:
const compartment_, J0_V1, J0_Ki, J0_n, J0_K1, J1_V2, J1_KK2, J2_k3, J2_KK3;
const J3_k4, J3_KK4, J4_V5, J4_KK5, J5_V6, J5_KK6, J6_k7, J6_KK7, J7_k8;
const J7_KK8, J8_V9, J8_KK9, J9_V10, J9_KK10;
end
'''

phrasedml_str = '''
model_3 = model "parameterScan2D"
sim_repeat = simulate uniform(0,3000,100)
task_1 = run sim_repeat on model_3
repeatedtask_1 = repeat task_1 for J1_KK2 in [1, 5, 10, 50, 60, 70, 80, 90, 100],
↳reset=true
repeatedtask_2 = repeat repeatedtask_1 for J4_KK5 in uniform(1, 40, 10), reset=true
plot repeatedtask_2.J4_KK5 vs repeatedtask_2.J1_KK2

```

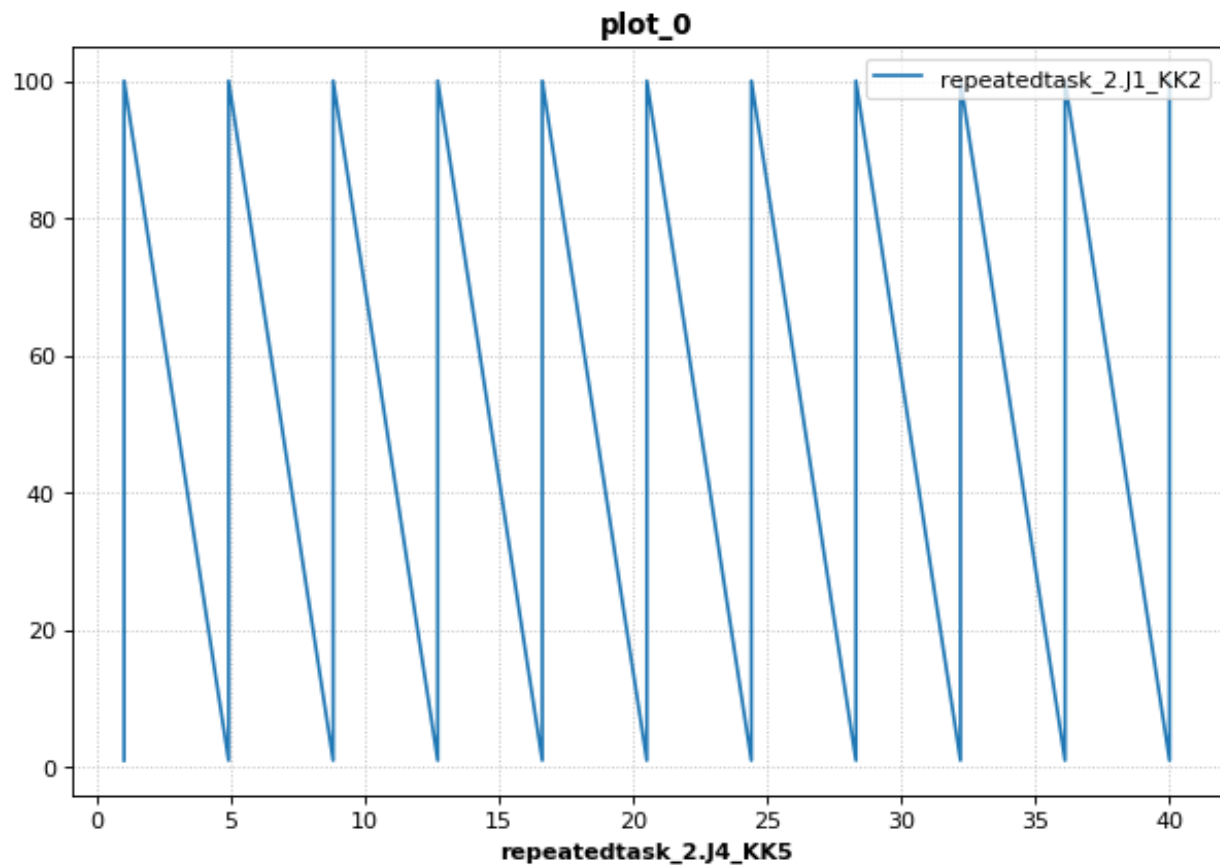
(continues on next page)

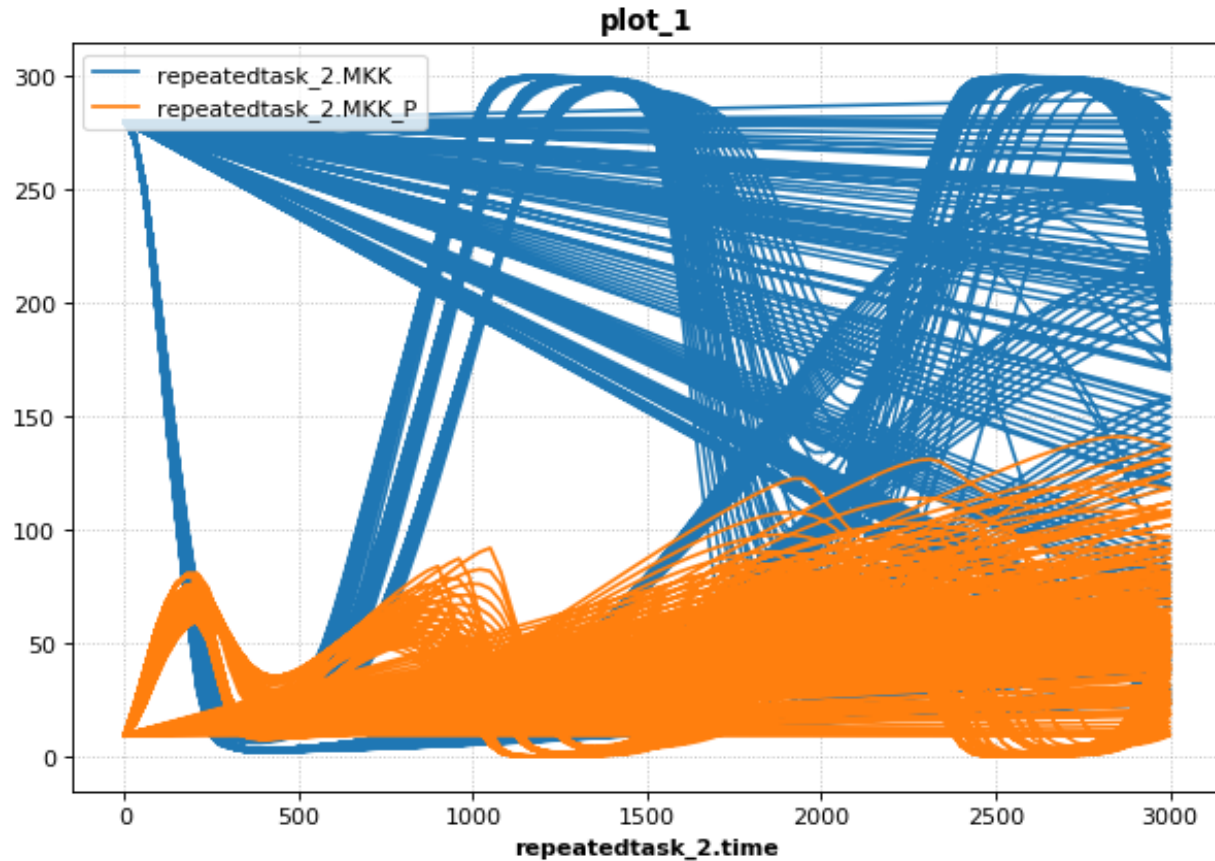
(continued from previous page)

```
plot repeatedtask_2.time vs repeatedtask_2.MKK, repeatedtask_2.MKK_P
'''

# create the inline OMEX string
inline_omex = '\n'.join([antimony_str, phrasedml_str])

# execute the inline OMEX
te.executeInlineOmex(inline_omex)
```





4.5.5 Stochastic Simulation and RNG Seeding

It is possible to programmatically set the RNG seed of a stochastic simulation in PhraSEDML using the `<simulation-name>.algorithm.seed = <value>` directive. Simulations run with the same seed are identical. If the seed is not specified, a different value is used each time, leading to different results.

```
# -*- coding: utf-8 -*-
"""
phrasedml repeated stochastic test
"""
import tellurium as te

antimony_str = '''
// Created by libAntimony v2.9
model *repeatedStochastic()

// Compartments and Species:
compartment compartment_;
species MKKK in compartment_, MKKK_P in compartment_, MKK in compartment_;
species MKK_P in compartment_, MKK_PP in compartment_, MAPK in compartment_;
species MAPK_P in compartment_, MAPK_PP in compartment_;

// Reactions:
J0: MKKK => MKKK_P; (J0_V1*MKKK)/((1 + (MAPK_PP/J0_Ki)^J0_n)*(J0_K1 + MKKK));
J1: MKKK_P => MKKK; (J1_V2*MKKK_P)/(J1_KK2 + MKKK_P);
```

(continues on next page)

(continued from previous page)

```

J2: MKK => MKK_P; (J2_k3*MKKK_P*MKK)/(J2_KK3 + MKK);
J3: MKK_P => MKK_PP; (J3_k4*MKKK_P*MKK_P)/(J3_KK4 + MKK_P);
J4: MKK_PP => MKK_P; (J4_V5*MKK_PP)/(J4_KK5 + MKK_PP);
J5: MKK_P => MKK; (J5_V6*MKK_P)/(J5_KK6 + MKK_P);
J6: MAPK => MAPK_P; (J6_k7*MKK_PP*MAPK)/(J6_KK7 + MAPK);
J7: MAPK_P => MAPK_PP; (J7_k8*MKK_PP*MAPK_P)/(J7_KK8 + MAPK_P);
J8: MAPK_PP => MAPK_P; (J8_V9*MAPK_PP)/(J8_KK9 + MAPK_PP);
J9: MAPK_P => MAPK; (J9_V10*MAPK_P)/(J9_KK10 + MAPK_P);

// Species initializations:
MKKK = 90;
MKKK_P = 10;
MKK = 280;
MKK_P = 10;
MKK_PP = 10;
MAPK = 280;
MAPK_P = 10;
MAPK_PP = 10;

// Compartment initializations:
compartment_ = 1;

// Variable initializations:
J0_V1 = 2.5;
J0_Ki = 9;
J0_n = 1;
J0_K1 = 10;
J1_V2 = 0.25;
J1_KK2 = 8;
J2_k3 = 0.025;
J2_KK3 = 15;
J3_k4 = 0.025;
J3_KK4 = 15;
J4_V5 = 0.75;
J4_KK5 = 15;
J5_V6 = 0.75;
J5_KK6 = 15;
J6_k7 = 0.025;
J6_KK7 = 15;
J7_k8 = 0.025;
J7_KK8 = 15;
J8_V9 = 0.5;
J8_KK9 = 15;
J9_V10 = 0.5;
J9_KK10 = 15;

// Other declarations:
const compartment_, J0_V1, J0_Ki, J0_n, J0_K1, J1_V2, J1_KK2, J2_k3, J2_KK3;
const J3_k4, J3_KK4, J4_V5, J4_KK5, J5_V6, J5_KK6, J6_k7, J6_KK7, J7_k8;
const J7_KK8, J8_V9, J8_KK9, J9_V10, J9_KK10;
end
'''

phrasedml_str = '''
model1 = model "repeatedStochastic"
timecourse1 = simulate.uniform_stochastic(0, 4000, 1000)
timecourse1.algorithm.seed = 1003

```

(continues on next page)

(continued from previous page)

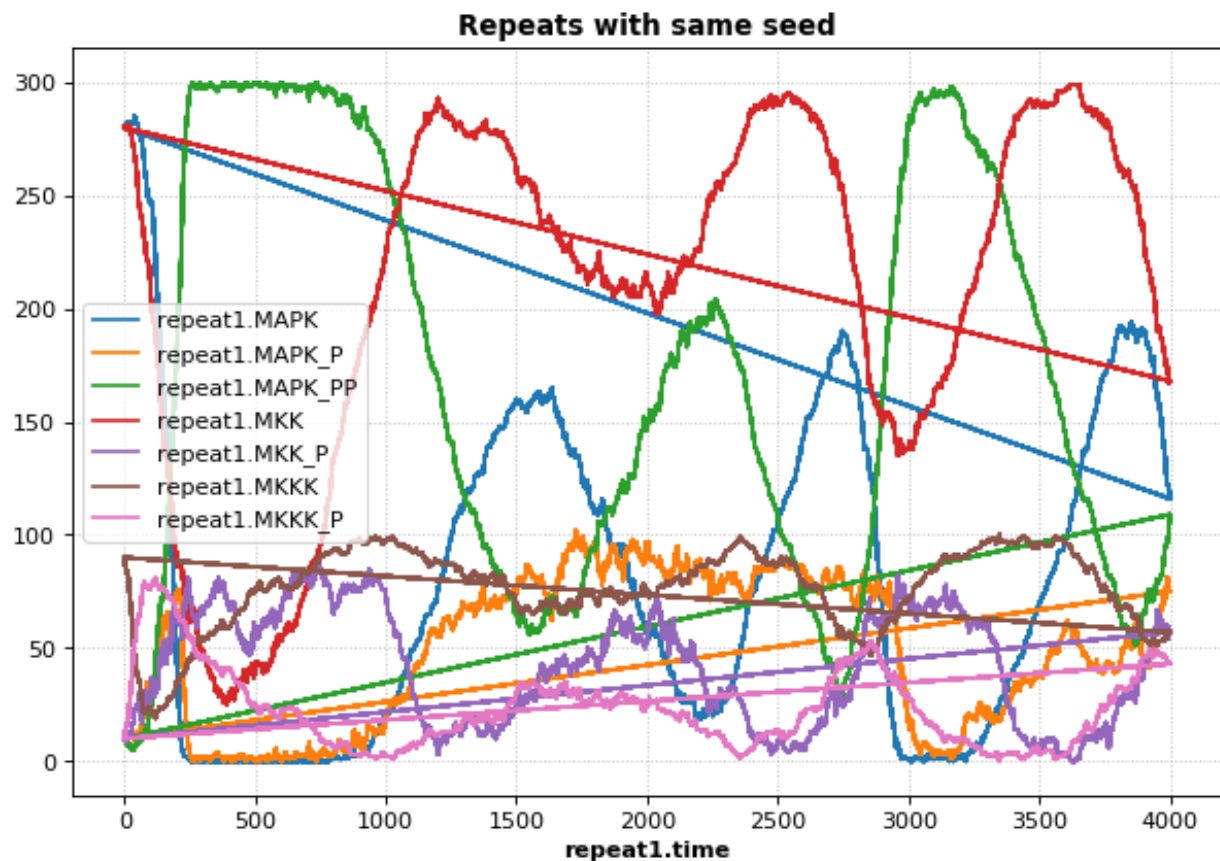
```

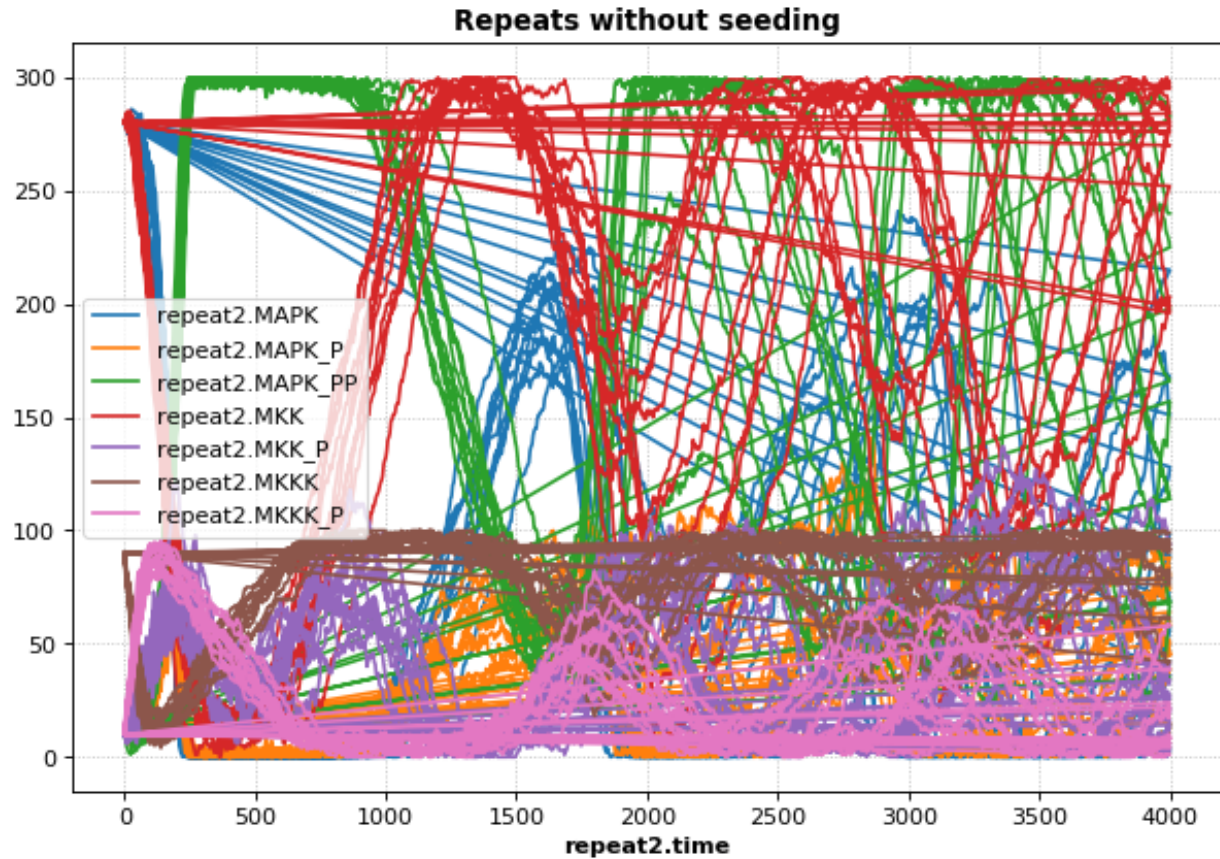
timecourse2 = simulate uniform_stochastic(0, 4000, 1000)
task1 = run timecourse1 on model1
task2 = run timecourse2 on model1
repeat1 = repeat task1 for local.x in uniform(0, 10, 10), reset=true
repeat2 = repeat task2 for local.x in uniform(0, 10, 10), reset=true
plot "Repeats with same seed" repeat1.time vs repeat1.MAPK, repeat1.MAPK_P, repeat1.
↳MAPK_PP, repeat1.MKK, repeat1.MKK_P, repeat1.MKKK, repeat1.MKKK_P
plot "Repeats without seeding" repeat2.time vs repeat2.MAPK, repeat2.MAPK_P, repeat2.
↳MAPK_PP, repeat2.MKK, repeat2.MKK_P, repeat2.MKKK, repeat2.MKKK_P
'''

# create the inline OMEX string
inline_omex = '\n'.join([antimony_str, phrasedml_str])

# execute the inline OMEX
te.executeInlineOmex(inline_omex)

```





4.5.6 Resetting Models

This example is another parameter scan which shows the effect of resetting the model or not after each simulation. When using the repeated task directive in PhraSEDML, you can pass the `reset=true` argument to reset the model to its initial conditions after each repeated simulation. Leaving this argument off causes the model to retain its current state between simulations. In this case, the time value is not reset.

```
import tellurium as te

antimony_str = """
model case_02
    J0: S1 -> S2; k1*S1;
    S1 = 10.0; S2=0.0;
    k1 = 0.1;
end
"""

phrasedml_str = """
model0 = model "case_02"
model1 = model model0 with S1=5.0
sim0 = simulate uniform(0, 6, 100)
task0 = run sim0 on model1
# reset the model after each simulation
task1 = repeat task0 for k1 in uniform(0.0, 5.0, 5), reset = true
# show the effect of not resetting for comparison
```

(continues on next page)

(continued from previous page)

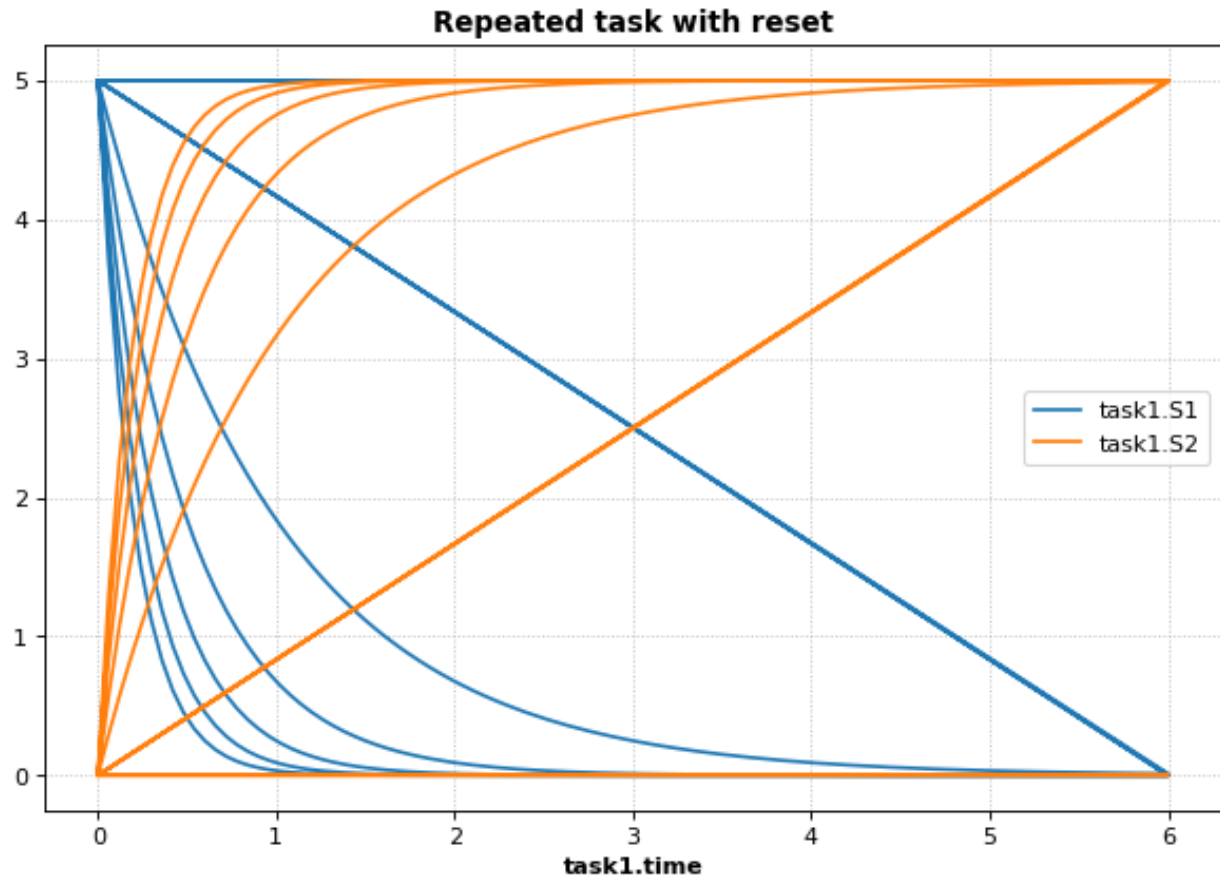
```

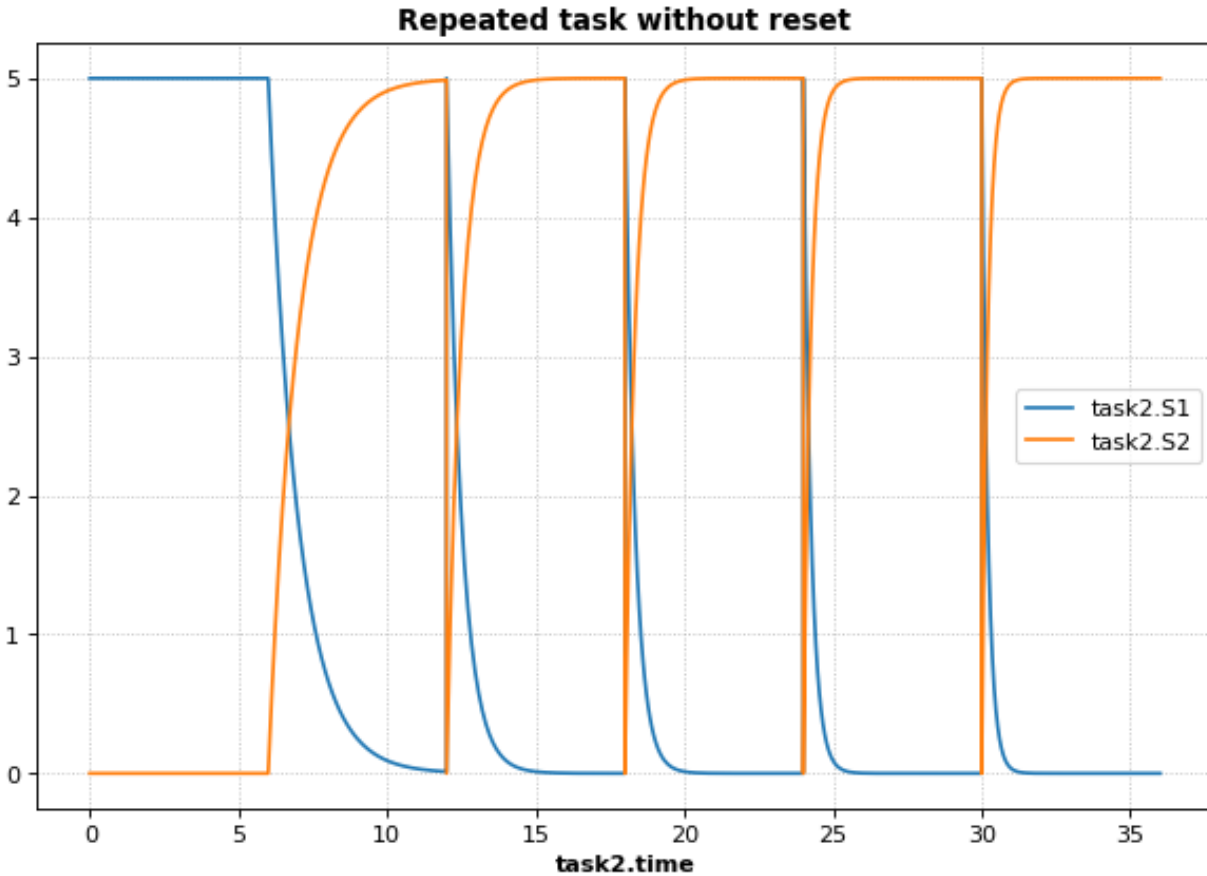
task2 = repeat task0 for k1 in uniform(0.0, 5.0, 5)
plot "Repeated task with reset"    task1.time vs task1.S1, task1.S2
plot "Repeated task without reset" task2.time vs task2.S1, task2.S2
"""

# create the inline OMEX string
inline_omex = '\n'.join([antimony_str, phrasedml_str])

# execute the inline OMEX
te.executeInlineOmex(inline_omex)

```





4.5.7 3d Plotting

This example shows how to use PhraSEDML to perform 3d plotting. The syntax is `plot <x> vs <y> vs <z>`, where `<x>`, `<y>`, and `<z>` are references to model state variables used in specific tasks.

```
import tellurium as te

antimony_str = '''
// Created by libAntimony v2.9
model *case_09()

// Compartments and Species:
compartment compartment_;
species MKKK in compartment_, MKKK_P in compartment_, MKK in compartment_;
species MKK_P in compartment_, MKK_PP in compartment_, MAPK in compartment_;
species MAPK_P in compartment_, MAPK_PP in compartment_;

// Reactions:
J0: MKKK => MKKK_P; (J0_V1*MKKK)/((1 + (MAPK_PP/J0_Ki)^J0_n)*(J0_K1 + MKKK));
J1: MKKK_P => MKKK; (J1_V2*MKKK_P)/(J1_KK2 + MKKK_P);
J2: MKK => MKK_P; (J2_k3*MKKK_P*MKK)/(J2_KK3 + MKK);
J3: MKK_P => MKK_PP; (J3_k4*MKKK_P*MKK_P)/(J3_KK4 + MKK_P);
J4: MKK_PP => MKK_P; (J4_V5*MKK_PP)/(J4_KK5 + MKK_PP);
J5: MKK_P => MKK; (J5_V6*MKK_P)/(J5_KK6 + MKK_P);
J6: MAPK => MAPK_P; (J6_k7*MKK_PP*MAPK)/(J6_KK7 + MAPK);
```

(continues on next page)

(continued from previous page)

```

J7: MAPK_P => MAPK_PP; (J7_k8*MKK_PP*MAPK_P)/(J7_KK8 + MAPK_P);
J8: MAPK_PP => MAPK_P; (J8_V9*MAPK_PP)/(J8_KK9 + MAPK_PP);
J9: MAPK_P => MAPK; (J9_V10*MAPK_P)/(J9_KK10 + MAPK_P);

// Species initializations:
MKKK = 90;
MKKK_P = 10;
MKK = 280;
MKK_P = 10;
MKK_PP = 10;
MAPK = 280;
MAPK_P = 10;
MAPK_PP = 10;

// Compartment initializations:
compartment_ = 1;

// Variable initializations:
J0_V1 = 2.5;
J0_Ki = 9;
J0_n = 1;
J0_K1 = 10;
J1_V2 = 0.25;
J1_KK2 = 8;
J2_k3 = 0.025;
J2_KK3 = 15;
J3_k4 = 0.025;
J3_KK4 = 15;
J4_V5 = 0.75;
J4_KK5 = 15;
J5_V6 = 0.75;
J5_KK6 = 15;
J6_k7 = 0.025;
J6_KK7 = 15;
J7_k8 = 0.025;
J7_KK8 = 15;
J8_V9 = 0.5;
J8_KK9 = 15;
J9_V10 = 0.5;
J9_KK10 = 15;

// Other declarations:
const compartment_, J0_V1, J0_Ki, J0_n, J0_K1, J1_V2, J1_KK2, J2_k3, J2_KK3;
const J3_k4, J3_KK4, J4_V5, J4_KK5, J5_V6, J5_KK6, J6_k7, J6_KK7, J7_k8;
const J7_KK8, J8_V9, J8_KK9, J9_V10, J9_KK10;
end
'''

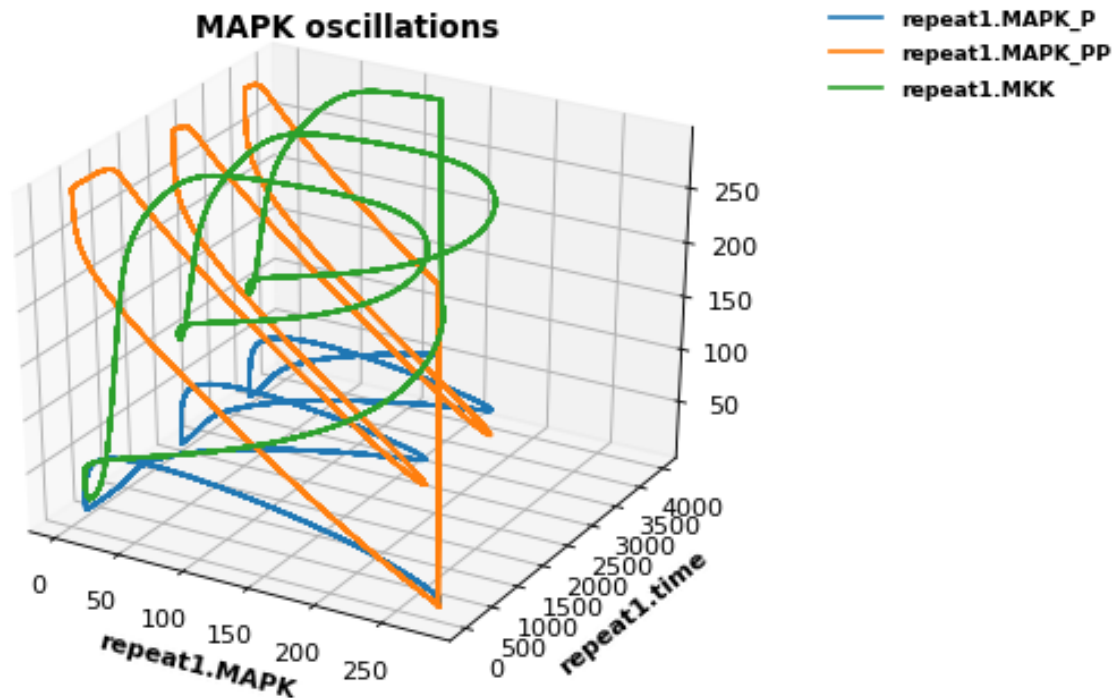
phrasedml_str = '''
modl = model "case_09"
# sim1 = simulate uniform_stochastic(0, 4000, 1000)
sim1 = simulate uniform(0, 4000, 1000)
task1 = run sim1 on modl
repeat1 = repeat task1 for local.x in uniform(0, 10, 10), reset=true
plot "MAPK oscillations" repeat1.MAPK vs repeat1.time vs repeat1.MAPK_P, repeat1.
↪MAPK vs repeat1.time vs repeat1.MAPK_PP, repeat1.MAPK vs repeat1.time vs repeat1.MKK
# report repeat1.MAPK vs repeat1.time vs repeat1.MAPK_P, repeat1.MAPK vs repeat1.
↪time vs repeat1.MAPK_PP, repeat1.MAPK vs repeat1.time vs repeat1.MKK (continues on next page)

```

(continued from previous page)

```
'''
# create the inline OMEX string
inline_omex = '\n'.join([antimony_str, phrasedml_str])

# execute the inline OMEX
te.executeInlineOmex(inline_omex)
```



4.6 Modeling Case Studies

This series of case studies shows some slight more advanced examples which correspond to common motifs in biological networks (negative feedback loops, etc.). To draw the network diagrams seen here, you will need [graphviz](#) installed.

4.6.1 Preliminaries

In order to draw the network graphs in these examples (i.e. using `r.draw()`), you will need [graphviz](#) and [pygraphviz](#) installed. Please consult the [Graphviz](#) documentation for instructions on installing it on your platform. If you cannot install [Graphviz](#) and [pygraphviz](#), you can still run the following examples, but the network diagrams will not be generated.

Also, due to limitations in [pygraphviz](#), these examples can only be run in the Jupyter notebook, not the [Tellurium notebook app](#).

Install pygraphviz (requires compilation)

Please run

```
<your-local-python-executable> -m pip install pygraphviz
```

from a terminal or command prompt to install pygraphviz. Then restart your kernel in this notebook (Language->Restart Running Kernel).

Troubleshooting Graphviz Installation

pygraphviz has [known problems](#) during installation on some platforms. On 64-bit Fedora Linux, we have been able to use the following command to install pygraphviz:

```
/path/to/python3 -m pip install pygraphviz --install-option="--include-path=/usr/  
↪include/graphviz" --install-option="--library-path=/usr/lib64/graphviz/"
```

You may need to modify the library/include paths in the above command. Some Linux distributions put 64-bit libraries in /usr/lib instead of /usr/lib64, in which case the command becomes:

```
/path/to/python3 -m pip install pygraphviz --install-option="--include-path=/usr/  
↪include/graphviz" --install-option="--library-path=/usr/lib/graphviz/"
```

Case Studies

4.6.2 Activator system

```
import warnings  
warnings.filterwarnings("ignore")  
  
import tellurium as te  
te.setDefaultPlottingEngine('matplotlib')  
  
# model Definition  
r = te.loada('''  
    #J1: S1 -> S2; Activator*kcat1*S1/(Km1+S1);  
    J1: S1 -> S2; SE2*kcat1*S1/(Km1+S1);  
    J2: S2 -> S1; Vm2*S2/(Km2+S2);  
  
    J3: T1 -> T2; S2*kcat3*T1/(Km3+T1);  
    J4: T2 -> T1; Vm4*T2/(Km4+T2);  
  
    J5:      -> E2; Vf5/(Ks5+T2^h5);  
    J6:      -> E3; Vf6*T2^h6/(Ks6+T2^h6);  
  
    #J7:      -> E1;  
    J8:      -> S; kcat8*E1  
  
    J9: E2 ->      ; k9*E2;  
    J10:E3 ->      ; k10*E3;  
  
    J11: S -> SE2; E2*kcat11*S/(Km11+S);  
    J12: S -> SE3; E3*kcat12*S/(Km12+S);
```

(continues on next page)

(continued from previous page)

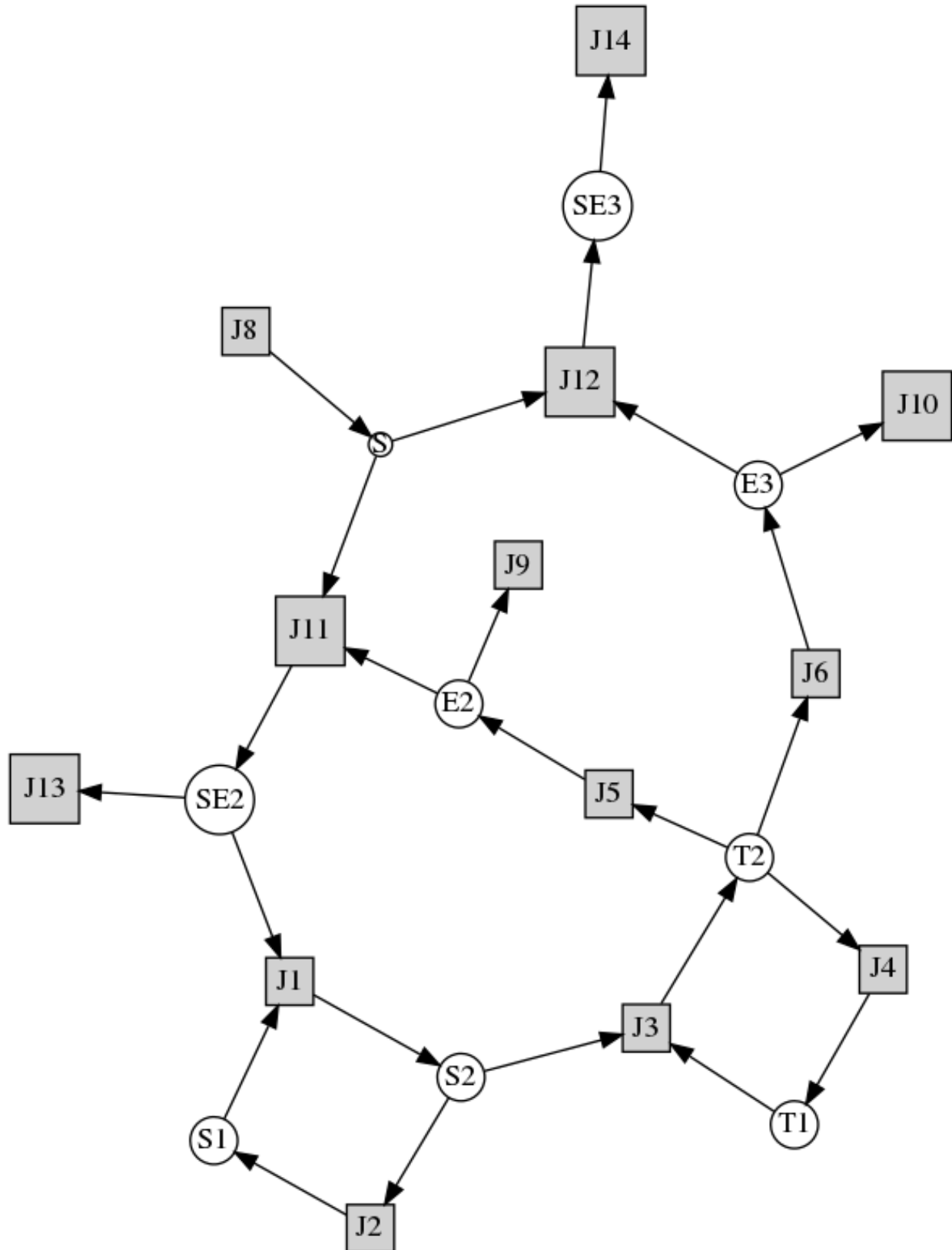
```

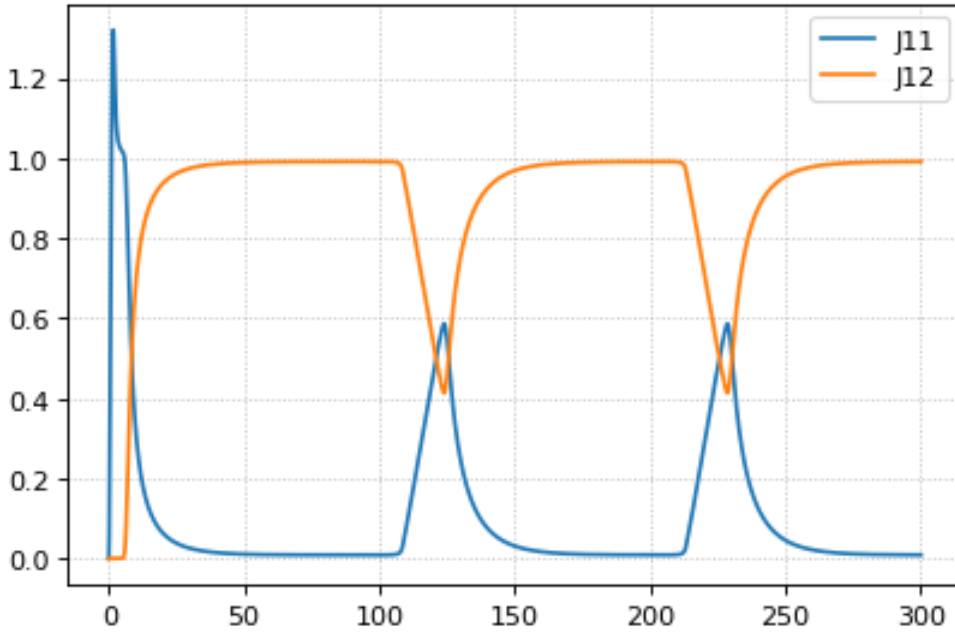
J13: SE2 -> ; SE2*kcat13;
J14: SE3 -> ; SE3*kcat14;

Km1 = 0.01; Km2 = 0.01; Km3 = 0.01; Km4 = 0.01; Km11 = 1; Km12 = 0.1;
S1 = 6; S2 =0.1; T1=6; T2 = 0.1;
SE2 = 0; SE3=0;
S=0;
E2 = 0; E3 = 0;
kcat1 = 0.1; kcat3 = 3; kcat8 =1; kcat11 = 1; kcat12 = 1; kcat13 = 0.1;
↪kcat14=0.1;
E1 = 1;
k9 = 0.1; k10=0.1;
Vf6 = 1;
Vf5 = 3;
Vm2 = 0.1;
Vm4 = 2;
h6 = 2; h5=2;
Ks6 = 1; Ks5 = 1;
Activator = 0;

    at (time > 100): Activator = 5;
'''
r.draw(width=300)
result = r.simulate (0, 300, 2000, ['time', 'J11', 'J12']);
r.plot(result);

```





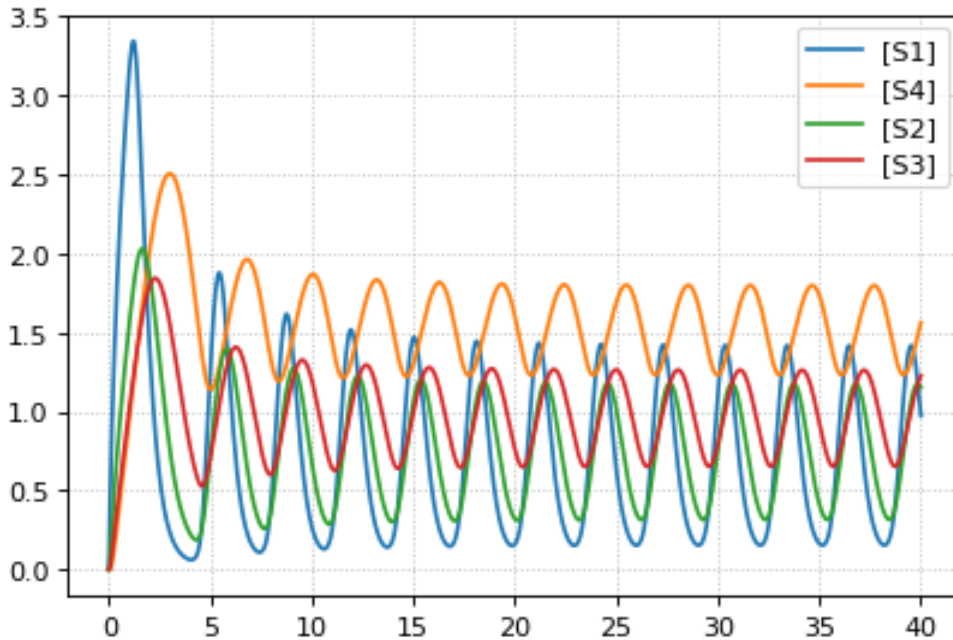
4.6.3 Feedback oscillations

```
# http://tellurium.analogmachine.org/testing/
import tellurium as te
r = te.loada ('''
model feedback()
    // Reactions:
    J0: $X0 -> S1; (VM1 * (X0 - S1/Keq1))/(1 + X0 + S1 + S4^h);
    J1: S1 -> S2; (10 * S1 - 2 * S2) / (1 + S1 + S2);
    J2: S2 -> S3; (10 * S2 - 2 * S3) / (1 + S2 + S3);
    J3: S3 -> S4; (10 * S3 - 2 * S4) / (1 + S3 + S4);
    J4: S4 -> $X1; (V4 * S4) / (KS4 + S4);

    // Species initializations:
    S1 = 0; S2 = 0; S3 = 0;
    S4 = 0; X0 = 10; X1 = 0;

    // Variable initialization:
    VM1 = 10; Keq1 = 10; h = 10; V4 = 2.5; KS4 = 0.5;
end''')

r.integrator.setValue('variable_step_size', True)
res = r.simulate(0, 40)
r.plot()
```



4.6.4 Bistable System

Example showing how to multiple time course simulations, merging the data and plotting it onto one plotting surface. Alternative is to use `setHold()`

Model is a bistable system, simulations start with different initial conditions resulting in different steady states reached.

```
import tellurium as te
import numpy as np

r = te.loada ('''
$Xo -> S1; 1 + Xo*(32+(S1/0.75)^3.2)/(1 + (S1/4.3)^3.2);
S1 -> $X1; k1*S1;

Xo = 0.09; X1 = 0.0;
S1 = 0.5; k1 = 3.2;
''')
print(r.selections)

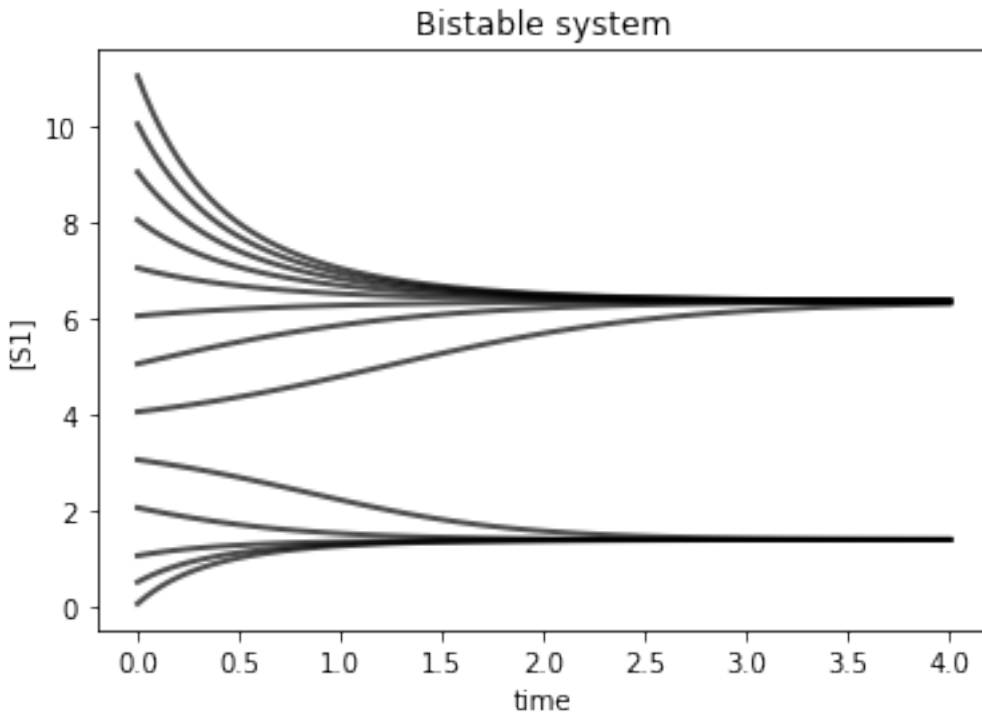
initValue = 0.05
m = r.simulate (0, 4, 100, selections=["time", "S1"])

for i in range (0,12):
    r.reset()
    r['[S1]'] = initValue
    res = r.simulate (0, 4, 100, selections=["S1"])
    m = np.concatenate([m, res], axis=1)
    initValue += 1

te.plotArray(m, color="black", alpha=0.7, loc=None,
             xlabel="time", ylabel="[S1]", title="Bistable system");
```



```
['time', '[S1]']
```



4.6.5 Events

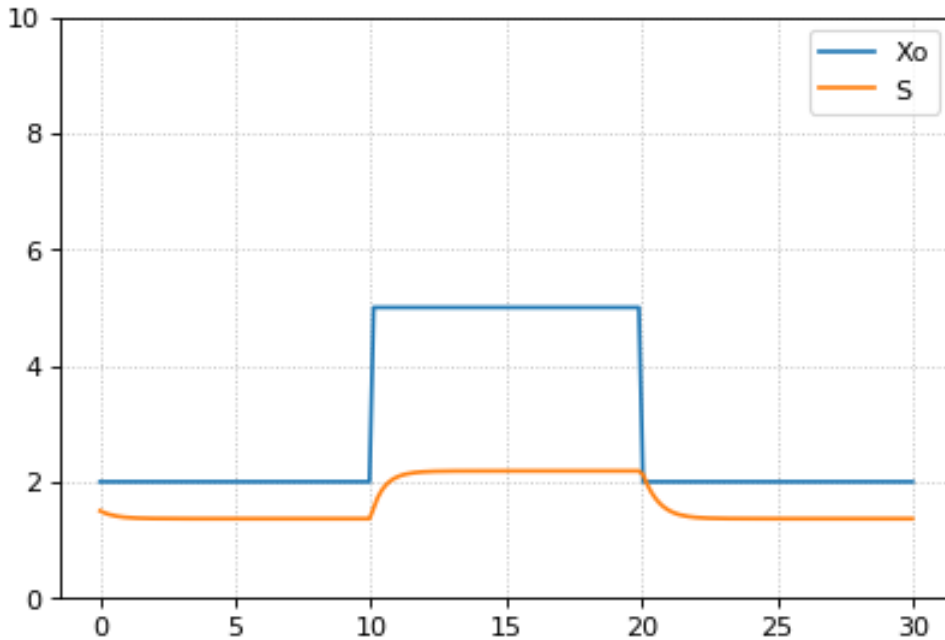
```
import tellurium as te
import matplotlib.pyplot as plt

# Example showing use of events and how to set the y axis limits
r = te.loada('''
    $Xo -> S;    Xo/(km + S^h);
    S -> $w;    k1*S;

    # initialize
    h = 1;    # Hill coefficient
    k1 = 1;    km = 0.1;
    S = 1.5; Xo = 2

    at (time > 10): Xo = 5;
    at (time > 20): Xo = 2;
''')

m1 = r.simulate(0, 30, 200, ['time', 'Xo', 'S'])
r.plot(ylim=(0,10))
```



4.6.6 Gene network

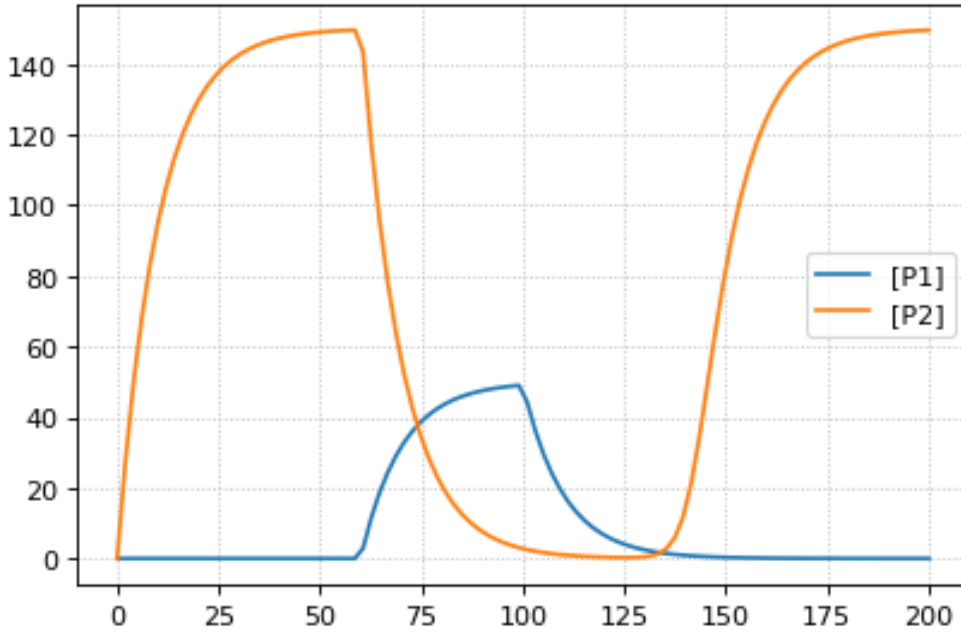
```
import tellurium as te
import numpy

# Model describes a cascade of two genes. First gene is activated
# second gene is repressed. Uses events to change the input
# to the gene regulatory network

r = te.loada('''
    v1: -> P1; Vm1*I^4/(Km1 + I^4);
    v2: P1 -> ; k1*P1;
    v3: -> P2; Vm2/(Km2 + P1^4);
    v4: P2 -> ; k2*P2;

    at (time > 60): I = 10;
    at (time > 100): I = 0.01;
    Vm1 = 5; Vm2 = 6; Km1 = 0.5; Km2 = 0.4;
    k1 = 0.1; k2 = 0.1;
    I = 0.01;
''')

result = r.simulate(0, 200, 100)
r.plot()
```



4.6.7 Stoichiometric matrix

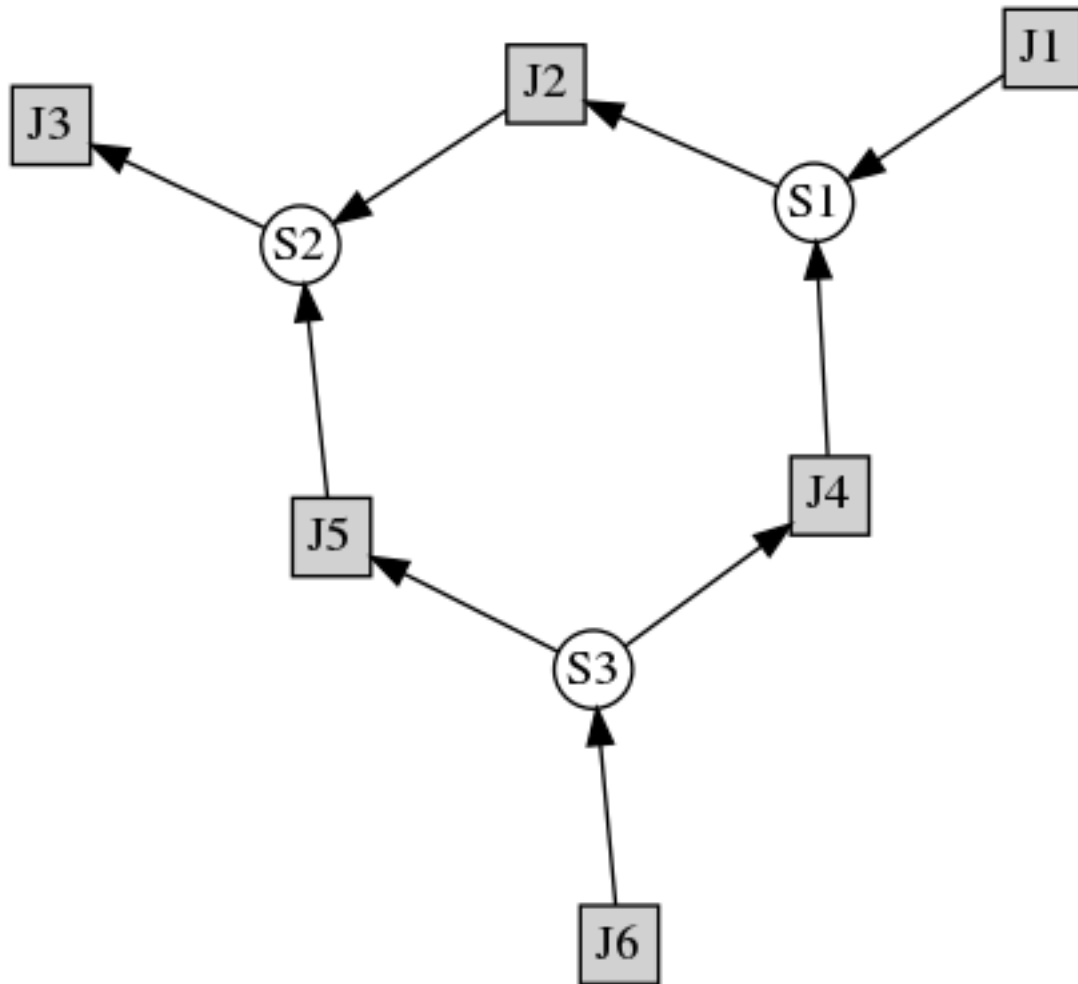
```
import tellurium as te

# Example of using antimony to create a stoichiometry matrix
r = te.loada('''
J1: -> S1; v1;
J2: S1 -> S2; v2;
J3: S2 -> ; v3;
J4: S3 -> S1; v4;
J5: S3 -> S2; v5;
J6: -> S3; v6;

v1=1; v2=1; v3=1; v4=1; v5=1; v6=1;
''')

print(r.getFullStoichiometryMatrix())
r.draw()
```

```
      J1, J2, J3, J4, J5, J6
S1 [[ 1, -1,  0,  1,  0,  0],
S2 [  0,  1, -1,  0,  1,  0],
S3 [  0,  0,  0, -1, -1,  1]]
```



4.6.8 Lorenz attractor

Example showing how to describe a model using ODES. Example implements the Lorenz attractor.

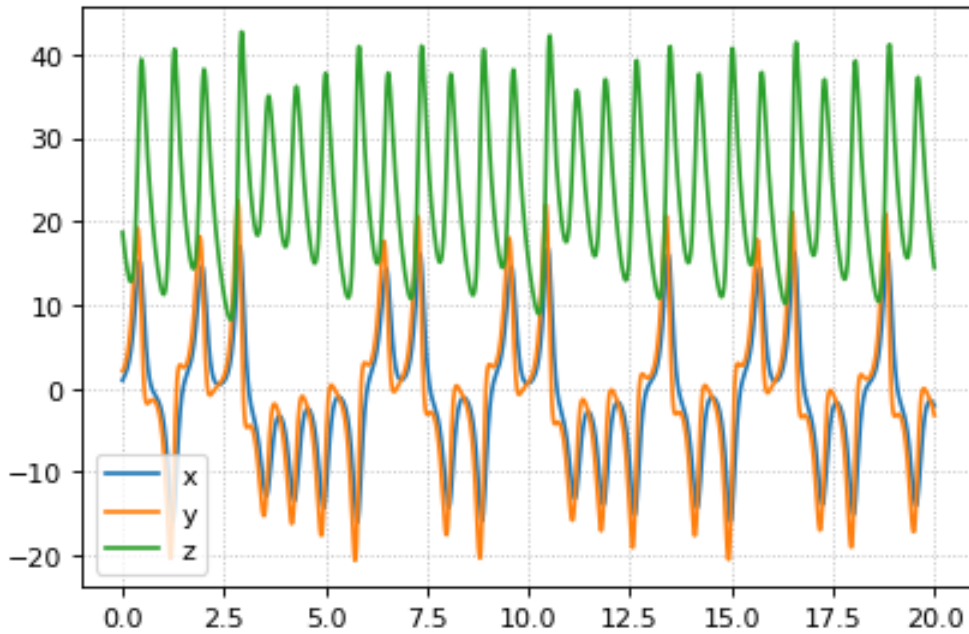
```
import tellurium as te

r = te.loada ('''
    x' = sigma*(y - x);
    y' = x*(rho - z) - y;
    z' = x*y - beta*z;

    x = 0.96259;  y = 2.07272;  z = 18.65888;

    sigma = 10;  rho = 28; beta = 2.67;
''')

result = r.simulate (0, 20, 1000, ['time', 'x', 'y', 'z'])
r.plot()
```



4.6.9 Time Course Parameter Scan

Do 5 simulations on a simple model, for each simulation a parameter, k_1 is changed. The script merges the data together and plots the merged array on to one plot.

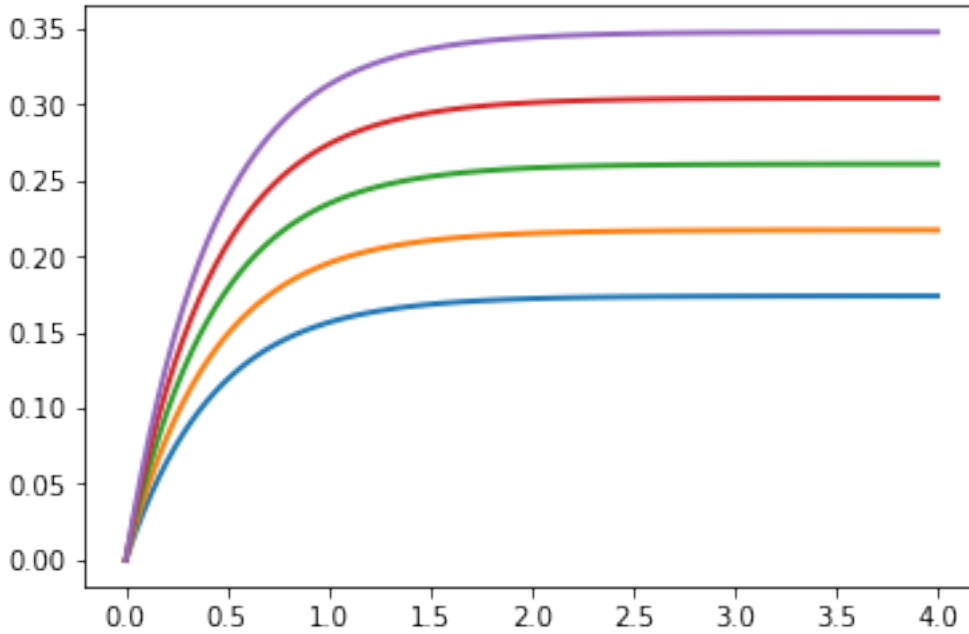
```
import tellurium as te
import numpy as np

r = te.loada (''
    J1: $X0 -> S1; k1*X0;
    J2: S1 -> $X1; k2*S1;

    X0 = 1.0; S1 = 0.0; X1 = 0.0;
    k1 = 0.4; k2 = 2.3;
'')

m = r.simulate (0, 4, 100, ["Time", "S1"])
for i in range (0,4):
    r.k1 = r.k1 + 0.1
    r.reset()
    m = np.hstack([m, r.simulate(0, 4, 100, ['S1'])])

# use plotArray to plot merged data
te.plotArray(m)
pass
```



4.6.10 Merge multiple simulations

Example of merging multiple simulations. In between simulations a parameter is changed.

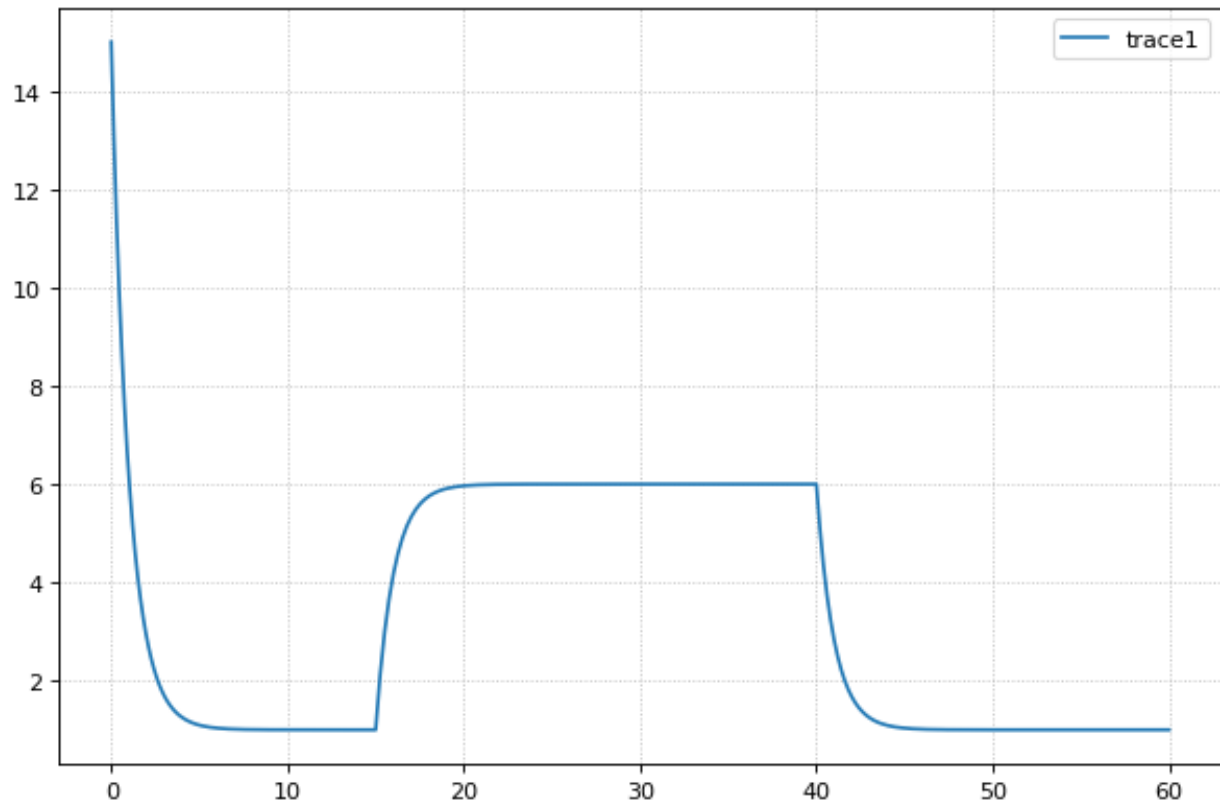
```
import tellurium as te
import numpy

r = te.loada('''
# Model Definition
v1: $Xo -> S1; k1*Xo;
v2: S1 -> $w; k2*S1;

# Initialize constants
k1 = 1; k2 = 1; S1 = 15; Xo = 1;
''')

# Time course simulation
m1 = r.simulate(0, 15, 100, ["Time", "S1"]);
r.k1 = r.k1 * 6;
m2 = r.simulate(15, 40, 100, ["Time", "S1"]);
r.k1 = r.k1 / 6;
m3 = r.simulate(40, 60, 100, ["Time", "S1"]);

m = numpy.vstack([m1, m2, m3])
p = te.plot(m[:,0], m[:,1], name='trace1')
```



4.6.11 Relaxation oscillator

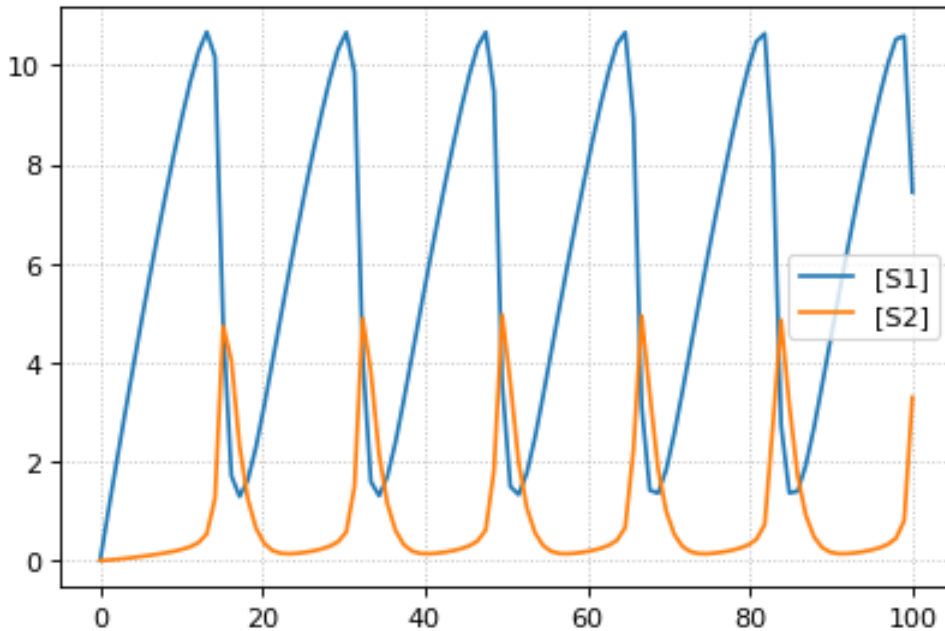
Oscillator that uses positive and negative feedback. An example of a relaxation oscillator.

```
import tellurium as te

r = te.loada('''
v1: $Xo -> S1; k1*Xo;
v2: S1 -> S2; k2*S1*S2^h/(10 + S2^h) + k3*S1;
v3: S2 -> $w; k4*S2;

# Initialize
h = 2; # Hill coefficient
k1 = 1; k2 = 2; Xo = 1;
k3 = 0.02; k4 = 1;
''')

result = r.simulate(0, 100, 100)
r.plot(result);
```



4.6.12 Scan hill coefficient

Negative Feedback model where we scan over the value of the Hill coefficient.

```
import tellurium as te
import numpy as np

r = te.loada('''
// Reactions:
J0: $X0 => S1; (J0_VM1*(X0 - S1/J0_Keq1))/(1 + X0 + S1 + S4^J0_h);
J1: S1 => S2; (10*S1 - 2*S2)/(1 + S1 + S2);
J2: S2 => S3; (10*S2 - 2*S3)/(1 + S2 + S3);
J3: S3 => S4; (10*S3 - 2*S4)/(1 + S3 + S4);
J4: S4 => $X1; (J4_V4*S4)/(J4_KS4 + S4);

// Species initializations:
S1 = 0;
S2 = 0;
S3 = 0;
S4 = 0;
X0 = 10;
X1 = 0;

// Variable initializations:
J0_VM1 = 10;
J0_Keq1 = 10;
J0_h = 2;
J4_V4 = 2.5;
J4_KS4 = 0.5;

// Other declarations:
const J0_VM1, J0_Keq1, J0_h, J4_V4, J4_KS4;
''')
```

(continues on next page)

(continued from previous page)

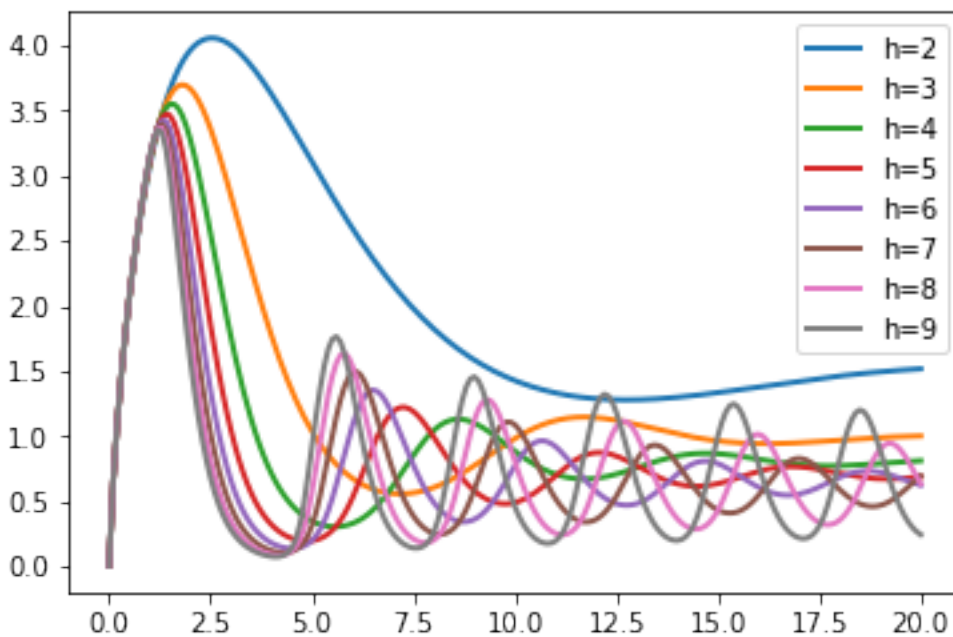
```

# time vector
result = r.simulate (0, 20, 201, ['time'])

h_values = [r.J0_h + k for k in range(0,8)]
for h in h_values:
    r.reset()
    r.J0_h = h
    m = r.simulate(0, 20, 201, ['S1'])
    result = numpy.hstack([result, m])

te.plotArray(result, labels=['h={}'.format(int(h)) for h in h_values])
pass

```



4.6.13 Compare simulations

```

import tellurium as te

r = te.loada ('''
    v1: $Xo -> S1;   k1*Xo;
    v2: S1 -> $w;    k2*S1;

    //initialize. Deterministic process.
    k1 = 1; k2 = 1; S1 = 20; Xo = 1;
''')

m1 = r.simulate (0,20,100);

# Stochastic process
r.resetToOrigin()
r.setSeed(1234)

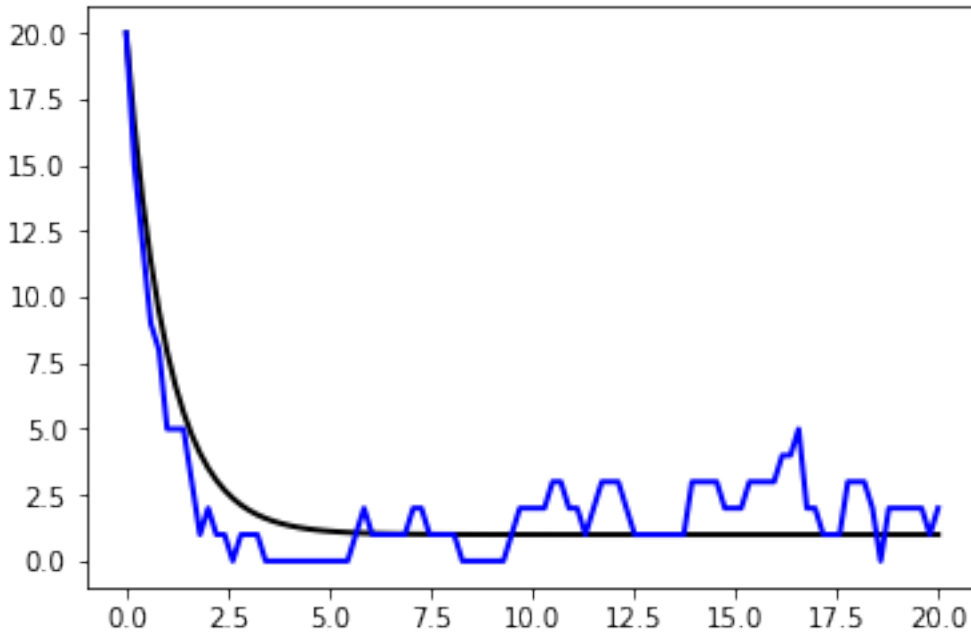
```

(continues on next page)

(continued from previous page)

```
m2 = r.gillespie(0, 20, 100, ['time', 'S1'])

# plot all the results together
te.plotArray(m1, color="black", show=False)
te.plotArray(m2, color="blue");
```



4.6.14 Sinus injection

Example that show how to inject a sinusoidal into the model and use events to switch it off and on.

```
import tellurium as te
import numpy

r = te.loada ('''
# Inject sin wave into model
Xo := sin (time*0.5)*switch + 2;

# Model Definition
v1: $Xo -> S1; k1*Xo;
v2: S1 -> S2; k2*S1;
v3: S2 -> $X1; k3*S2;

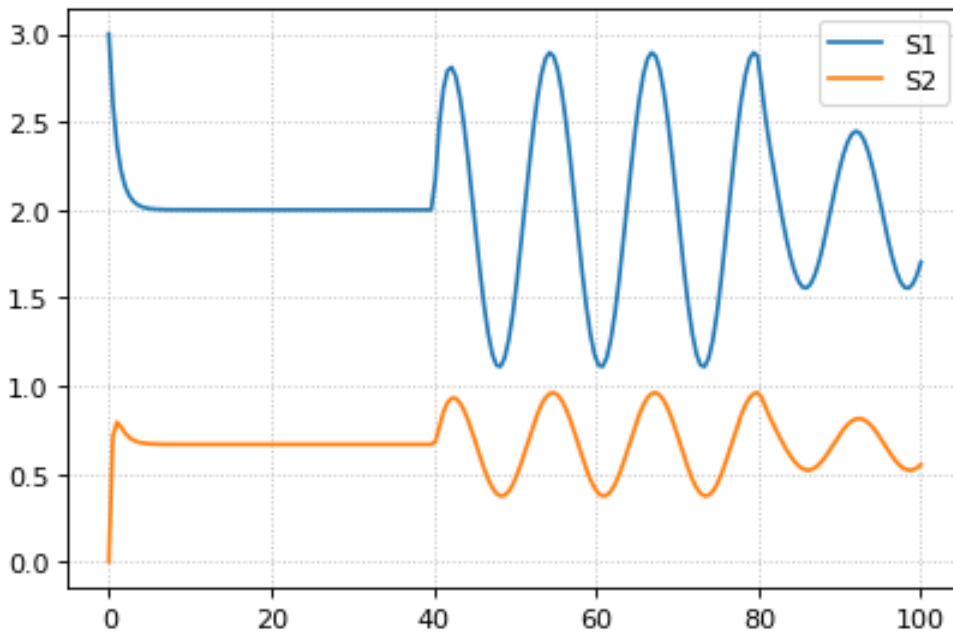
at (time > 40): switch = 1;
at (time > 80): switch = 0.5;

# Initialize constants
k1 = 1; k2 = 1; k3 = 3; S1 = 3;
S2 = 0;
switch = 0;
''')
```

(continues on next page)

(continued from previous page)

```
result = r.simulate (0, 100, 200, ['time', 'S1', 'S2'])
r.plot(result);
```



4.6.15 Protein phosphorylation cycle

Simple protein phosphorylation cycle. Steady state concentration of the phosphorylated protein is plotted as a function of the cycle kinase. In addition, the plot is repeated for various values of K_m .

```
import tellurium as te
import numpy as np

r = te.loada ('''
    S1 -> S2; k1*S1/(Km1 + S1);
    S2 -> S1; k2*S2/(Km2 + S2);

    k1 = 0.1; k2 = 0.4; S1 = 10; S2 = 0;
    Km1 = 0.1; Km2 = 0.1;
''')

for i in range (1,8):
    numbers = np.linspace (0, 1.2, 200)
    result = np.empty ([0,2])
    for value in numbers:
        r.k1 = value
        r.steadyState()
        row = np.array ([value, r.S2])
        result = np.vstack ((result, row))
    te.plotArray(result, show=False, labels=['Km1={}'.format(r.Km1)],
                  resetColorCycle=False,
                  xlabel='k1', ylabel="S2",
                  title="Steady State S2 for different Km1 & Km2",
```

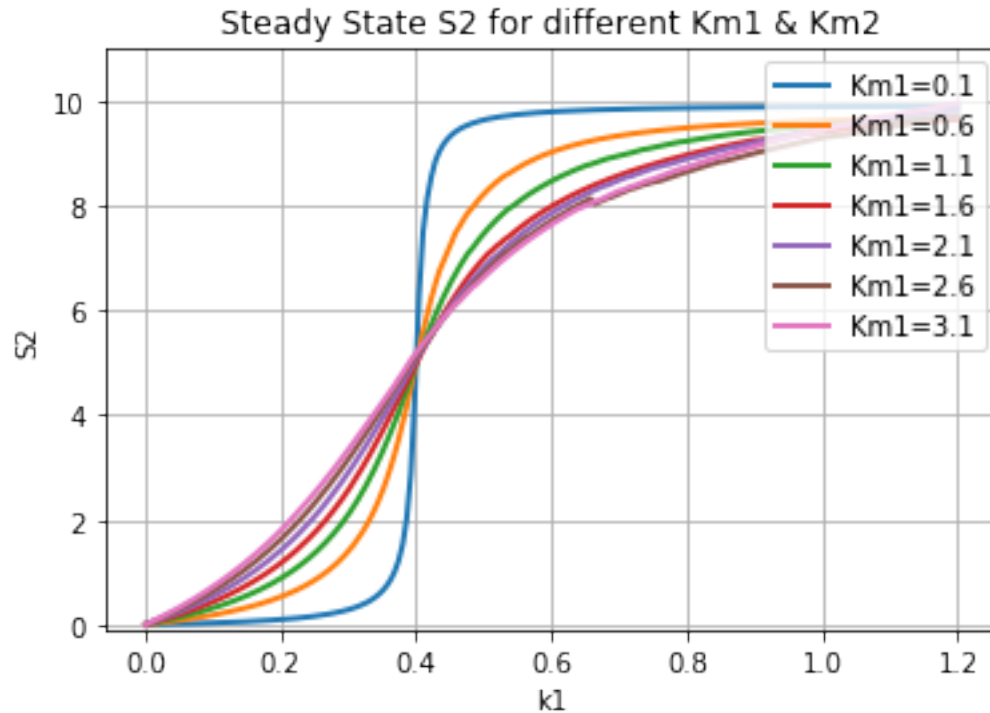
(continues on next page)

(continued from previous page)

```

        ylim=[-0.1, 11], grid=True)
r.k1 = 0.1
r.Km1 = r.Km1 + 0.5;
r.Km2 = r.Km2 + 0.5;

```



4.7 Miscellaneous

4.7.1 How to install additional packages

If you are using Tellurium notebook or Tellurium Spyder, you can install additional package using `installPackage` function. In Tellurium Spyder, you can also install packages using included command Prompt. For more information, see [Running Command Prompt for Tellurium Spyder](#).

```

import tellurium as te
# install cobra (https://github.com/opencobra/cobrapy)
te.installPackage('cobra')
# update cobra to latest version
te.upgradePackage('cobra')
# remove cobra
# te.removePackage('cobra')

```

Antimony Reference

Different authoring tools have different ways of allowing the user to build models, and these approaches have individual advantages and disadvantages. In Tellurium, the main approach to building models is to use a human-readable, text-based definition language called [Antimony](#). Antimony is designed to interconvert between the SBML standard and a shorthand form that allows editing without the structure and overhead of working with XML directly. This guide will show you the intricacies of working with Antimony.

5.1 Background

Since the advent of SBML (the Systems Biology Markup Language) computer models of biological systems have been able to be transferred easily between different labs and different computer programs without loss of specificity. But SBML was not designed to be readable or writable by humans, only by computer programs, so other programs have sprung up to allow users to more easily create the models they need.

Many of these programs are GUI-based, and allow drag-and-drop editing of species and reactions, such as [CellDesigner](#). A few, like Jarnac, take a text-based approach, and allow the creation of models in a text editor. This has the advantage of being usable in an automated setting, such as generating models from a template metalanguage ([TemplateSB](#) is such a metalanguage for Antimony) and readable by others without translation. Antimony (so named because the chemical symbol of the element is ‘Sb’) was designed as a successor to Jarnac’s model definition language, with some new features that mesh with newer elements of SBML, some new features we feel will be generally applicable, and some new features that are designed to aid the creation of genetic networks specifically. Antimony is available as a library and a Python package.

Antimony is the main method of building models in Tellurium. Its main features include:

- Easily define species, reactions, compartments, events, and other elements of a biological model.
- Package and re-use models as modules with defined or implied interfaces.
- Create ‘DNA strand’ elements, which can pass reaction rates to downstream elements, and inherit and modify reaction rates from upstream elements.

5.2 Change Log

In the 2.5 release of Antimony, translation of Antimony concepts to and from the Hierarchical Model Composition package was developed further to be much more robust, and a new test system was added to ensure that Antimony's 'flattening' routine (which exports plain SBML) matches libSBML's flattening routine.

In the 2.4 release of Antimony, use of the Hierarchical Model Composition package constructs in the SBML translation became standard, due to the package being fully accepted by the SBML community.

In the 2.2/2.3 release of Antimony, units, conversion factors, and deletions were added.

In the 2.1 version of Antimony, the 'import' handling became much more robust, and it became additionally possible to export hierarchical models using the Hierarchical Model Composition package constructs for SBML level 3.

In the 2.0 version of Antimony, it became possible to export models as CellML. This requires the use of the CellML API, which is now available as an SDK. Hierarchical models are exported using CellML's hierarchy, translated to accommodate their 'black box' requirements.

5.3 Contents

Contents

- *Antimony Reference*
 - *Background*
 - *Change Log*
 - *Contents*
 - *Introduction & Basics*
 - *Examples*
 - * *Comments*
 - * *Reactions*
 - * *Rate Laws and Initializing Values*
 - * *Boundary Species*
 - * *Compartments*
 - * *Assignments*
 - * *Assignments in Time*
 - * *Piecewise Assignments*
 - * *Events*
 - * *Function Definitions*
 - * *Modular Models*
 - * *Importing Files*
 - * *Units*
 - *Simulating Models*

- *Language Reference*
 - * *Species and Reactions*
 - * *Modules*
 - * *Module conversion factors*
 - * *Submodel deletions*
 - * *Constant and variable symbols*
 - * *Compartments*
 - * *Events*
 - * *Assignment Rules*
 - * *Signals*
 - *Step Input*
 - *Ramp*
 - *Ramp then Stop*
 - *Pulse*
 - *Sinusoidal Input*
 - * *Rate Rules*
 - * *Display Names*
 - * *DNA Strands*
 - * *Interactions*
 - * *Other files*
 - * *Importing and Exporting Antimony Models*
 - * *Appendix: Converting between SBML and Antimony*
 - * *Further Reading*

5.4 Introduction & Basics

Creating a model in Antimony is designed to be very straightforward and simple. Model elements are created and defined in text, with a simple syntax.

The most common way to use Antimony is to create a reaction network, where processes are defined wherein some elements are consumed and other elements are created. Using the language of SBML, the processes are called ‘reactions’ and the elements are called ‘species’, but any set of processes and elements may be modeled in this way. The syntax for defining a reaction in Antimony is to list the species being consumed, separated by a +, followed by an arrow \rightarrow , followed by another list of species being created, followed by a semicolon. If this reaction has a defined mathematical rate at which this happens, that rate can be listed next:

```
S1 -> S2; k1*S1
```

The above model defines a reaction where S1 is converted to S2 at a rate of ‘k1*S1’.

This model cannot be simulated, however, because a simulator would not know what the conditions are to start the

simulation. These values can be set by using an equals sign: cillator:

```
S1 -> S2; k1*S1
S1 = 10
S2 = 0
k1 = 0.1
```

The above, then, is a complete model that can be simulated by any software that understands SBML (to which Antimony models can be converted).

If you want to give your model a name, you can do that by wrapping it with the text: `model [name]` `[reactions, etc.] end`:

```
# Simple UniUni reaction with first-order mass-action kinetics
model example1
  S1 -> S2; k1*S1
  S1 = 10
  S2 = 0
  k1 = 0.1
end
```

In subsequent examples in this tutorial, we'll be using this syntax to name the examples, but for simple models, the name is optional. Later, when we discuss submodels, this will become more important.

There are many more complicated options in Antimony, but the above has enough power to define a wide variety of models, such as this oscillator:

```
model osc1i
  #Reactions:
  J0:    -> S1;  J0_v0
  J1: S1 ->    ;  J1_k3*S1
  J2: S1 -> S2; (J2_k1*S1 - J2_k2*S2)*(1 + J2_c*S2^J2_q)
  J3: S2 ->    ;  J3_k2*S2

  # Species initializations:
  S1 = 0
  S2 = 1

  # Variable initializations:
  J0_v0 = 8
  J1_k3 = 0
  J2_k1 = 1
  J2_k2 = 0
  J2_c  = 1
  J2_q  = 3
  J3_k2 = 5
end
```

5.5 Examples

5.5.1 Comments

Single-line comments in Antimony can be created using the `#` or `//` symbols, and multi-line comments can be created by surrounding them with `/* [comments] */`.


```

/* This is an example of a multi-line
   comment for this tutorial */
model example2
  J0: S1 -> S2 + S3; k1*S1 #Mass-action kinetics
  S1 = 10 #The initial concentration of S1
  S2 = 0 #The initial concentration of S2
  S3 = 3 #The initial concentration of S3
  k1 = 0.1 #The value of the kinetic parameter from J0.
end

```

The names of the reaction and the model are saved in SBML, but any comments are not.

5.5.2 Reactions

Reactions can be created with multiple reactants and/or products, and the stoichiometries can be set by adding a number before the name of the species:

```

# Production of S1
  -> S1; k0
# Conversion from S1 to S2
S1 -> S2; k1*S1
# S3 is the adduct of S1 and S2
S1 + S2 -> S3; k2*S1*S2
# Dimerization of S1
2 S1 -> S2; k3*S1*S1
# More complex stoichiometry
S1 + 2 S2 -> 3 S3 + 5 S4; k4*S1*S2*S2
# Degradation of S4
S4 -> ; k5*S4

```

5.5.3 Rate Laws and Initializing Values

Reactions can be defined with a wide variety of rate laws

```

model pathway()
  # Examples of different rate laws and initialization

  S1 -> S2; k1*S1
  S2 -> S3; k2*S2 - k3*S3
  S3 -> S4; Vm*S3/(Km + S3)
  S4 -> S5; Vm*S4^n/(Km + S4)^n

  S1 = 10
  S2 = 0
  S3 = 0
  S4 = 0
  S5 = 0
  k1 = 0.1
  k2 = 0.2
  k3 = 0.2
  Vm = 6.7
  Km = 1E-3
  n = 4
end

```

5.5.4 Boundary Species

Boundary species are those species which are unaffected by the model. Usually this means they are fixed. There are two ways to declare boundary species.

- 1) Using a dollar sign to indicate that a particular species is fixed:

```
model pathway()  
  # Example of using $ to fix species  
  
  $S1 -> S2; k1*S1  
  S2 -> S3; k2*S2  
  S3 -> $S4; k3*S3  
end
```

- 2) Using the const keyword to declare species are fixed:

```
model pathway()  
  # Examples of using the const keyword to fix species  
  
  const S1, S4  
  S1 -> S2; k1*S1  
  S2 -> S3; k2*S2  
  S3 -> S4; k3*S3  
end
```

5.5.5 Compartments

For multi-compartment models, or models where the compartment size changes over time, you can define the compartments in Antimony by using the `compartment` keyword, and designate species as being in particular compartments with the `in` keyword:

```
model pathway()  
  # Examples of different compartments  
  
  compartment cytoplasm = 1.5, mitochondria = 2.6  
  const S1 in mitochondria  
  var S2 in cytoplasm  
  var S3 in cytoplasm  
  const S4 in cytoplasm  
  
  S1 -> S2; k1*S1  
  S2 -> S3; k2*S2  
  S3 -> S4; k3*S3  
end
```

5.5.6 Assignments

You can also initialize elements with more complicated formulas than simple numbers:

```
model pathway()  
  # Examples of different assignments  
  
  A = 1.2
```

(continues on next page)

(continued from previous page)

```

k1 = 2.3 + A
k2 = sin(0.5)
k3 = k2/k1

S1 -> S2; k1*S1
S2 -> S3; k2*S2
S3 -> S4; k3*S3
end

```

5.5.7 Assignments in Time

If you want to define some elements as changing in time, you can either define the formula a variable equals at all points in time with a `:=`, or you can define how a variable changes in time using `X'` (a [rate rule](#)) in which case you'll also need to define its initial starting value. The keyword `time` represents time.

```

model pathway()
  # Examples of assignments that change in time

  k1 := sin(time) # k1 will always equal the sine of time
  k2 = 0.2
  k2' = k1        #' k2 starts at 0.2, and changes according to the value
                  #   of k1: d(k2)/dt = k1

  S1 -> S2; k1*S1
  S2 -> S3; k2*S2
end

```

5.5.8 Piecewise Assignments

You can use `piecewise` to define piecewise assignments.

```

model pathway()
  # Examples of piecewise assignments
  $Xo -> S1; k1*Xo;
  S1 -> S2; k2*S1;
  S2 -> $X1; k3*S2;

  k1 := piecewise(0.1, time > 50, 20)

  k2 = 0.45; k3 = 0.34; Xo = 5;
end

```

Above will return `k1 = 0.1` if `time > 50` and `20` otherwise. A more complicated piecewise assignment can be defined as well.

```

model pathway()
  $Xo -> S1; k1*Xo;
  S1 -> S2; k2*S1;
  S2 -> $X1; k3*S2;

  k1 := piecewise(5, time > 20, 8, S2 < 100, 15)

```

(continues on next page)

(continued from previous page)

```
k2 = 0.45; k3 = 0.34; Xo = 5;
end
```

The above piecewise call will return 5 if time > 20, else it will return 8 if S2 < 100, else it will return 15. The piecewise function has this general “do this if this is true, else ...” pattern, and can be extended to include as many conditions as needed.

5.5.9 Events

Events are discontinuities in model simulations that change the definitions of one or more symbols at the moment when certain conditions apply. The condition is expressed as a boolean formula, and the definition changes are expressed as assignments, using the keyword `at`:

```
at (x>5): y=3, x=r+2
```

In a model with this event, at any moment when `x` transitions from being less than or equal to 5 to being greater than five, `y` will be assigned the value of 3, and `x` will be assigned the value of `r+2`, using whatever value `r` has at that moment. The following model sees the conversion of S1 to S2 until a threshold is reached, at which point the cycle is reset.

```
model reset()

  S1 -> S2; k1*S1

  E1: at (S2>9): S2=0, S1=10

  S1 = 10
  S2 = 0
  k1 = 0.5
end
```

For more advanced usage of events, see [Antimony’s reference documentation on events](#).

5.5.10 Function Definitions

You may create user-defined functions in a similar fashion to the way you create modules, and then use these functions in Antimony equations. These functions must be basic single equations, and act in a similar manner to macro expansions. As an example, you might define the quadratic equation and use it in a later equation as follows:

```
function quadratic(x, a, b, c)
  a*x^2 + b*x + c
end

model quad1
  S3 := quadratic(s1, k1, k2, k3);
end
```

This effectively defines S3 to always equal the equation $k1*s1^2 + k2*s1 + k3$.

5.5.11 Modular Models

Antimony was actually originally designed to allow the modular creation of models, and has a basic syntax set up to do so. For a full discussion of Antimony modularity, see the full documentation, but at the most basic level, you

define a re-usable module with the ‘model’ syntax, followed by parentheses where you define the elements you wish to expose, then import it by using the model’s name, and the local variables you want to connect to that module

```
# This creates a model 'side_reaction', exposing the variables 'S' and 'k1':
model side_reaction(S, k1)
  J0: S + E -> SE; k1*k2*S*E - k2*ES;
  E = 3;
  SE = E+S;
  k2 = 0.4;
end

# In this model, 'side_reaction' is imported twice:
model full_pathway
  -> S1; k1
  S1 -> S2; k2*S1
  S2 -> ; k3*S2

  A: side_reaction(S1, k4)
  B: side_reaction(S2, k5)

  S1 = 0
  S2 = 0
  k1 = 0.3
  k2 = 2.3
  k3 = 3.5
  k4 = 0.0004
  k5 = 1
end
```

In this model, A is a submodel that creates a side-reaction of S1 with A.E and A.SE, and B is a submodel that creates a side-reaction of S2 with B.E and B.SE. It is important to note that there is no connection between A.E and B.E (nor A.ES and B.ES): they are completely different species in the model.

5.5.12 Importing Files

More than one file may be used to define a set of modules in Antimony through the use of the ‘import’ keyword. At any point in the file outside of a module definition, use the word ‘import’ followed by the name of the file in quotation marks, and Antimony will include the modules defined in that file as if they had been cut and pasted into your file at that point. SBML files may also be included in this way:

```
import "models1.txt"
import "oscli.xml"

model mod2()
  A: mod1();
  B: oscli();
end
```

In this example, the file `models1.txt` is an Antimony file that defines the module `mod1`, and the file `oscli.xml` is an SBML file that defines a model named `oscli`. The Antimony module `mod2` may then use modules from either or both of the other imported files.

5.5.13 Units

While units do not affect the mathematics of SBML or Antimony models, you can define them in Antimony for annotation purposes by using the `unit` keyword:

```
unit substance = 1e-6 mole;
unit hour = 3600 seconds;
```

Adding an ‘s’ to the end of a unit name to make it plural is fine when defining a unit: 3600 second is the same as 3600 seconds. Compound units may be created by using formulas with `*`, `/`, and `^`. However, you must use base units when doing so (‘base units’ defined as those listed in Table 2 of the [SBML Level 3 Version 1 specification](#), which mostly are SI and SI-derived units).

```
unit micromole = 10e-6 mole / liter;
unit daily_feeding = 1 item / 86400 seconds
unit voltage = 1000 grams * meters^2 / seconds^-3 * ampere^-1
```

You may use units when defining formulas using the same syntax as above: any number may be given a unit by writing the name of the unit after the number. When defining a symbol (of any numerical type: species, parameter, compartment, etc.), you can either use the same technique to give it an initial value and a unit, or you may just define its units by using the ‘has’ keyword:

```
unit foo = 100 mole/5 liter;
x = 40 foo/3 seconds; # '40' now has units of 'foo' and '3' units of 'seconds'.
y = 3.3 foo;          # 'y' is given units of 'foo' and an initial
                      #   value of '3.3'.
z has foo;            # 'z' is given units of 'foo'.
```

Antimony does not calculate any derived units: in the above example, ‘x’ is fully defined in terms of moles per liter per second, but it is not annotated as such.

As with many things in Antimony, you may use a unit before defining it: ‘x = 10 ml’ will create a parameter x and a unit ‘ml’.

5.6 Simulating Models

Antimony models can be converted into a `RoadRunner` instance, which can be used to simulate the model. The function `loada` converts Antimony into a simulator instance. This example shows how to get the Antimony / SBML representation of the current state of the model. After running a simulation, the concentrations of the state variables will change. You can retrieve the Antimony representation of the current state of the model using `getCurrentAntimony` on the `RoadRunner` instance. This example shows the change in the Antimony / SBML representation. The example also shows how to use variable stepping in a simulation.

```
import tellurium as te

print('-' * 80)
te.printVersionInfo()
print('-' * 80)

r = te.loada('''
model example
  p1 = 0;
  at time>=10: p1=10;
  at time>=20: p1=0;
end
```

(continues on next page)

(continued from previous page)

```

'''
# convert current state of model back to Antimony / SBML
ant_str_before = r.getCurrentAntimony()
sbml_str_before = r.getCurrentSBML()
# r.exportToSBML('/path/to/test.xml')

# set selections
r.selections=['time', 'p1']
r.integrator.setValue("variable_step_size", False)
r.resetAll()
s1 = r.simulate(0, 40, 40)
r.plot()
# hitting the trigger point directly works
r.resetAll()
s2 = r.simulate(0, 40, 21)
r.plot()

# variable step size also does not work
r.integrator.setValue("variable_step_size", True)
r.resetAll()
s3 = r.simulate(0, 40)
r.plot()

# convert current state of model back to Antimony / SBML
ant_str_after = r.getCurrentAntimony()
sbml_str_after = r.getCurrentSBML()

import difflib
print("Comparing Antimony at time 0 & 40 (expect no differences)")
print('\n'.join(list(difflib.unified_diff(ant_str_before.splitlines(), ant_str_after.
↪splitlines(), fromfile="before.sb", tofile="after.sb"))))

# now simulate up to time 15
r.resetAll()
s4 = r.simulate(0, 15)
r.plot()

# convert current state of model back to Antimony / SBML
ant_str_after2 = r.getCurrentAntimony()
sbml_str_after2 = r.getCurrentSBML()

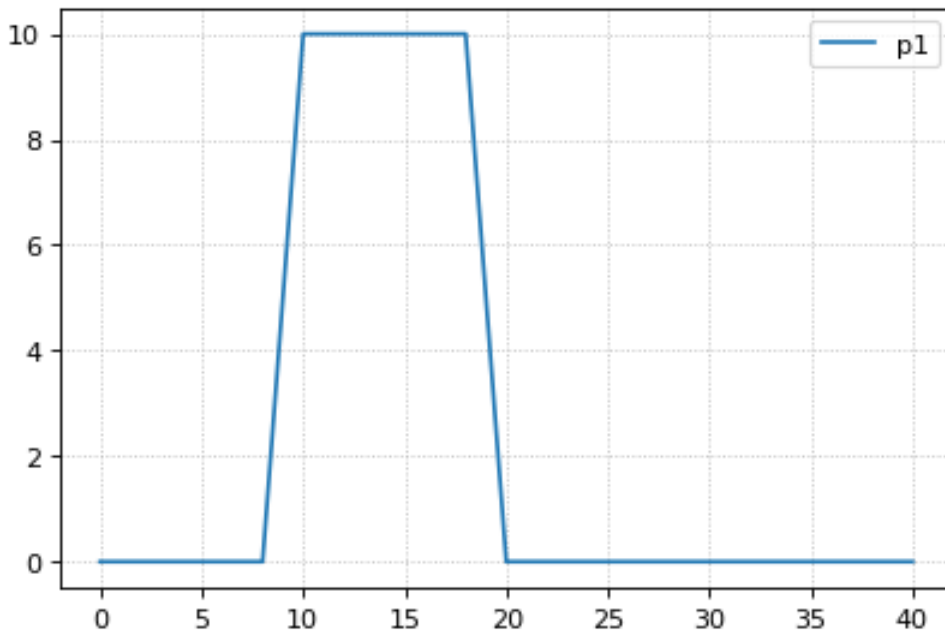
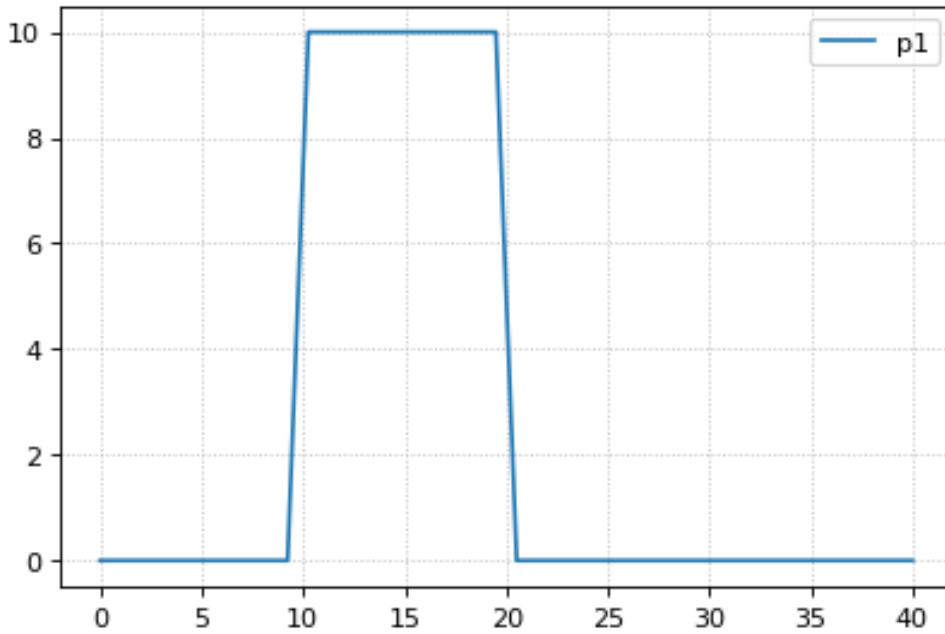
print("Comparing Antimony at time 0 & 15")
print('\n'.join(list(difflib.unified_diff(ant_str_before.splitlines(), ant_str_after2.
↪splitlines(), fromfile="before.sb", tofile="after.sb"))))

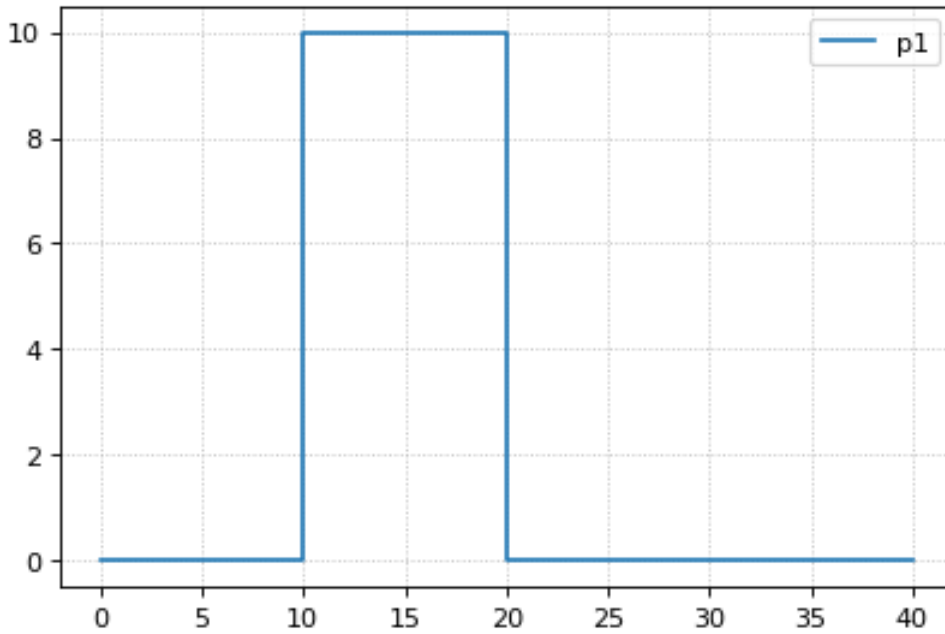
```

```

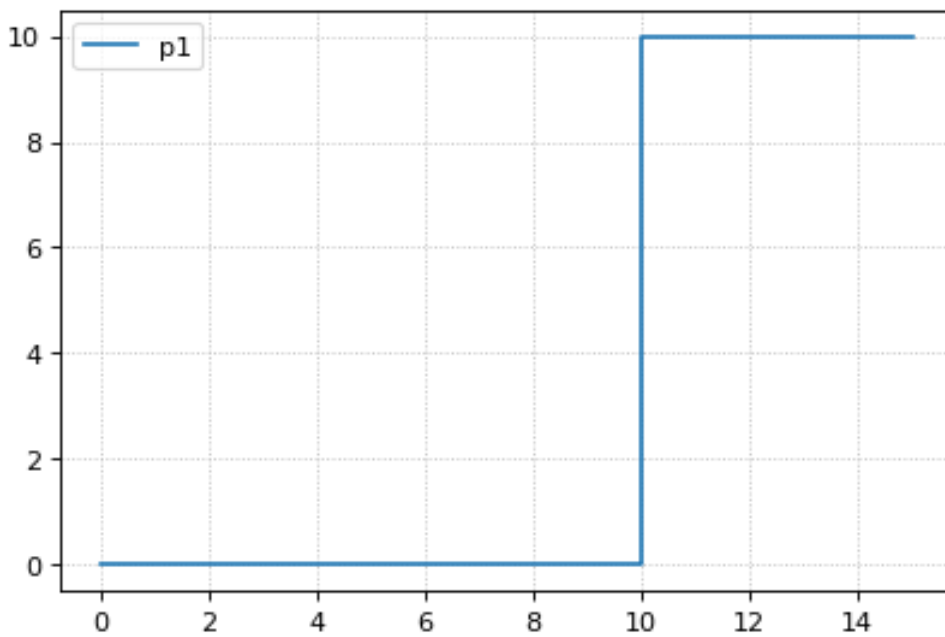
-----
tellurium : 2.1.0
roadrunner : 1.5.1
antimony : 2.9.4
libsbml : 5.15.0
libsedml : 0.4.3
phrasedml : 1.0.9
-----

```





Comparing Antimony at time 0 & 40 (expect no differences)



Comparing Antimony at time 0 & 15

```
--- before.sb
```

```
+++ after.sb
```

```
@@ -6,7 +6,7 @@
```

```
  _E1: at time >= 20: p1 = 0;
```

(continues on next page)

(continued from previous page)

```
// Variable initializations:  
- p1 = 0;  
+ p1 = 10;  
  
// Other declarations:  
var p1;
```

```
r.getSimulationData()
```

```
      time, p1  
[[      0,  0],  
 [0.000514839, 0],  
 [   5.1489, 0],  
 [      10, 0],  
 [      10, 10],  
 [  10.0002, 10],  
 [  12.2588, 10],  
 [      15, 10]]
```

5.7 Language Reference

5.7.1 Species and Reactions

The simplest Antimony file may simply have a list of reactions containing species, along with some initializations. Reactions are written as two lists of species, separated by a `->`, and followed by a semicolon:

```
S1 + E -> ES;
```

Optionally, you may provide a reaction rate for the reaction by including a mathematical expression after the semicolon, followed by another semicolon:

```
S1 + E -> ES; k1*k2*S1*E - k2*ES;
```

You may also give the reaction a name by prepending the name followed by a colon:

```
J0: S1 + E -> ES; k1*k2*S1*E - k2*ES;
```

The same effect can be achieved by setting the reaction rate separately, by assigning the reaction rate to the reaction name with an `=`:

```
J0: S1 + E -> ES;  
J0 = k1*k2*S1*E - k2*ES;
```

You may even define them in the opposite order-they are all ways of saying the same thing.

If you want, you can define a reaction to be irreversible by using `=>` instead of `->`:

```
J0: S1 + E => ES;
```

However, if you additionally provide a reaction rate, that rate is not checked to ensure that it is compatible with an irreversible reaction.

At this point, Antimony will make several assumptions about your model. It will assume (and require) that all symbols that appear in the reaction itself are species. Any symbol that appears elsewhere that is not used or defined as a species is ‘undefined’; ‘undefined’ symbols may later be declared or used as species or as ‘formulas’, Antimony’s term for constants and packaged equations like SBML’s assignment rules. In the above example, k_1 and k_2 are (thus far) undefined symbols, which may be assigned straightforwardly:

```
J0: S1 + E -> ES; k1*k2*S1*E - k2*ES;
k1 = 3;
k2 = 1.4;
```

More complicated expressions are also allowed, as are the creation of symbols which exist only to simplify or clarify other expressions:

```
pH = 7;
k3 = -log10 (pH);
```

The initial concentrations of species are defined in exactly the same way as formulas, and may be just as complex (or simple):

```
S1 = 2;
E = 3;
ES = S1 + E;
```

Order for any of the above (and in general in Antimony) does not matter at all: you may use a symbol before defining it, or define it before using it. As long as you do not use the same symbol in an incompatible context (such as using the same name as a reaction and a species), your resulting model will still be valid. Antimony files written by libAntimony will adhere to a standard format of defining symbols, but this is not required.

5.7.2 Modules

Antimony input files may define several different models, and may use previously-defined models as parts of newly-defined models. Each different model is known as a ‘module’, and is minimally defined by putting the keyword ‘model’ (or ‘module’, if you like) and the name you want to give the module at the beginning of the model definitions you wish to encapsulate, and putting the keyword ‘end’ at the end:

```
model example
  S + E -> ES;
end
```

After this module is defined, it can be used as a part of another model (this is the one time that order matters in Antimony). To import a module into another module, simply use the name of the module, followed by parentheses:

```
model example
  S + E -> ES;
end

model example2
  example();
end
```

This is usually not very helpful in and of itself—you’ll likely want to give the submodule a name so you can refer to the things inside it. To do this, prepend a name followed by a colon:

```
model example2
  A: example();
end
```

Now, you can modify or define elements in the submodule by referring to symbols in the submodule by name, prepended with the name you've given the module, followed by a `.`:

```
model example2
  A: example();
  A.S = 3;
end
```

This results in a model with a single reaction $A.S + A.E \rightarrow A.ES$ and a single initial condition $A.S = 3$.

You may also import multiple copies of modules, and modules that themselves contain submodules:

```
model example3
  A: example();
  B: example();
  C: example2();
end
```

This would result in a model with three reactions and a single initial condition.

```
A.S + A.E -> A.ES
B.S + B.E -> B.ES
C.A.S + C.A.E -> C.A.ES
C.A.S = 3;
```

You can also use the species defined in submodules in new reactions:

```
model example4
  A: example();
  A.S -> ; kdeg*A.S;
end
```

When combining multiple submodules, you can also 'attach' them to each other by declaring that a species in one submodule is the same species as is found in a different submodule by using the `is` keyword $A.S \text{ is } B.S$. For example, let's say that we have a species which is known to bind reversibly to two different species. You could set this up as the following:

```
model side_reaction
  J0: S + E -> SE; k1*k2*S*E - k2*ES;
  S = 5;
  E = 3;
  SE = E+S;
  k1 = 1.2;
  k2 = 0.4;
end

model full_reaction
  A: side_reaction();
  B: side_reaction();
  A.S is B.S;
end
```

If you wanted, you could give the identical species a new name to more easily use it in the `full_reaction` module:

```

model full_reaction
  var species S;
  A: side_reaction();
  B: side_reaction()
  A.S is S;
  B.S is S;
end

```

In this system, S is involved in two reversible reactions with exactly the same reaction kinetics and initial concentrations. Let's now say the reaction rate of the second side-reaction takes the same form, but that the kinetics are twice as fast, and the starting conditions are different:

```

model full_reaction
  var species S;
  A: side_reaction();
  A.S is S;
  B: side_reaction();
  B.S is S;
  B.k1 = 2.4;
  B.k2 = 0.8;
  B.E = 10;
end

```

Note that since we defined the initial concentration of SE as $S + E$, B.SE will now have a different initial concentration, since B.E has been changed.

Finally, we add a third side reaction, one in which S binds irreversibly, and where the complex it forms degrades. We'll need a new reaction rate, and a whole new reaction as well:

```

model full_reaction
  var species S;
  A: side_reaction();
  A.S is S;
  B: side_reaction();
  B.S is S;
  B.k1 = 2.4;
  B.k2 = 0.8;
  B.E = 10;
  C: side_reaction();
  C.S is S;
  C.J0 = C.k1*C.k2*S*C.E
  J3: C.SE -> ; C.SE*k3;
  k3 = 0.02;
end

```

Note that defining the reaction rate of C.J0 used the symbol S; exactly the same result would be obtained if we had used C.S or even A.S or B.S. Antimony knows that those symbols all refer to the same species, and will give them all the same name in subsequent output.

For convenience and style, modules may define an interface where some symbols in the module are more easily renamed. To do this, first enclose a list of the symbols to export in parentheses after the name of the model when defining it:

```

model side_reaction(S, k1)
  J0: S + E -> SE; k1*k2*S*E - k2*ES;
  S = 5;
  E = 3;

```

(continues on next page)

(continued from previous page)

```
SE = E+S;  
k1 = 1.2;  
k2 = 0.4;  
end
```

Then when you use that module as a submodule, you can provide a list of new symbols in parentheses:

```
A: side_reaction(spec2, k2);
```

is equivalent to writing:

```
A.S is spec2;  
A.k1 is k2;
```

One thing to be aware of when using this method: Since wrapping definitions in a defined model is optional, all ‘bare’ declarations are defined to be in a default module with the name `__main`. If there are no unwrapped definitions, `__main` will still exist, but will be empty.

As a final note: use of the `is` keyword is not restricted to elements inside submodules. As a result, if you wish to change the name of an element (if, for example, you want the reactions to look simpler in Antimony, but wish to have a more descriptive name in the exported SBML), you may use `is` as well:

```
A -> B;  
A is ABA;  
B is ABA8OH;
```

is equivalent to writing:

```
ABA -> ABA8OH;
```

5.7.3 Module conversion factors

Occasionally, the unit system of a submodel will not match the unit system of the containing model, for one or more model elements. In this case, you can use conversion factor constructs to bring the submodule in line with the containing model.

If time is different in the submodel (affecting reactions, rate rules, delay, and ‘time’), use the `timeconv` keyword when declaring the submodel:

```
A1: submodel(), timeconv=60;
```

This construct means that one unit of time in the submodel multiplied by the time conversion factor should equal one unit of time in the parent model.

Reaction extent may also be different in the submodel when compared to the parent model, and may be converted with the `extentconv` keyword:

```
A1: submodel(), extentconv=1000;
```

This construct means that one unit of reaction extent in the submodel multiplied by the extent conversion factor should equal one unit of reaction extent in the parent model.

Both time and extent conversion factors may be numbers (as above) or they may be references to constant parameters. They may also both be used at once:

```
A1: submodel(), timeconv=tconv, extentconv=xconv;
```

Individual components of submodels may also be given conversion factors, when the `is` keyword is used. The following two constructs are equivalent ways of applying conversion factor `cf` to the synchronized variables `x` and `A1.y`:

```
A1.y * cf is x;
A1.y is x / cf;
```

When flattened, all of these conversion factors will be incorporated into the mathematics.

5.7.4 Submodel deletions

Sometimes, an element of a submodel has to be removed entirely for the model to make sense as a whole. A degradation reaction might need to be removed, for example, or a now-superfluous species. To delete an element of a submodel, use the `delete` keyword:

```
delete A1.S1;
```

In this case, `S1` will be removed from submodel `A1`, as will any reactions `S1` participated in, plus any mathematical formulas that had `S1` in them.

Similarly, sometimes it is necessary to clear assignments and rules to a variable. To accomplish this, simply declare a new assignment or rule for the variable, but leave it blank:

```
A1.S1 = ;
A1.S2 := ;
A1.S3' = ;
```

This will remove the appropriate initial assignment, assignment rule, or rate rule (respectively) from the submodel.

5.7.5 Constant and variable symbols

Some models have ‘boundary species’ in their reactions, or species whose concentrations do not change as a result of participating in a reaction. To declare that a species is a boundary species, use the ‘`const`’ keyword:

```
const S1;
```

While you’re declaring it, you may want to be more specific by using the ‘`species`’ keyword:

```
const species S1;
```

If a symbol appears as a participant in a reaction, Antimony will recognize that it is a species automatically, so the use of the keyword ‘`species`’ is not required. If, however, you have a species which never appears in a reaction, you will need to use the ‘`species`’ keyword.

If you have several species that are all constant, you may declare this all in one line:

```
const species S1, S2, S3;
```

While species are variable by default, you may also declare them so explicitly with the ‘`var`’ keyword:

```
var species S4, S5, S6;
```

Alternatively, you may declare a species to be a boundary species by prepending a ‘`$`’ in front of it:

```
S1 + $E -> ES;
```

This would set the level of ‘E’ to be constant. You can use this symbol in declaration lists as well:

```
species S1, $S2, $S3, S4, S5, $S6;
```

This declares six species, three of which are variable (by default) and three of which are constant.

Likewise, formulas are constant by default. They may be initialized with an equals sign, with either a simple or a complex formula:

```
k1 = 5;  
k2 = 2*S1;
```

You may also explicitly declare whether they are constant or variable:

```
const k1;  
var k2;
```

and be more specific and declare that both are formulas:

```
const formula k1;  
var formula k2;
```

Variables defined with an equals sign are assigned those values at the start of the simulation. In SBML terms, they use the ‘Initial Assignment’ values. If the formula is to vary during the course of the simulation, use the Assignment Rule (or Rate Rule) syntax, described later.

You can also mix-and-match your declarations however best suits what you want to convey:

```
species S1, S2, S3, S4;  
formula k1, k2, k3, k4;  
const S1, S4, k1, k3;  
var S2, S3, k2, k4;
```

Antimony is a pure model definition language, meaning that all statements in the language serve to build a static model of a dynamic biological system. Unlike Jarnac, sequential programming techniques such as re-using a variable for a new purpose will not work:

```
pH = 7;  
k1 = -log10(pH);  
pH = 8.2;  
k2 = -log10(pH);
```

In a sequential programming language, the above would result in different values being stored in k1 and k2. (This is how Jarnac works, for those familiar with that language/simulation environment.) In a pure model definition language like Antimony, ‘pH’, ‘k1’, ‘k2’, and even the formula ‘-log10(pH)’ are static symbols that are being defined by Antimony statements, and not processed in any way. A simulator that requests the mathematical expression for k1 will receive the string ‘-log10(pH)’; the same string it will receive for k2. A request for the mathematical expression for pH will receive the string “8.2”, since that’s the last definition found in the file. As such, k1 and k2 will end up being identical.

As a side note, we considered having libAntimony store a warning when presented with an input file such as the example above with a later definition overwriting an earlier definition. However, there was no way with our current interface to let the user know that a warning had been saved, and it seemed like there could be a number of cases where the user might legitimately want to override an earlier definition (such as when using submodules, as we’ll get to in a bit). So for now, the above is valid Antimony input that just so happens to produce exactly the same output as:


```
pH = 8.2;
k1 = -log10(pH);
k2 = -log10(pH);
```

5.7.6 Compartments

A compartment is a demarcated region of space that contains species and has a particular volume. In Antimony, you may ignore compartments altogether, and all species are assumed to be members of a default compartment with the imaginative name ‘default_compartment’ with a constant volume of 1. You may define other compartments by using the ‘compartment’ keyword:

```
compartment comp1;
```

Compartments may also be variable or constant, and defined as such with ‘var’ and ‘const’:

```
const compartment comp1;
var compartment comp2;
```

The volume of a compartment may be set with an ‘=’ in the same manner as species and reaction rates:

```
comp1 = 5;
comp2 = 3*comp1;
```

To declare that something is in a compartment, the ‘in’ keyword is used, either during declaration:

```
compartment comp1 in comp2;
const species S1 in comp2;
S2 in comp2;
```

or during assignment for reactions:

```
J0 in comp1: x -> y; k1*x;
y -> z; k2*y in comp2;
```

or submodules:

```
M0 in comp2: submod();
submod2(y) in comp3;
```

or other variables:

```
S1 in comp2 = 5;
```

Here are Antimony’s rules for determining which compartment something is in:

- If the symbol has been declared to be in a compartment, it is in that compartment.
- If not, if the symbol is in a DNA strand (see the next section) which has been declared to be in a compartment, it is in that compartment. If the symbol is in multiple DNA strands with conflicting compartments, it is in the compartment of the last declared DNA strand that has a declared compartment in the model.
- If not, if the symbol is a member of a reaction with a declared compartment, it is in that compartment. If the symbol is a member of multiple reactions with conflicting compartments, it is in the compartment of the last declared reaction that has a declared compartment.

- If not, if the symbol is a member of a submodule with a declared compartment, it is in that compartment. If the symbol is a member of multiple submodules with conflicting compartments, it is in the compartment of the last declared submodule that has a declared compartment.
- If not, the symbol is in the compartment ‘default_compartment’, and is treated as having no declared compartment for the purposes of determining the compartments of other symbols.

Note that declaring that one compartment is ‘in’ a second compartment does not change the compartment of the symbols in the first compartment:

```
compartment c1, c2;  
species s1 in c1, s2 in c1;  
c1 in c2;
```

yields:

```
symbol compartment  
s1 c1  
s2 c1  
c1 c2  
c2 default_compartment
```

Compartments may not be circular: `c1 in c2; c2 in c3; c3 in c1` is illegal.

5.7.7 Events

Events are discontinuities in model simulations that change the definitions of one or more symbols at the moment when certain conditions apply. The condition is expressed as a boolean formula, and the definition changes are expressed as assignments, using the keyword ‘at’ and the following syntax:

```
at: variable1=formula1, variable2=formula2 [etc];
```

such as:

```
at (x>5): y=3, x=r+2;
```

You may also give the event a name by prepending it with a colon:

```
E1: at (x>=5): y=3, x=r+2;
```

(you may also claim an event is ‘in’ a compartment just like everything else (‘E1 in comp1:’). This declaration will never change the compartment of anything else.)

In addition, there are a number of concepts in SBML events that can now be encoded in Antimony. If event assignments are to occur after a delay, this can be encoded by using the ‘after’ keyword:

```
E1: at 2 after (x>5): y=3, x=r+2;
```

This means to wait two time units after x transitions from less than five to more than five, then change y to 3 and x to $r+2$. The delay may also itself be a formula:

```
E1: at 2*z/y after (x>5): y=3, x=r+2;
```

For delayed events (and to a certain extent with simultaneous events, discussed below), one needs to know what values to use when performing event assignments: the values from the time the event was triggered, or the values from the time the event assignments are being executed? By default (in Antimony, as in SBML Level 2) the first holds true:

event assignments are to use values from the moment the event is triggered. To change this, the keyword ‘fromTrigger’ is used:

```
E1: at 2*z/y after (x>5), fromTrigger=false: y=3, x=r+2;
```

You may also declare ‘fromTrigger=true’ to explicitly declare what is the default.

New complications can arise when event assignments from multiple events are to execute at the same time: which event assignments are to be executed first? By default, there is no defined answer to this question: as long as both sets of assignments are executed, either may be executed first. However, if the model depends on a particular order of execution, events may be given priorities, using the priority keyword:

```
E1: at ((x>5) && (z>4)), priority=1: y=3, x=r+2;
E2: at ((x>5) && (q>7)), priority=0: y=5: x=r+6;
```

In situations where $z>4$, $q>7$, and $x>5$, and then x increases, both E1 and E2 will trigger at the same time. Since both modify the same values, it makes a difference in which order they are executed—in this case, whichever happens last takes precedence. By giving the events priorities (higher priorities execute first) the result of this situation is deterministic: E2 will execute last, and y will equal 5 and not 3.

Another question is whether, if at the beginning of the simulation the trigger condition is ‘true’, it should be considered to have just transitioned to being true or not. The default is no, meaning that no event may trigger at time 0. You may override this default by using the ‘t0’ keyword:

```
E1: at (x>5), t0=false: y=3, x=r+2;
```

In this situation, the value at t_0 is considered to be false, meaning it can immediately transition to true if x is greater than 5, triggering the event. You may explicitly state the default by using ‘ $t_0 = \text{true}$ ’.

Finally, a different class of events is often modeled in some situations where the trigger condition must persist in being true from the entire time between when the event is triggered to when it is executed. By default, this is not the case for Antimony events, and, once triggered, all events will execute. To change the class of your event, use the keyword ‘persistent’:

```
E1: at 3 after (x>5), persistent=true: y=3, x=r+2;
```

For this model, x must be greater than 5 for three seconds before executing its event assignments: if x dips below 5 during that time, the event will not fire. To explicitly declare the default situation, use ‘persistent=false’.

The ability to change the default priority, t_0 , and persistent characteristics of events was introduced in SBML Level 3, so if you translate your model to SBML Level 2, it will lose the ability to define functionality other than the default. For more details about the interpretation of these event classifications, see the SBML Level 3 specification.

5.7.8 Assignment Rules

In some models, species and/or variables change in a manner not described by a reaction. When a variable receives a new value at every point in the model, this can be expressed in an assignment rule, which in Antimony is formulated with a ‘:=’ as:

```
Ptot := P1 + P2 + PE;
```

In this example, ‘Ptot’ will continually be updated to reflect the total amount of ‘P’ present in the model.

Each symbol (species or formula) may have only one assignment rule associated with it. If an Antimony file defines more than one rule, only the last will be saved.

When species are used as the target of an assignment rule, they are defined to be ‘boundary species’ and thus ‘const’. Antimony doesn’t have a separate syntax for boundary species whose concentrations never change vs. boundary

species whose concentrations change due to assignment rules (or rate rules, below). SBML distinguishes between boundary species that may change and boundary species that may not, but in Antimony, all boundary species may change as the result of being in an Assignment Rule or Rate Rule.

5.7.9 Signals

Signals can be generated by combining assignment rules with events.

Step Input

The simplest signal is input step. The following code implements a step that occurs at time = 20 with a magnitude of f . A trigger is used to set a trigger variable α which is used to initiate the step input in an assignment expression.

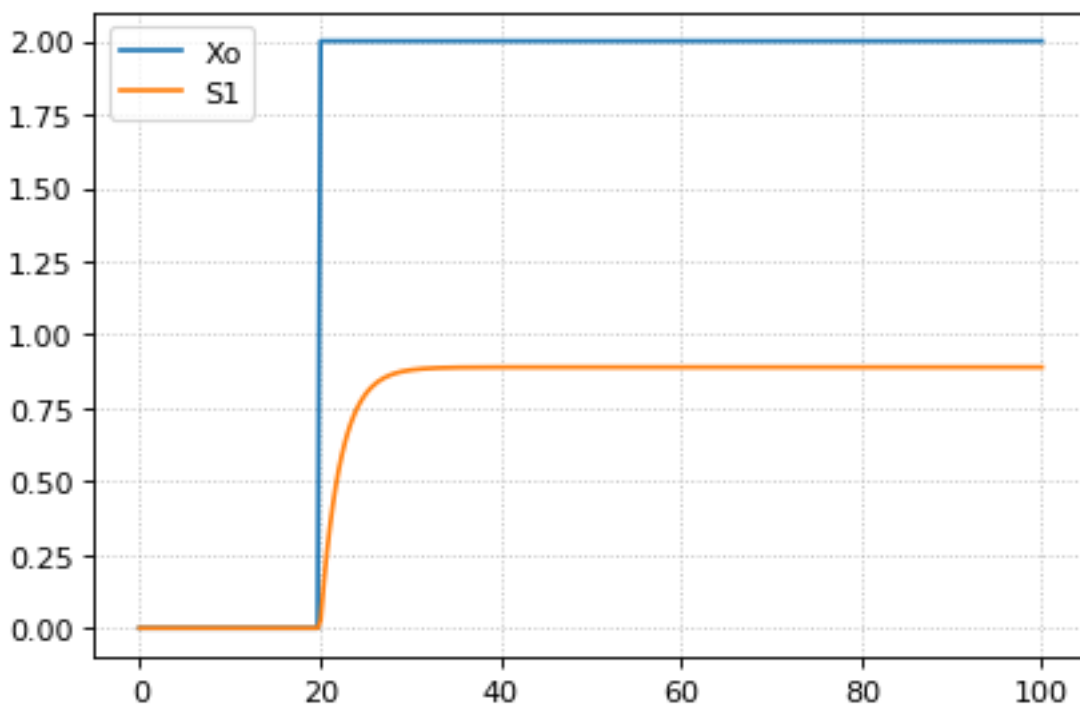
```
import tellurium as te
import roadrunner

r = te.loada("""
$Xo -> S1; k1*Xo;
S1 -> $X1; k2*S1;

k1 = 0.2; k2 = 0.45;

alpha = 0; f = 2
Xo := alpha*f
at time > 20:
    alpha = 1
""")

m = r.simulate(0, 100, 300, ['time', 'Xo', 'S1'])
r.plot()
```



Ramp

The following code starts a ramp at 20 time units by setting the p1 variable to one. This variable is used to activate a ramp function.

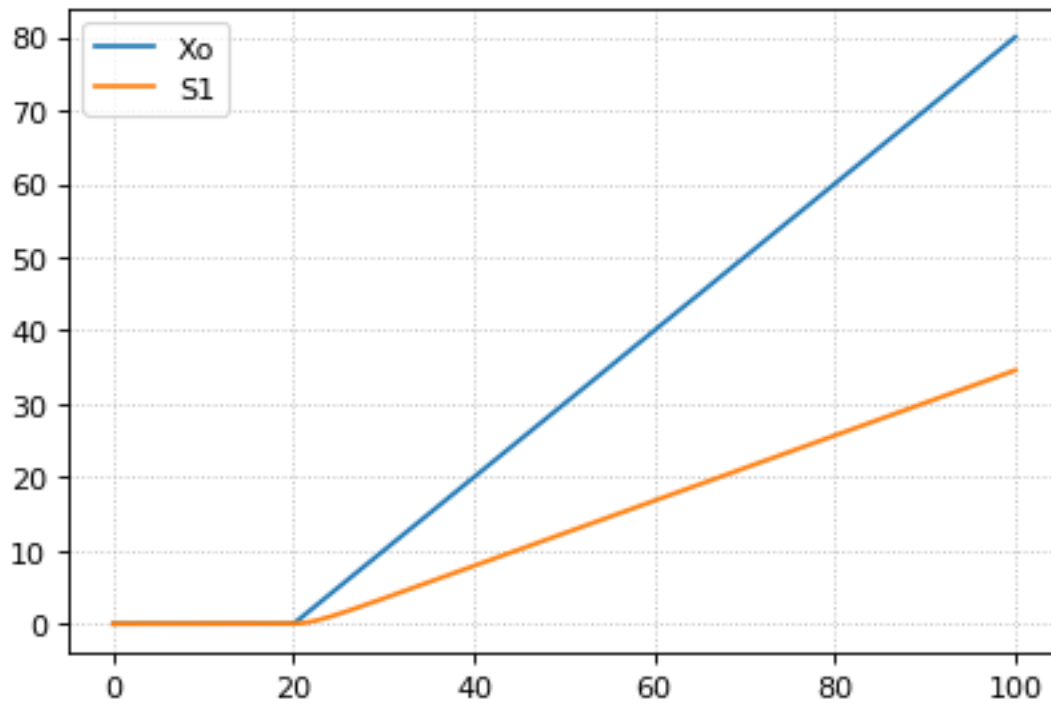
```
import tellurium as te
import roadrunner

r = te.loada("""
$Xo -> S1; k1*Xo;
S1 -> $X1; k2*S1;

k1 = 0.2; k2 = 0.45;

p1 = 0;
Xo := p1*(time - 20)
at time > 20:
    p1 = 1
""")

m = r.simulate(0, 100, 200, ['time', 'Xo', 'S1'])
r.plot()
```



Ramp then Stop

The following code starts a ramp at 20 time units by setting the p1 variable to one and then stopping the ramp 20 time units later. At 20 time units later a new term is switched on which subtracts the ramp slope that results in a horizontal line.

```
import tellurium as te
import roadrunner
```

(continues on next page)

(continued from previous page)

```

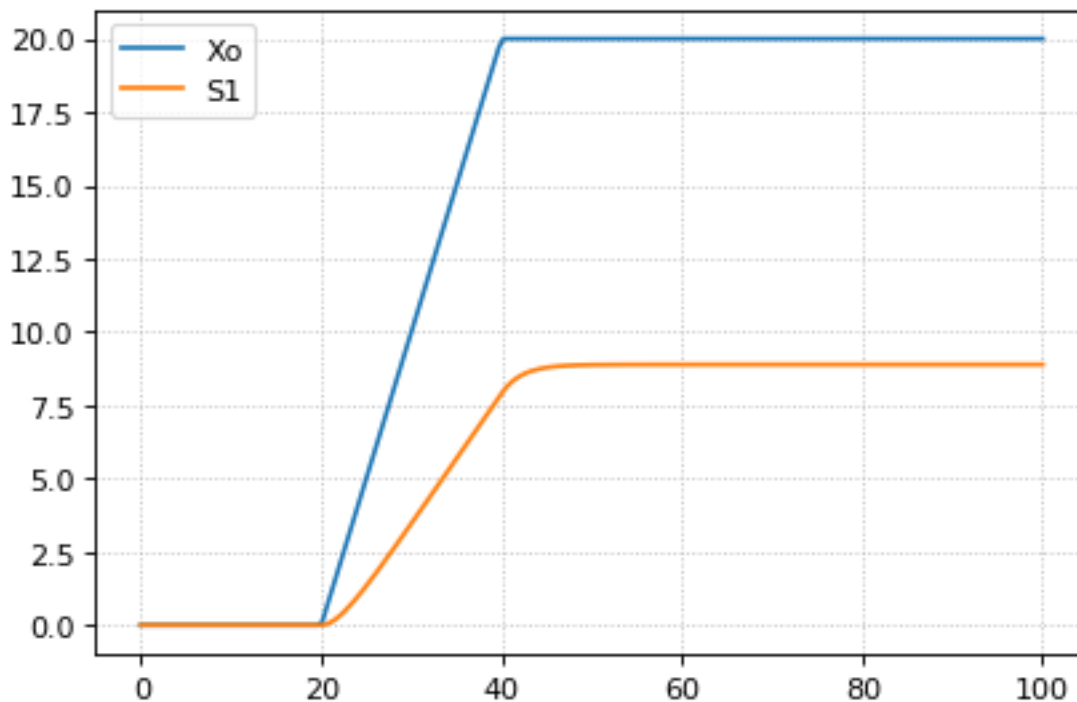
r = te.loada("""
$Xo -> S1; k1*Xo;
S1 -> $X1; k2*S1;

k1 = 0.2; k2 = 0.45;

p1 = 0; p2 = 0
Xo := p1*(time - 20) - p2*(time - 40)
at time > 20:
    p1 = 1
at time > 40:
    p2 = 1
""")

m = r.simulate(0, 100, 200, ['time', 'Xo', 'S1'])
r.plot()

```



Pulse

The following code starts a pulse at 20 time units by setting the p1 variable to one and then stops the pulse 20 time units later by setting p2 equal to zero.

```

import tellurium as te
import roadrunner

r = te.loada("""
$Xo -> S1; k1*Xo;
S1 -> $X1; k2*S1;

```

(continues on next page)

(continued from previous page)

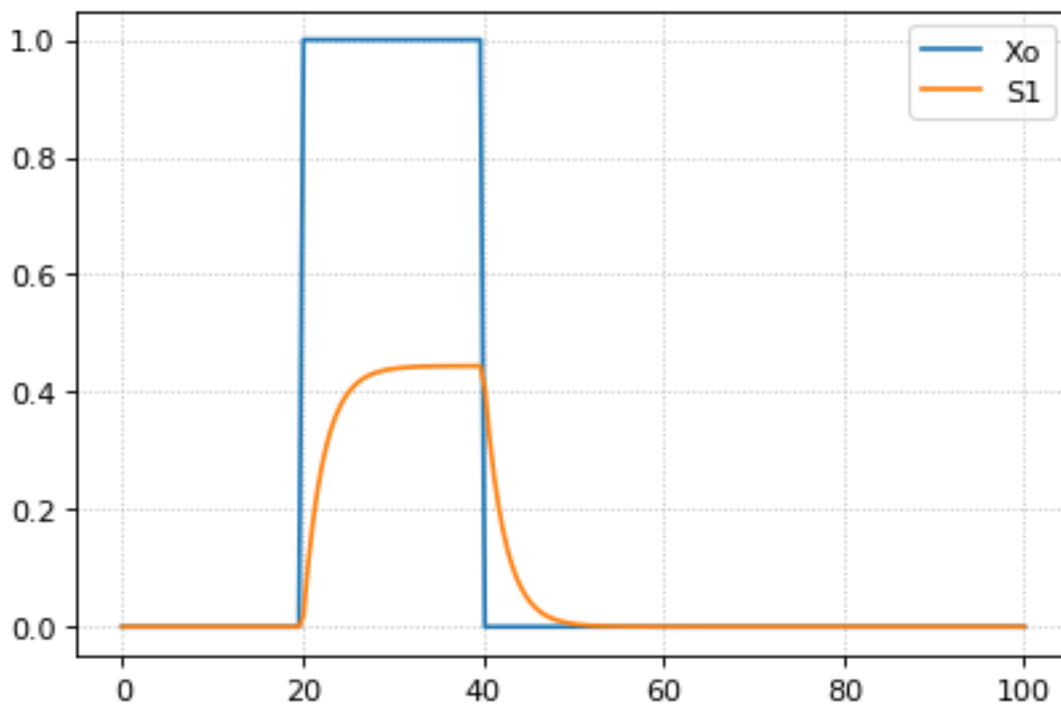
```

k1 = 0.2; k2 = 0.45;

p1 = 0; p2 = 1
Xo := p1*p2
at time > 20:
    p1 = 1
at time > 40:
    p2 = 0
""")

m = r.simulate (0, 100, 200, ['time', 'Xo', 'S1'])
r.plot()

```



Sinusoidal Input

The following code starts a sinusoidal input at 20 time units by setting the p1 variable to one.

```

import tellurium as te
import roadrunner

r = te.loada("""
$Xo -> S1; k1*Xo;
S1 -> $X1; k2*S1;

k1 = 0.2; k2 = 0.45;

p1 = 0;
Xo := p1*(sin (time) + 1)

```

(continues on next page)

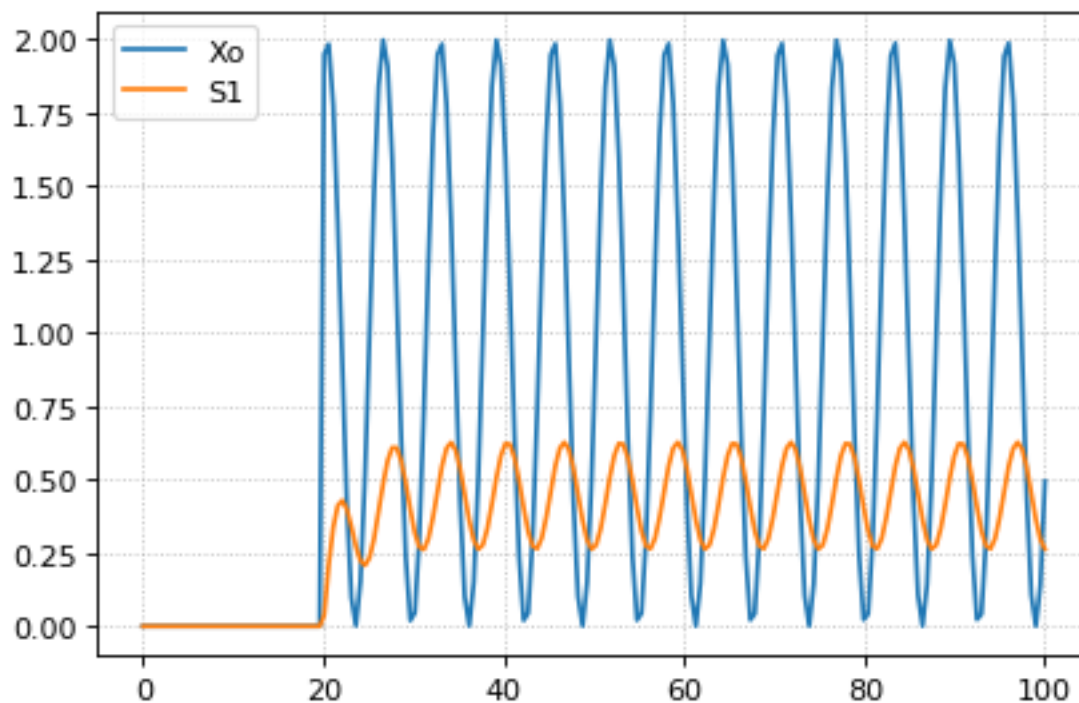
(continued from previous page)

```

at time > 20:
    p1 = 1
    "")

m = r.simulate (0, 100, 200, ['time', 'Xo', 'S1'])
r.plot()

```



5.7.10 Rate Rules

Rate rules define the change in a symbol's value over time instead of defining its new value. In this sense, they are similar to reaction rate kinetics, but without an explicit stoichiometry of change. These may be modeled in Antimony by appending an apostrophe to the name of the symbol, and using an equals sign to define the rate:

```
S1' = V1*(1 - S1)/(K1 + (1 - S1)) - V2*S1/(K2 + S1)
```

Note that unlike initializations and assignment rules, formulas in rate rules may be self-referential, either directly or indirectly.

Any symbol may have only one rate rule or assignment rule associated with it. Should it find more than one, only the last will be saved.

5.7.11 Display Names

When some tools visualize models, they make a distinction between the 'id' of an element, which must be unique to the model and which must conform to certain naming conventions, and the 'name' of an element, which does not have to be unique and which has much less stringent naming requirements. In Antimony, it is the id of elements which is used everywhere. However, you may also set the 'display name' of an element by using the 'is' keyword and putting the name in quotes:


```
A.k1 is "reaction rate k1";
S34 is "Ethyl Alcohol";
```

5.7.12 DNA Strands

A new concept in Antimony that has not been modeled explicitly in previous model definition languages such as SBML is the idea of having DNA strands where downstream elements can inherit reaction rates from upstream elements. DNA strands are declared by connecting symbols with `--`:

```
--P1--G1--stop--P2--G2--
```

You can also give the strand a name:

```
dna1: --P1--G1--
```

By default, the reaction rate or formula associated with an element of a DNA strand is equal to the reaction rate or formula of the element upstream of it in the strand. Thus, if P1 is a promoter and G1 is a gene, in the model:

```
dna1: --P1--G1--
P1 = S1*k;
G1: -> prot1;
```

the reaction rate of G1 will be $S1*k$.

It is also possible to modulate the inherited reaction rate. To do this, we use ellipses `...` as shorthand for ‘the formula for the element upstream of me’. Let’s add a ribosome binding site that increases the rate of production of protein by a factor of three, and say that the promoter actually increases the rate of protein production by $S1*k$ instead of setting it to $S1*k$:

```
dna1: --P1--RBS1--G1--
P1 = S1*k + ...;
RBS1 = ...*3;
G1: -> prot1;
```

Since in this model, nothing is upstream of P1, the upstream rate is set to zero, so the final reaction rate of G1 is equal to $(S1*k + 0) * 3$.

Valid elements of DNA strands include formulas (operators), reactions (genes), and other DNA strands. Let’s wrap our model so far in a submodule, and then use the strand in a new strand:

```
model strand1()
  dna1: --P1--RBS1--G1--
  P1 = S1*k + ...;
  RBS1 = ...*3;
  G1: -> prot1;
end

model fullstrand()
  A: strand1();
  fulldna: P2--A.dna1
  P2 = S2*k2;
end
```

In the model `fullstrand`, the reaction that produces `A.prot1` is equal to $(A.S1*A.k + (S2*k2)) * 3$.

Operators and genes may be duplicated and appear in multiple strands:

```
dna1:  --P1--RBS1--G1--
dna2:  P2--dna1
dna3:  P2--RBS2--G1
```

Strands, however, count as unique constructs, and may only appear as singletons or within a single other strand (and may not, of course, exist in a loop, being contained in a strand that it itself contains).

If the reaction rate or formula for any duplicated symbol is left at the default or if it contains ellipses explicitly (`'...'`), it will be equal to the sum of all reaction rates in all the strands in which it appears. If we further define our above model:

```
dna1:  --P1--RBS1--G1--
dna2:  P2--dna1
dna3:  P2--RBS2--G1
P1 = ...+0.3;
P2 = ...+1.2;
RBS1 = ...*0.8;
RBS2 = ...*1.1;
G1: -> prot1;
```

The reaction rate for the production of `'prot1'` will be equal to $((0+1.2)+0.3)*0.8 + ((0+1.2)*1.1)$. If you set the reaction rate of G1 without using an ellipsis, but include it in multiple strands, its reaction rate will be a multiple of the number of strands it is a part of. For example, if you set the reaction rate of G1 above to `"k1*S1"`, and include it in two strands, the net reaction rate will be $k1*S1 + k1*S1$.

The purpose of prepending or postfixing a `'--'` to a strand is to declare that the strand in question is designed to have DNA attached to it at that end. If exactly one DNA strand is defined with an upstream `'--'` in its definition in a submodule, the name of that module may be used as a proxy for that strand when creating attaching something upstream of it, and visa versa with a defined downstream `'--'` in its definition:

```
model twostrands
  --P1--RBS1--G1
  P2--RBS2--G2--
end

model long
  A: twostrands();
  P3--A
  A--G3
end
```

The module `'long'` will have two strands: `P3-A.P1-A.RBS1-A.G1` and `A.P2-A.RBS2-A.G2-G3`.

Submodule strands intended to be used in the middle of other strands should be defined with `'--'` both upstream and downstream of the strand in question:

```
model oneexported
  --P1--RBS1--G1--
  P2--RBS2--G2
end

model full
  A: oneexported()
  P2--A--stop
end
```

If multiple strands are defined with upstream or downstream `"--"` marks, it is illegal to use the name of the module containing them as proxy.

5.7.13 Interactions

Some species act as activators or repressors of reactions that they do not actively participate in. Typical models do not bother mentioning this explicitly, as it will show up in the reaction rates. However, for visualization purposes and/or for cases where the reaction rates might not be known explicitly, you may declare these interactions using the same format as reactions, using different symbols instead of \rightarrow : for activations, use $\rightarrow o$; for inhibitions, use $\rightarrow |$, and for unknown interactions or for interactions which sometimes activate and sometimes inhibit, use \rightarrow (:

```
J0: S1 + E -> SE;
i1: S2 -| J0;
i2: S3 -o J0;
i3: S4 - ( J0;
```

If a reaction rate is given for the reaction in question, that reaction must include the species listed as interacting with that reaction. This, then, is legal:

```
J0: S1 + E -> SE; k1*S1*E/S2
i1: S2 -| J0;
```

because the species S2 is present in the formula $k1*S1*E/S2$. If the concentration of an inhibitory species increases, it should decrease the reaction rate of the reaction it inhibits, and vice versa for activating species. The current version of libAntimony (v2.4) does not check this, but future versions may add the check.

When the reaction rate is not known, species from interactions will be added to the SBML 'listOfModifiers' for the reaction in question. Normally, the kinetic law is parsed by libAntimony and any species there are added to the list of modifiers automatically, but if there is no kinetic law to parse, this is how to add species to that list.

5.7.14 Other files

More than one file may be used to define a set of modules in Antimony through the use of the 'import' keyword. At any point in the file outside of a module definition, use the word `import` followed by the name of the file in quotation marks, and Antimony will include the modules defined in that file as if they had been cut and pasted into your file at that point. SBML files may also be included in this way:

```
import "models1.txt"
import "oscli.xml"

model mod2()
  A: mod1();
  B: oscli();
end
```

In this example, the file 'models1.txt' is an Antimony file that defines the module 'mod1', and the file 'oscli.xml' is an SBML file that defines a model named 'oscli'. The Antimony module 'mod2' may then use modules from either or both of the other imported files.

Remember that imported files act like they were cut and pasted into the main file. As such, any bare declarations in the main file and in the imported files will all contribute to the default '___main' module. Most SBML files will not contribute to this module, unless the name of the model in the file is `___main` (for example, if it was created by the antimony converter).

By default, libantimony will examine the 'import' text to determine whether it is a relative or absolute filename, and, if relative, will prepend the directory of the working file to the import text before attempting to load the file. If it cannot find it there, it is possible to tell the libantimony API to look in different directories for files loaded from import statements.

However, if the working directory contains a `.antimony` file, or if one of the named directories contains a `.antimony` file, import statements can be subverted. Each line of this file must contain three tab-delimited strings: the name of the file which contains an import statement, the text of the import statement, and the filename where the program should look for the file. Thus, if a file `file1.txt` contains the line `import "file2.txt"`, and a `.antimony` file is discovered with the line:

```
file1.txt      file2.txt      antimony/import/file2.txt
```

The library will attempt to load `'antimony/import/file2.txt'` instead of looking for `'file2.txt'` directly. For creating files in-memory or when reading antimony models from strings, the first string may be left out:

```
file2.txt      antimony/import/file2.txt
```

The first and third entries may be relative filenames: the directory of the `.antimony` file itself will be added internally when determining the file's actual location. The second entry must be exactly as it appears in the first file's `'import'` directive, between the quotation marks.

5.7.15 Importing and Exporting Antimony Models

Once you have created an Antimony file, you can convert it to SBML or CellML using `'sbtranslate'` or the `'QTAntimony'` visual editor (both available from <http://antimony.sourceforge.net/>) This will convert each of the models defined in the Antimony text file into a separate SBML model, including the overall `'__main'` module (if it contains anything). These files can then be used for simulation or visualization in other programs.

QTAntimony can be used to edit and translate Antimony, SBML, and CellML models. Any file in those three formats can be opened, and from the `'View'` menu, you can turn on or off the SBML and CellML tabs. Select the tabs to translate and view the working model in those different formats.

The SBML tabs can additionally be configured to use the `'Hierarchical Model Composition'` package constructs. Select `'Edit/Flatten SBML tab(s)'` or hit control-F to toggle between this version and the old `'flattened'` version of SBML. (To enable this feature if you compile Antimony yourself, you will need the latest versions of libSBML with the SBML `'comp'` package enabled, and to select `'WITH_COMP_SBML'` from the CMake menu.)

As there were now several different file formats available for translation, the old command-line translators still exist (`antimony2sbml`; `sbml2antimony`), but have been supplanted by the new `'sbtranslate'` executable. Instructions for use are available by running `sbtranslate` from the command line, but in brief: any number of files to translate may be added to the command line, and the desired output format is given with the `'-o'` flag: `-o antimony`, `-o sbml`, `-o cellml`, or `-o sbml-comp` (the last to output files with the SBML `'comp'` package constructs).

Examples:

```
sbtranslate modell1.txt modell2.txt -o sbml
```

will create one flattened SBML file for the main model in the two Antimony files in the working directory. Each file will be of the format `"[prefix].xml"`, where `[prefix]` is the original filename with `'txt'` removed (if present).

```
sbtranslate oscli.xml ffn.xml -o antimony
```

will output two files in the working directory: `'oscli.txt'` and `'ffn.txt'` (in the antimony format).

```
sbtranslate modell1.txt -o sbml-comp
```

will output `'modell1.xml'` in the working directory, containing all models in the `'modell1.txt'` file, using the SBML `'comp'` package.

5.7.16 Appendix: Converting between SBML and Antimony

For reference, here are some of the differences you will see when converting models between SBML and Antimony:

- Local parameters in SBML reactions become global parameters in Antimony, with the reaction name prepended. If a different symbol already has the new name, a number is appended to the variable name so it will be unique. These do not get converted back to local parameters when converting Antimony back to SBML.
- Algebraic rules in SBML disappear in Antimony.
- Any element with both a value (or an initial amount/concentration for species) and an initial assignment in SBML will have only the initial assignment in Antimony.
- Stoichiometry math in SBML disappears in Antimony.
- All `constant=true` species in SBML are set `const` in Antimony, even if that same species is set `boundary=false`.
- All `boundary=true` species in SBML are set `const` in Antimony, even if that same species is set `constant=false`.
- Boundary ('const') species in Antimony are set `boundary=true` and `constant=false` in SBML.
- Variable ('var') species in Antimony are set `boundary=false` and `constant=false` in SBML.
- Modules in Antimony are flattened in SBML (unless you use the `comp` option).
- DNA strands in Antimony disappear in SBML.
- DNA elements in Antimony no longer retain the ellipses syntax in SBML, but the effective reaction rates and assignment rules should be accurate, even for elements appearing in multiple DNA strands. These reaction rates and assignment rules will be the sum of the rate at all duplicate elements within the DNA strands.
- Any symbol with the MathML csymbol 'time' in SBML becomes 'time' in Antimony.
- Any formula with the symbol 'time' in it in Antimony will become the MathML csymbol 'time' in SBML.
- The MathML csymbol 'delay' in SBML disappears in Antimony.
- Any SBML version 2 level 1 function with the MathML csymbol 'time' in it will become a local variable with the name 'time_ref' in Antimony. This 'time_ref' is added to the function's interface (as the last in the list of symbols), and any uses of the function are modified to use 'time' in the call. In other words, a function "function(x, y): x+y*time" becomes "function(x, y, time_ref): x + y*time_ref", and formulas that use "function(A, B)" become "function(A, B, time)"
- A variety of Antimony keywords, if found in SBML models as IDs, are renamed to add an appended '_'. So the ID `compartment` becomes `compartment_`, `model` becomes `model_`, etc.

5.7.17 Further Reading

- Lucian Smith's [example models](#) show how to use the `comp` package.
- [Antimony's manual](#) in PDF format.

Parameter Scan

ParameterScan is a package for Tellurium that provides a different way to run simulations and plot graphs than given in the standard Tellurium library. While the standard syntax in Tellurium asks you to put parameters, such as the amount of time for the simulation to run, as arguments in the simulation function, ParameterScan allows you to set these values before calling the function. This is especially useful for more complicated 3D plots that often take many arguments to customize. Therefore, ParameterScan also provides several plotting options that would be cumbersome to handle using the traditional approach. This tutorial will go over how to use all of ParameterScan's functionality.

6.1 Loading a Model

To use ParameterScan you will need to import it. The easiest way to do this is the command 'from tellurium import ParameterScan' after the standard 'import tellurium as te'.

```
import tellurium as te
from tellurium import ParameterScan

cell = '''
    $Xo -> S1; vo;
    S1 -> S2; k1*S1 - k2*S2;
    S2 -> $X1; k3*S2;

    vo = 1
    k1 = 2; k2 = 0; k3 = 3;
'''

rr = te.loadAntimonyModel(cell)
p = ParameterScan(rr)

p.startTime = 0
p.endTime = 20
p.numberOfPoints = 50
p.width = 2
p.xlabel = 'Time'
```

(continues on next page)

(continued from previous page)

```
p.ylabel = 'Concentration'
p.title = 'Cell'

p.plotArray()
```

After you load the model, you can use it to create an object of `ParameterScan`. This allows you to set many different parameters and easily change them.

6.2 Class Methods

These are the methods of the `ParameterScan` class.

`plotArray()`

The `plotArray()` method works much the same as `te.plotArray()`, but with some increased functionality. It automatically runs a simulation based on the given parameters, and plots a graph of the results. Accepted parameters are `startTime`, `endTime`, `numberOfPoints`, `width`, `color`, `xlabel`, `ylabel`, `title`, `integrator`, and `selection`.

`plotGraduatedArray()`

This method runs several simulations, each one with a slightly higher starting concentration of a chosen species than the last. It then plots the results. Accepted parameters are `startTime`, `endTime`, `value`, `startValue`, `endValue`, `numberOfPoints`, `width`, `color`, `xlabel`, `ylabel`, `title`, `integrator`, and `polyNumber`.

`plotPolyArray()`

This method runs the same simulation as `plotGraduatedArray()`, but plots each graph as a polygon in 3D space. Accepted parameters are `startTime`, `endTime`, `value`, `startValue`, `endValue`, `numberOfPoints`, `color`, `alpha`, `xlabel`, `ylabel`, `title`, `integrator`, and `polyNumber`.

`plotMultiArray(param1, param1Range, param2, param2Range)`

This method plots a grid of arrays with different starting conditions. It is the only method in `ParameterScan` that takes arguments when it is called. The two `param` arguments specify the species or constant that should be set, and the `range` arguments give the different values that should be simulated with. The resulting grid is an array for each possible combination of the two ranges. For instance, `plotMultiArray('k1', [1, 2], 'k2', [1, 2, 3])` would result in a grid of six arrays, one with `k1 = 1` and `k2 = 1`, the next with `k1 = 1` and `k2 = 2`, and so on. The advantage of this method is that you can plot multiple species in each array. Accepted parameters are `startTime`, `endTime`, `numberOfPoints`, `width`, `title`, and `integrator`.

`plotSurface()`

This method produces a color-coded 3D surface based on the concentration of one species and the variation of two factors (usually time and an equilibrium constant). Accepted parameters are `startTime`, `endTime`, `numberOfPoints`, `startValue`, `endValue`, `independent`, `dependent`, `color`, `xlabel`, `ylabel`, `title`, `integrator`, `colormap`, `colorbar`, and `antialias`.

`plot2DParameterScan(p1, p1Range, p2, p2Range)`

Create a 2D parameter scan and plots the results. The two parameters `p1`, `p2` are the id of the first and second parameter. `p1Range` and `p2Range` are the range of the first and second parameter respectively. Parameters `start` and `end` represents the starting time and ending time, and `points` is the number of points to simulate. Will generate a 2D plot containing multiple graphs of the given parameters at various values in the given ranges.

`createColormap(color1, color2)`

This method allows you to create a custom colormap for `plotSurface()`. It returns a colormap that stretches between `color1` and `color2`. Colors can be input as RGB tuple lists (i.e. `[0.5, 0, 1]`), or as strings with either a standard color

name or a hex value. The first color becomes bottom of the colormap (i.e. lowest values in `plotArray()`) and the second becomes the top.

```
createColorPoints()
```

This method creates a color list (i.e. sets 'color') spanning the range of a colormap. The colormap can either be predefined or user-defined with `createColormap()`. This set of colors will then be applied to arrays, including `plotArray()`, `plotPolyArray()`, and `plotGraduatedArray()`. Note: This method gets the number of colors to generate from the `polyNumber` parameter. If using it with `plotArray()` or `plotGraduatedArray()`, setting this parameter to the number of graphs you are expecting will obtain better results.

6.3 Example

Let's say that we want to examine how the value of a rate constant (parameter) affects how the concentration of a species changes over time. There are several ways this can be done, but the simplest is to use `plotGraduatedArray()`. Here is an example script:

```
import tellurium as te
from tellurium import ParameterScan

r = te.loada('''
    $Xo -> S1; vo;
    S1 -> S2; k1*S1 - k2*S2;
    S2 -> $X1; k3*S2;

    vo = 1
    k1 = 2; k2 = 0; k3 = 3;
''')

p = ParameterScan(r)

p.endTime = 6
p.numberOfPoints = 100
p.polyNumber = 5
p.startValue = 1
p.endValue = 5
p.value = 'k1'
p.selection = ['S1']

p.plotGraduatedArray()
```

Another way is to use `createColormap()` and `plotSurface()` to create a 3D graph of the same model as above. After loading the model, we can use this code:

```
p.endTime = 6
p.colormap = p.createColormap([.12, .56, 1], [.86, .08, .23])
p.dependent = 'S1'
p.independent = ['time', 'k1']
p.startValue = 1
p.endValue = 5
p.numberOfPoints = 100

p.plotSurface()
```

6.3.1 Perform Parameter Scan

plotPolyArray() example

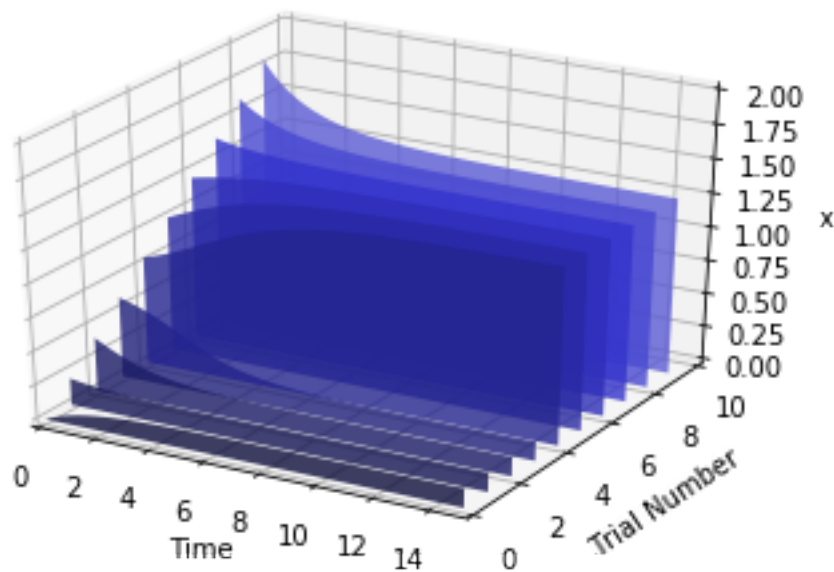
```
import tellurium as te

r = te.loada('''
    J1: $Xo -> x; 0.1 + k1*x^4/(k2+x^4);
    x -> $w; k3*x;

    k1 = 0.9;
    k2 = 0.3;
    k3 = 0.7;
    x = 0;
''')

# parameter scan
p = te.ParameterScan(r,
    # settings
    startTime = 0,
    endTime = 15,
    numberOfPoints = 50,
    polyNumber = 10,
    endValue = 1.8,
    alpha = 0.8,
    value = "x",
    selection = "x",
    color = ['#0F0F3D', '#141452', '#1A1A66', '#1F1F7A', '#24248F', '#2929A3',
            '#2E2EB8', '#3333CC', '#4747D1', '#5C5CD6']
)

# plot
p.plotPolyArray()
```



plotSurface() example

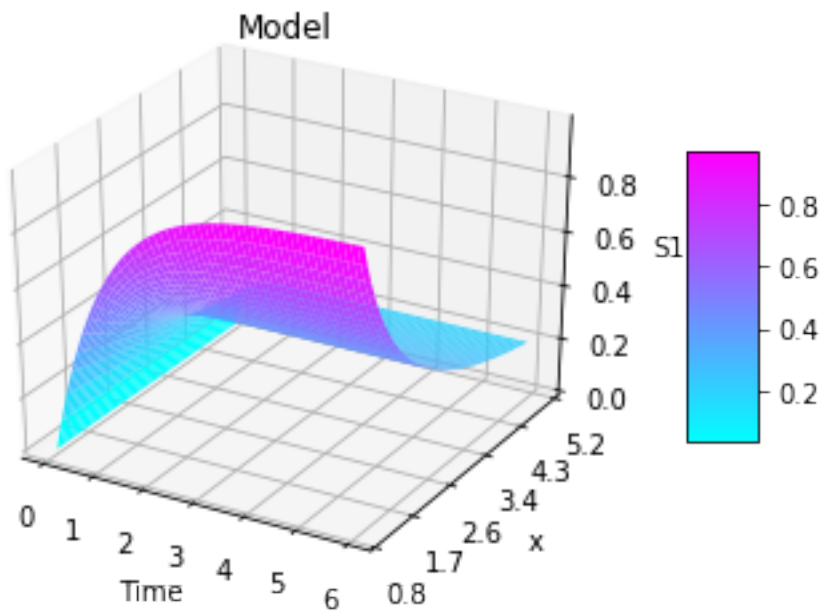
```

r = te.loada('''
    $Xo -> S1; vo;
    S1 -> S2; k1*S1 - k2*S2;
    S2 -> $X1; k3*S2;

    vo = 1
    k1 = 2; k2 = 0; k3 = 3;
''')

# parameter scan
p = te.ParameterScan(r,
    # settings
    startTime = 0,
    endTime = 6,
    numberOfPoints = 50,
    startValue = 1,
    endValue = 5,
    colormap = "cool",
    independent = ["Time", "k1"],
    dependent = "S1",
    xlabel = "Time",
    ylabel = "x",
    title = "Model"
)
# plot
p.plotSurface()

```



```

import warnings
warnings.filterwarnings("ignore")

import tellurium as te
from tellurium.analysis.parameterscan import plot2DParameterScan

# model definitions
r = te.loada("""

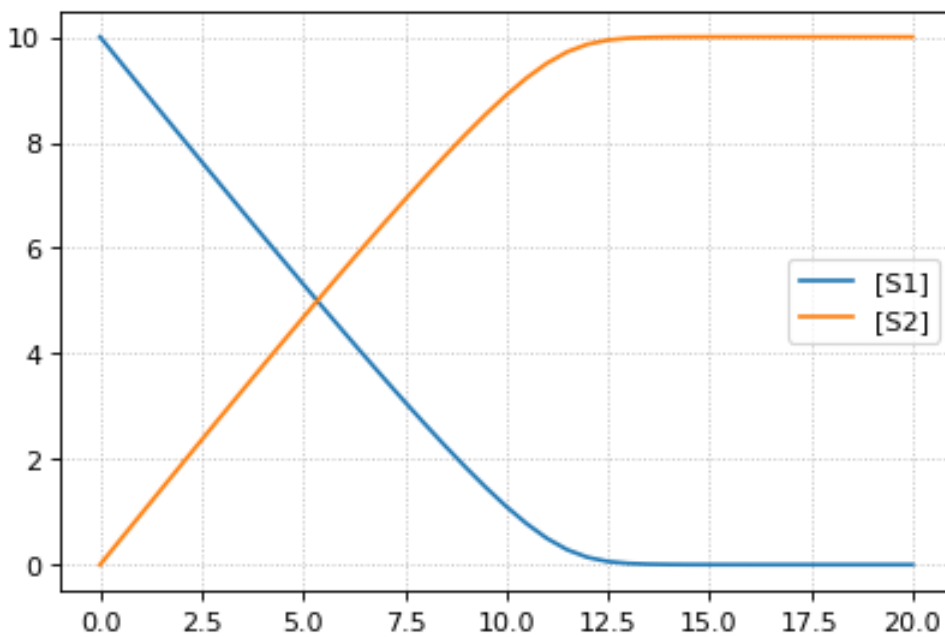
```

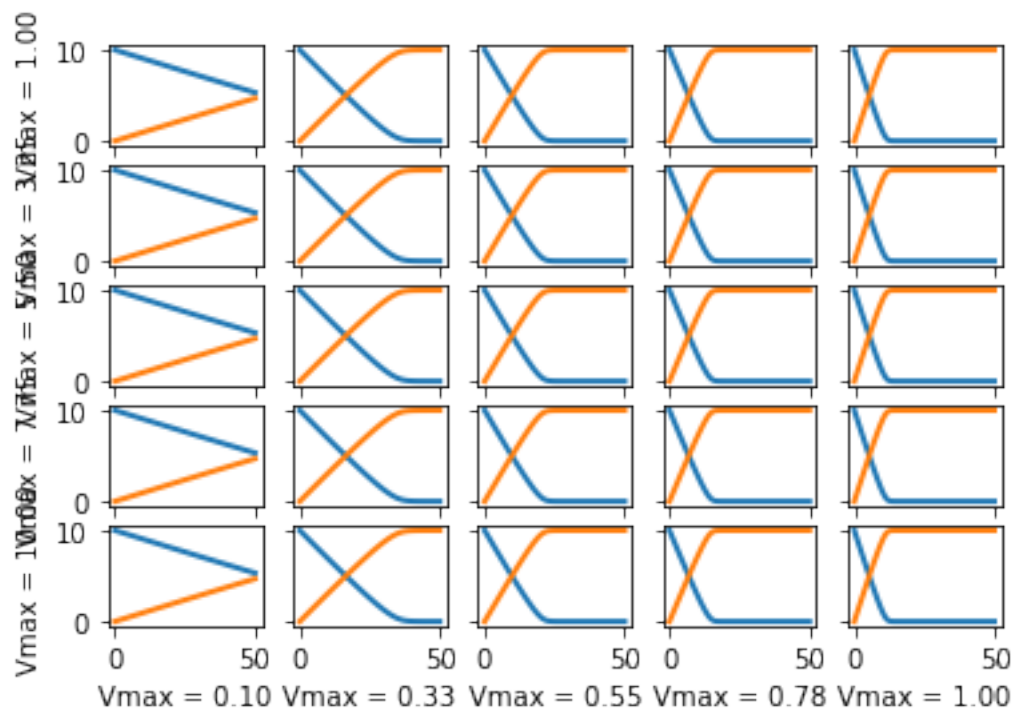
(continues on next page)

(continued from previous page)

```
model test
  J0: S1 -> S2; Vmax * (S1/(Km+S1))
  S1 = 10; S2 = 0;
  Vmax = 1; Km = 0.5;
end
"""
s = r.simulate(0, 20, 41)
r.plot(s)

import numpy as np
plot2DParameterScan(r,
                    p1='Vmax', p1Range=np.linspace(1, 10, num=5),
                    p2='Vmax', p2Range=np.linspace(0.1, 1.0, num=5),
                    start=0, end=50, points=101)
```





6.4 Properties

These are the properties of the `ParameterScan` class.

`alpha`: Sets opaqueness of polygons in `plotPolyArray()`. Should be a number from 0-1. Set to 0.7 by default.

`color`: Sets color for use in all plotting functions except `plotSurface()` and `plotMultiArray()`. Should be a list of at least one string. All legal HTML color names are accepted. Additionally, for `plotArray()` and `plotGraduatedArray()`, this parameter can determine the appearance of the line as according to PyPlot definitions. For example, setting color to `['ro']` would produce a graph of red circles. For examples on types of lines in PyPlot, go to http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot. If there are more graphs than provided color selections, subsequent graphs will start back from the beginning of the list.

`colorbar`: True shows a color legend for `plotSurface()`, False does not. Set to True by default.

`colormap`: The name of the colormap you want to use for `plotSurface()`. Legal names can be found at http://matplotlib.org/examples/color/colormaps_reference.html and should be strings. Alternatively, you can create a custom colormap using the `createColorMap` method.

`dependent`: The dependent variable for `plotSurface()`. Should be a string of a valid species.

`endTime`: When the simulation ends. Default is 20.

`endValue`: For `plotGraduatedArray()`, assigns the final value of the independent variable other than time. For `plotPolyArray()` and `plotSurface()` assigns the final value of the parameter being varied. Should be a string of a valid parameter.

`independent`: The independent variable for `plotSurface()`. Should be a list of two strings: one for time, and one for a parameter.

`integrator`: The integrator used to calculate results for all plotting methods. Set to `'ccode'` by default, but another option is `'gillespie'`.

legend: A bool that determines whether a legend is displayed for `plotArray()`, `plotGraduatedArray()`, and `plotMultiArray()`. Default is `True`.

numberOfPoints: Number of points in simulation results. Default is 50. Should be an integer.

polyNumber: The number of graphs for `plotGraduatedArray()` and `plotPolyArray()`. Default is 10.

rr: A pointer to a loaded RoadRunner model. `ParameterScan()` takes it as its only argument.

selection: The species to be shown in the graph in `plotArray()` and `plotMultiArray()`. Should be a list of at least one string.

sameColor: Set this to `True` to force `plotGraduatedArray()` to be all in one color. Default color is blue, but another color can be chosen via the “color” parameter. Set to `False` by default.

startTime: When the simulation begins. Default is 0.

startValue: For `plotGraduatedArray()`, assigns the beginning value of the independent variable other than time. For `plotPolyArray()` and `plotSurface()` assigns the beginning value of the parameter being varied. Default is whatever the value is in the loaded model, or if not specified there, 0.

title: Default is no title. If set to a string, it will display above any of the plotting methods.

value: The item to be varied between graphs in `plotGraduatedArray()` and `plotPolyArray()`. Should be a string of a valid species or parameter.

width: Sets line width in `plotArray()`, `plotGraduatedArray()`, and `plotMultiArray()`. Won’t have any effect on special line types (see color). Default is 2.5.

xlabel: Sets a title for the x-axis. Should be a string. Not setting it results in an appropriate default; to create a graph with no title for the x-axis, set it to `None`.

ylabel: Sets a title for the y-axis. Should be a string. Not setting it results in an appropriate default; to create a graph with no title for the x-axis, set it to `None`.

zlabel: Sets a title for the z-axis. Should be a string. Not setting it results in an appropriate default; to create a graph with no title for the x-axis, set it to `None`.

6.5 SteadyStateScan

This class is part of `ParameterScan` but provides some added functionality. It allows the user to plot graphs of the steady state values of one or more species as dependent on the changing value of an equilibrium constant on the x-axis. To use it, use the same import statement as before: `from tellurium import SteadyStateScan`. Then, you can use `SteadyStateScan` on a loaded model by using `ss = SteadyStateScan(rr)`. Right now, the only working method is `plotArray()`, which needs the parameters of `value`, `startValue`, `endValue`, `numberOfPoints`, and `selection`. The parameter `‘value’` refers to the equilibrium constant, and should be the string of the chosen constant. The start and end value parameters are numbers that determine the domain of the x-axis. The `‘numberOfPoints’` parameter refers to the number of data points (i.e. a larger value gives a smoother graph) and `‘selection’` is a list of strings of one or more species that you would like in the graph.

6.5.1 Steady state scan

Using `te.ParameterScan.SteadyStateScan` for scanning the steady state.

```
import tellurium as te
import matplotlib.pyplot as plt
import tellurium as te
import numpy as np
```

(continues on next page)

(continued from previous page)

```

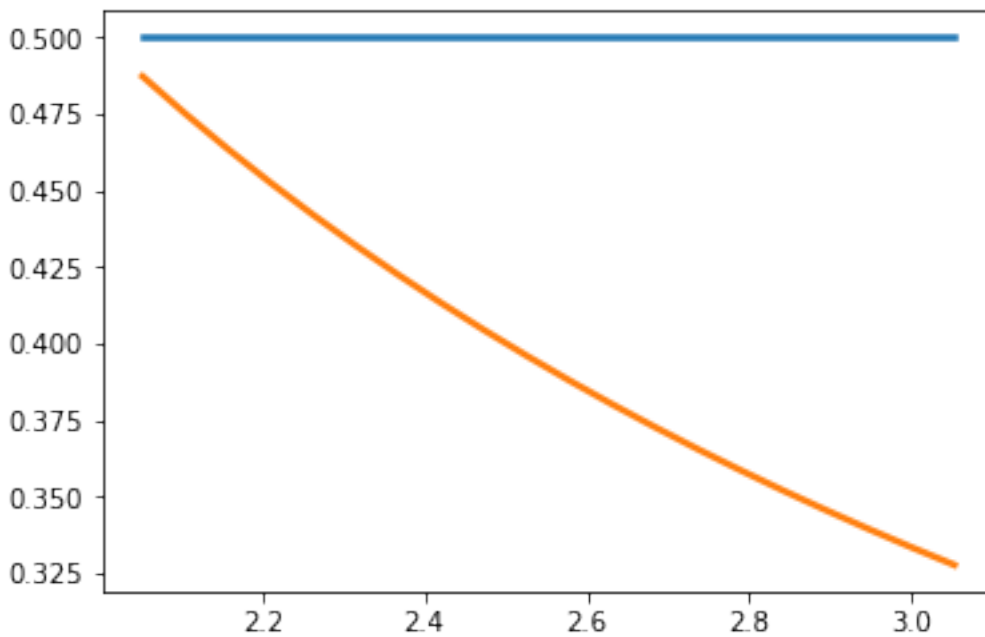
from roadrunner import Config

r = te.loada('''
  $Xo -> S1; vo;
  S1 -> S2; k1*S1 - k2*S2;
  S2 -> $X1; k3*S2;

  vo = 1
  k1 = 2; k2 = 0; k3 = 3;
''')

p = te.SteadyStateScan(r,
  value = 'k3',
  startValue = 2,
  endValue = 3,
  numberOfPoints = 20,
  selection = ['S1', 'S2']
)
p.plotArray()

```



```

array([[2.05263158, 0.5, 0.48717949],
       [2.10526316, 0.5, 0.475],
       [2.15789474, 0.5, 0.46341463],
       [2.21052632, 0.5, 0.45238095],
       [2.26315789, 0.5, 0.44186047],
       [2.31578947, 0.5, 0.43181818],
       [2.36842105, 0.5, 0.42222222],
       [2.42105263, 0.5, 0.41304348],
       [2.47368421, 0.5, 0.40425532],
       [2.52631579, 0.5, 0.39583333],
       [2.57894737, 0.5, 0.3877551],
       [2.63157895, 0.5, 0.38],
       [2.68421053, 0.5, 0.37254902],

```

(continues on next page)

(continued from previous page)

```
[2.73684211, 0.5      , 0.36538462],  
[2.78947368, 0.5      , 0.35849057],  
[2.84210526, 0.5      , 0.35185185],  
[2.89473684, 0.5      , 0.34545455],  
[2.94736842, 0.5      , 0.33928571],  
[3.         , 0.5      , 0.33333333],  
[3.05263158, 0.5      , 0.32758621]])
```


7.1 Installing Packages

Tellurium provides utility methods for installing Python packages from [PyPI](#). These methods simply delegate to `pip`, and are usually more reliable than running `!pip install xyz`.

`tellurium.installPackage(name)`

Install pip package. This has the advantage you don't have to manually track down the currently running Python interpreter and switch to the command line (useful e.g. in the Tellurium notebook viewer).

Parameters `name` (*str*) – package name

`tellurium.upgradePackage(name)`

Upgrade pip package.

Parameters `name` (*str*) – package name

`tellurium.uninstallPackage(name)`

Uninstall pip package.

Parameters `name` (*str*) – package name

`tellurium.searchPackage(name)`

Search pip package for package name.

Parameters `name` (*str*) – package name

7.1.1 How to install additional packages

If you are using Tellurium notebook or Tellurium Spyder, you can install additional package using `installPackage` function. In Tellurium Spyder, you can also install packages using included command Prompt. For more information, see [Running Command Prompt for Tellurium Spyder](#).

```
import tellurium as te
# install cobra (https://github.com/opencobra/cobrapy)
te.installPackage('cobra')
# update cobra to latest version
te.upgradePackage('cobra')
# remove cobra
# te.removePackage('cobra')
```

7.2 Utility Methods

The most useful methods here are the notices routines. Roadrunner will often issue warning or informational messages. For repeated simulation such messages will clutter up the outputs. `noticesOff` and `noticesOn` can be used to turn on and off the messages.

`tellurium.getVersionInfo()`

Returns version information for tellurium included packages.

Returns list of tuples (package, version)

`tellurium.printVersionInfo()`

Prints version information for tellurium included packages.

see also: `getVersionInfo()`

`tellurium.getTelluriumVersion()`

Version number of tellurium.

Returns version

Return type str

`tellurium.noticesOff()`

Switch off the generation of notices to the user. Call this to stop roadrunner from printing warning message to the console.

See also `noticesOn()`

`tellurium.noticesOn()`

Switch on notice generation to the user.

See also `noticesOff()`

`tellurium.saveToFile(filePath, str)`

Save string to file.

see also: `readFromFile()`

Parameters

- **filePath** – file path to save to
- **str** – string to save

`tellurium.readFromFile(filePath)`

Load a file and return contents as a string.

see also: `saveToFile()`

Parameters **filePath** – file path to read from

Returns string representation of the contents of the file

Tellurium's version can be obtained via `te.__version__`. `.printVersionInfo()` also returns information from certain constituent packages.

```
import tellurium as te

# to get the tellurium version use
print('te.__version__')
print(te.__version__)
# or
print('te.getTelluriumVersion()')
print(te.getTelluriumVersion())

# to print the full version info use
print('-' * 80)
te.printVersionInfo()
print('-' * 80)
```

```
te.__version__
2.1.0
te.getTelluriumVersion()
2.1.0
-----
tellurium : 2.1.0
roadrunner : 1.5.1
antimony : 2.9.4
libsbml : 5.15.0
libsedml : 0.4.3
phrasedml : 1.0.9
-----
```

```
from builtins import range
# Load SBML file
r = te.loada("""
model test
    J0: X0 -> X1; k1*X0;
    X0 = 10; X1=0;
    k1 = 0.2
end
""")

import matplotlib.pyplot as plt

# Turn off notices so they don't clutter the output
te.noticesOff()
for i in range(0, 20):
    result = r.simulate (0, 10)
    r.reset()
    r.plot(result, loc=None, show=False,
           linewidth=2.0, linestyle='-', color='black', alpha=0.8)
    r.k1 = r.k1 + 0.2
# Turn the notices back on
te.noticesOn()
```

```
# create tmp file
import tempfile
ftmp = tempfile.NamedTemporaryFile(suffix=".xml")
# load model
```

(continues on next page)

(continued from previous page)

```
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
# save to file
te.saveToFile(ftmp.name, r.getMatlab())
```

```
# or easier via
r.exportToMatlab(ftmp.name)
```

```
# load file
sbmlstr = te.readFromFile(ftmp.name)
print('%' + '*'*80)
print('Converted MATLAB code')
print('%' + '*'*80)
print(sbmlstr)
```

```
*****
Converted MATLAB code
*****
% How to use:
%
% __main takes 3 inputs and returns 3 outputs.
%
% [t x rInfo] = __main(tspan,solver,options)
% INPUTS:
% tspan - the time vector for the simulation. It can contain every time point,
% or just the start and end (e.g. [0 1 2 3] or [0 100]).
% solver - the function handle for the odeN solver you wish to use (e.g. @ode23s).
% options - this is the options structure returned from the MATLAB odeset
% function used for setting tolerances and other parameters for the solver.
%
% OUTPUTS:
% t - the time vector that corresponds with the solution. If tspan only contains
% the start and end times, t will contain points spaced out by the solver.
% x - the simulation results.
% rInfo - a structure containing information about the model. The fields
% within rInfo are:
%   stoich - the stoichiometry matrix of the model
%   floatingSpecies - a cell array containing floating species name, initial
%   value, and indicator of the units being in concentration or amount
%   compartments - a cell array containing compartment names and volumes
%   params - a cell array containing parameter names and values
%   boundarySpecies - a cell array containing boundary species name, initial
%   value, and indicator of the units being in concentration or amount
%   rateRules - a cell array containing the names of variables used in a rate rule
%
% Sample function call:
%   options = odeset('RelTol',1e-12,'AbsTol',1e-9);
%   [t x rInfo] = __main(linspace(0,100,100),@ode23s,options);
%
function [t x rInfo] = __main(tspan,solver,options)
    % initial conditions
    [x rInfo] = model();

    % initial assignments

    % assignment rules
```

(continues on next page)

(continued from previous page)

```

% run simulation
[t x] = feval(solver,@model,tspan,x,options);

% assignment rules

function [xdot rInfo] = model(time,x)
% x(1)      S1
% x(2)      S2

% List of Compartments
vol__default_compartment = 1;           %default_compartment

% Global Parameters
rInfo.g_p1 = 0.1;           % k1

if (nargin == 0)

    % set initial conditions
    xdot(1) = 10*vol__default_compartment;           % S1 = S1 [Concentration]
    xdot(2) = 0*vol__default_compartment;           % S2 = S2 [Concentration]

    % reaction info structure
    rInfo.stoich = [
        -1
        1
    ];

    rInfo.floatingSpecies = {           % Each row: [Species Name, Initial Value,
    ↪isAmount (1 for amount, 0 for concentration)]
        'S1' , 10, 0
        'S2' , 0, 0
    };

    rInfo.compartments = {           % Each row: [Compartment Name, Value]
        'default_compartment' , 1
    };

    rInfo.params = {           % Each row: [Parameter Name, Value]
        'k1' , 0.1
    };

    rInfo.boundarySpecies = {           % Each row: [Species Name, Initial Value,
    ↪isAmount (1 for amount, 0 for concentration)]
    };

    rInfo.rateRules = {           % List of variables involved in a rate rule
    };

else

    % calculate rates of change
    R0 = rInfo.g_p1*(x(1));

    xdot = [
        - R0
        + R0
    ];

```

(continues on next page)

(continued from previous page)

```
end;

%listOfSupportedFunctions
function z = pow (x,y)
    z = x^y;

function z = sqr (x)
    z = x*x;

function z = piecewise(varargin)
    numArgs = nargin;
    result = 0;
    foundResult = 0;
    for k=1:2: numArgs-1
        if varargin{k+1} == 1
            result = varargin{k};
            foundResult = 1;
            break;
        end
    end
    if foundResult == 0
        result = varargin{numArgs};
    end
    z = result;

function z = gt(a,b)
    if a > b
        z = 1;
    else
        z = 0;
    end

function z = lt(a,b)
    if a < b
        z = 1;
    else
        z = 0;
    end

function z = geq(a,b)
    if a >= b
        z = 1;
    else
        z = 0;
    end

function z = leq(a,b)
    if a <= b
        z = 1;
    else
```

(continues on next page)

(continued from previous page)

```

    z = 0;
end

function z = neq(a,b)
    if a ~= b
        z = 1;
    else
        z = 0;
    end

function z = and(varargin)
    result = 1;
    for k=1:nargin
        if varargin{k} ~= 1
            result = 0;
            break;
        end
    end
    z = result;

function z = or(varargin)
    result = 0;
    for k=1:nargin
        if varargin{k} ~= 0
            result = 1;
            break;
        end
    end
    z = result;

function z = xor(varargin)
    foundZero = 0;
    foundOne = 0;
    for k = 1:nargin
        if varargin{k} == 0
            foundZero = 1;
        else
            foundOne = 1;
        end
    end
    if foundZero && foundOne
        z = 1;
    else
        z = 0;
    end

function z = not(a)
    if a == 1
        z = 0;
    else
        z = 1;

```

(continues on next page)

(continued from previous page)

```
end

function z = root(a,b)
    z = a^(1/b);
```

7.3 Model Loading

There are a variety of methods to load models into libRoadrunner.

`tellurium.loada(ant)`
Load model from Antimony string.

See also: `loadAntimonyModel()`

```
r = te.loada('S1 -> S2; k1*S1; k1=0.1; S1=10.0; S2 = 0.0')
```

Parameters `ant` (*str* | *file*) – Antimony model

Returns RoadRunner instance with model loaded

Return type `roadrunner.ExtendedRoadRunner`

`tellurium.loadAntimonyModel(ant)`
Load Antimony model with tellurium.

See also: `loada()`

```
r = te.loadAntimonyModel('S1 -> S2; k1*S1; k1=0.1; S1=10.0; S2 = 0.0')
```

Parameters `ant` (*str* | *file*) – Antimony model

Returns RoadRunner instance with model loaded

Return type `roadrunner.ExtendedRoadRunner`

`tellurium.loadSBMLModel(sbml)`
Load SBML model from a string or file.

Parameters `sbml` (*str* | *file*) – SBML model

Returns RoadRunner instance with model loaded

Return type `roadrunner.ExtendedRoadRunner`

`tellurium.loadCellMLModel(cellml)`
Load CellML model with tellurium.

Parameters `cellml` (*str* | *file*) – CellML model

Returns RoadRunner instance with model loaded

Return type `roadrunner.ExtendedRoadRunner`

Antimony files can be read with `te.loada` or `te.loadAntimonyModel`. For SBML `te.loadSBMLModel`, for CellML `te.loadCellMLModel` is used. All the functions accept either model strings or respective model files.


```

import tellurium as te

# Load an antimony model
ant_model = '''
    S1 -> S2; k1*S1;
    S2 -> S3; k2*S2;

    k1= 0.1; k2 = 0.2;
    S1 = 10; S2 = 0; S3 = 0;
'''

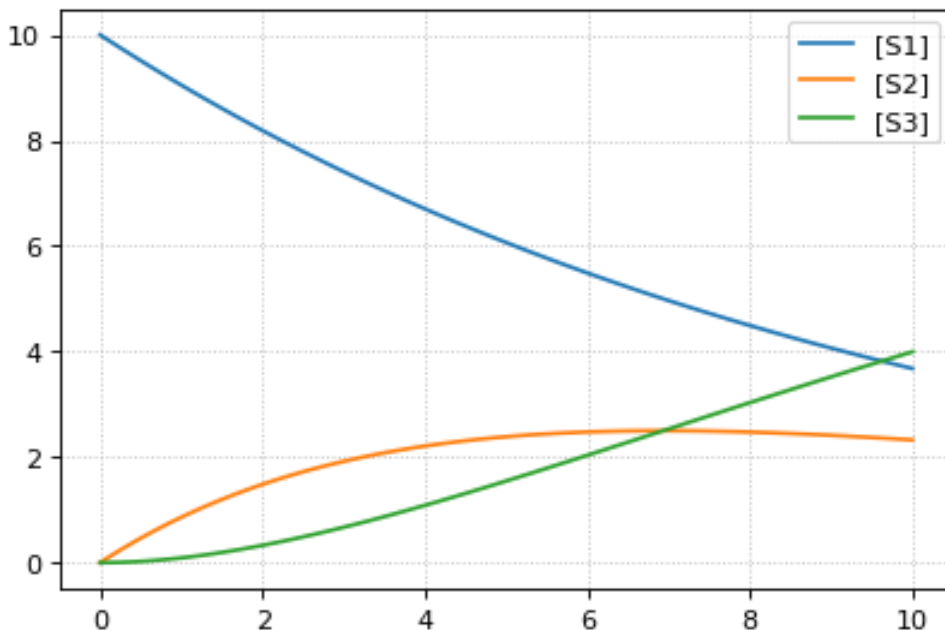
# At the most basic level one can load the SBML model directly using libRoadRunner
print('--- load using roadrunner ---')
import roadrunner
# convert to SBML model
sbml_model = te.antimonyToSBML(ant_model)
r = roadrunner.RoadRunner(sbml_model)
result = r.simulate(0, 10, 100)
r.plot(result)

# The method loada is simply a shortcut to loadAntimonyModel
print('--- load using tellurium ---')
r = te.loada(ant_model)
result = r.simulate(0, 10, 100)
r.plot(result)

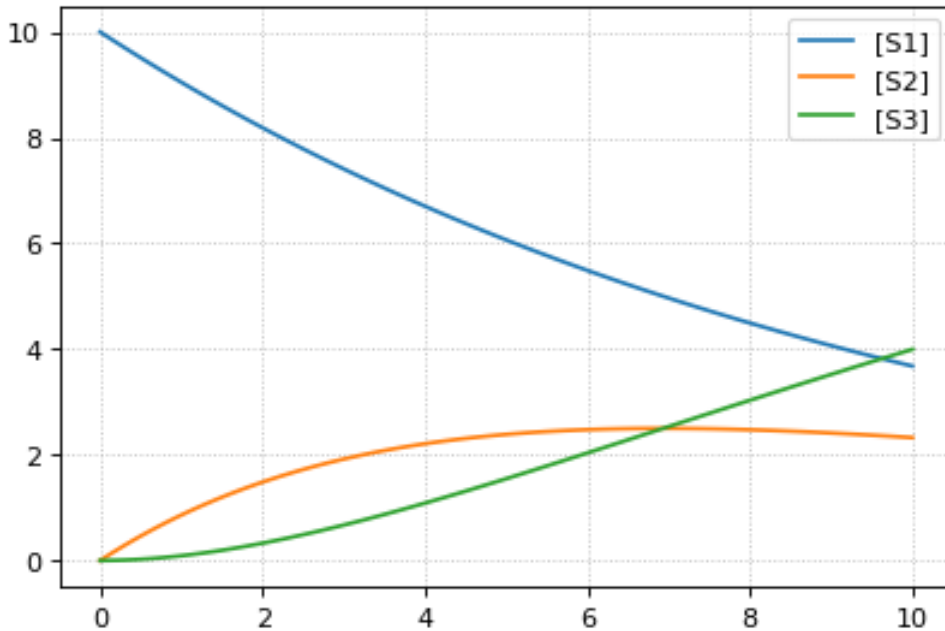
# same like
r = te.loadAntimonyModel(ant_model)

```

```
--- load using roadrunner ---
```



```
--- load using tellurium ---
```



7.4 Interconversion Utilities

Use these routines interconvert various standard formats

`tellurium.antimonyToSBML(ant)`

Convert Antimony to SBML string.

Parameters *ant* (*str* | *file*) – Antimony string or file

Returns SBML

Return type str

`tellurium.antimonyToCellML(ant)`

Convert Antimony to CellML string.

Parameters *ant* (*str* | *file*) – Antimony string or file

Returns CellML

Return type str

`tellurium.sbmlToAntimony(sbml)`

Convert SBML to antimony string.

Parameters *sbml* (*str* | *file*) – SBML string or file

Returns Antimony

Return type str

`tellurium.sbmlToCellML(sbml)`

Convert SBML to CellML string.

Parameters *sbml* (*str* | *file*) – SBML string or file

Returns CellML

Return type str

tellurium.**cellmlToAntimony**(*cellml*)

Convert CellML to antimony string.

Parameters *cellml* (*str* | *file*) – CellML string or file

Returns antimony

Return type str

tellurium.**cellmlToSBML**(*cellml*)

Convert CellML to SBML string.

Parameters *cellml* (*str* | *file*) – CellML string or file

Returns SBML

Return type str

Tellurium can convert between Antimony, SBML, and CellML.

```
import tellurium as te

# antimony model
ant_model = """
    S1 -> S2; k1*S1;
    S2 -> S3; k2*S2;

    k1= 0.1; k2 = 0.2;
    S1 = 10; S2 = 0; S3 = 0;
"""

# convert to SBML
sbml_model = te.antimonyToSBML(ant_model)
print('sbml_model')
print('*'*80)
# print first 10 lines
for line in list(sbml_model.splitlines())[:10]:
    print(line)
print('...')

# convert to CellML
cellml_model = te.antimonyToCellML(ant_model)
print('cellml_model (from Antimony)')
print('*'*80)
# print first 10 lines
for line in list(cellml_model.splitlines())[:10]:
    print(line)
print('...')

# or from the sbml
cellml_model = te.sbmlToCellML(sbml_model)
print('cellml_model (from SBML)')
print('*'*80)
# print first 10 lines
for line in list(cellml_model.splitlines())[:10]:
    print(line)
print('...')
```

sbml_model

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by libAntimony version v2.9.4 with libSBML version 5.15.0. -->
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3"
↳version="1">
  <model id="__main" name="__main">
    <listOfCompartments>
      <compartment sboTerm="SBO:0000410" id="default_compartment"
↳spatialDimensions="3" size="1" constant="true"/>
    </listOfCompartments>
    <listOfSpecies>
      <species id="S1" compartment="default_compartment"
↳initialConcentration="10" hasOnlySubstanceUnits="false"
↳boundaryCondition="false" constant="false"/>
      <species id="S2" compartment="default_compartment"
↳initialConcentration="0" hasOnlySubstanceUnits="false"
↳boundaryCondition="false" constant="false"/>
    ...
cellml_model (from Antimony)
*****
<?xml version="1.0"?>
<model xmlns:cellml="http://www.cellml.org/cellml/1.1#" xmlns="http://www.
↳cellml.org/cellml/1.1#" name="__main">
<component name="__main">
<variable initial_value="10" name="S1" units="dimensionless"/>
<variable initial_value="0" name="S2" units="dimensionless"/>
<variable initial_value="0.1" name="k1" units="dimensionless"/>
<variable name="_J0" units="dimensionless"/>
<variable initial_value="0" name="S3" units="dimensionless"/>
<variable initial_value="0.2" name="k2" units="dimensionless"/>
<variable name="_J1" units="dimensionless"/>
...
cellml_model (from SBML)
*****
<?xml version="1.0"?>
<model xmlns:cellml="http://www.cellml.org/cellml/1.1#" xmlns="http://www.
↳cellml.org/cellml/1.1#" name="__main">
<component name="__main">
<variable initial_value="10" name="S1" units="dimensionless"/>
<variable initial_value="0" name="S2" units="dimensionless"/>
<variable initial_value="0" name="S3" units="dimensionless"/>
<variable initial_value="0.1" name="k1" units="dimensionless"/>
<variable initial_value="0.2" name="k2" units="dimensionless"/>
<variable name="_J0" units="dimensionless"/>
<variable name="_J1" units="dimensionless"/>
...
```

7.5 Export Utilities

Use these routines to convert the current model state into other formats, like Matlab, CellML, Antimony and SBML.

class tellurium.tellurium.**ExtendedRoadRunner** (*args, **kwargs)

exportToAntimony (*filePath*, *current=True*)

Save current model as Antimony file.

Parameters

- **current** (*bool*) – export current model state
- **filePath** (*str*) – file path of Antimony file

exportToCellML (*filePath*, *current=True*)

Save current model as CellML file.

Parameters

- **current** (*bool*) – export current model state
- **filePath** (*str*) – file path of CellML file

exportToMatlab (*filePath*, *current=True*)

Save current model as Matlab file. To save the original model loaded into roadrunner use *current=False*.

Parameters

- **self** (*RoadRunner.roadrunner*) – RoadRunner instance
- **filePath** (*str*) – file path of Matlab file

exportToSBML (*filePath*, *current=True*)

Save current model as SBML file.

Parameters

- **current** (*bool*) – export current model state
- **filePath** (*str*) – file path of SBML file

getAntimony (*current=False*)

Antimony string of the original model loaded into roadrunner.

Parameters **current** (*bool*) – return current model state

Returns Antimony

Return type str

getCellML (*current=False*)

CellML string of the original model loaded into roadrunner.

Parameters **current** (*bool*) – return current model state

Returns CellML string

Return type str

getCurrentAntimony ()

Antimony string of the current model state.

See also: [getAntimony\(\)](#) :returns: Antimony string :rtype: str

getCurrentCellML ()

CellML string of current model state.

See also: [getCellML\(\)](#) :returns: CellML string :rtype: str

getCurrentMatlab ()

Matlab string of current model state.

Parameters **current** (*bool*) – return current model state

Returns Matlab string

Return type str

getMatlab (*current=False*)

Matlab string of the original model loaded into roadrunner.

See also: `getCurrentMatlab()` :returns: Matlab string :rtype: str

Given a `RoadRunner` instance, you can get an SBML representation of the current state of the model using `getCurrentSBML`. You can also get the initial SBML from when the model was loaded using `getSBML`. Finally, `exportToSBML` can be used to export the current model state to a file.

```
import tellurium as te
import tempfile

# load model
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
# file for export
f_sbml = tempfile.NamedTemporaryFile(suffix=".xml")

# export current model state
r.exportToSBML(f_sbml.name)

# to export the initial state when the model was loaded
# set the current argument to False
r.exportToSBML(f_sbml.name, current=False)

# The string representations of the current model are available via
str_sbml = r.getCurrentSBML()

# and of the initial state when the model was loaded via
str_sbml = r.getSBML()
print(str_sbml)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by libAntimony version v2.9.4 with libSBML version 5.15.0. -->
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1">
  <model id="__main" name="__main">
    <listOfCompartments>
      <compartment sboTerm="SBO:0000410" id="default_compartment" spatialDimensions="3
↪" size="1" constant="true"/>
    </listOfCompartments>
    <listOfSpecies>
      <species id="S1" compartment="default_compartment" initialConcentration="10"
↪hasOnlySubstanceUnits="false" boundaryCondition="false" constant="false"/>
      <species id="S2" compartment="default_compartment" hasOnlySubstanceUnits="false
↪" boundaryCondition="false" constant="false"/>
    </listOfSpecies>
    <listOfParameters>
      <parameter id="k1" value="0.1" constant="true"/>
    </listOfParameters>
    <listOfReactions>
      <reaction id="_J0" reversible="true" fast="false">
        <listOfReactants>
          <speciesReference species="S1" stoichiometry="1" constant="true"/>
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="S2" stoichiometry="1" constant="true"/>
        </listOfProducts>
      </reaction>
    </listOfReactions>
  </model>
</sbml>
```

(continues on next page)

(continued from previous page)

```

    </listOfProducts>
    <kineticLaw>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <apply>
          <times/>
          <ci> k1 </ci>
          <ci> S1 </ci>
        </apply>
      </math>
    </kineticLaw>
  </reaction>
</listOfReactions>
</model>
</sbml>

```

Similar to the SBML functions above, you can also use the functions `getCurrentAntimony` and `exportToAntimony` to get or export the current Antimony representation.

```

import tellurium as te
import tempfile

# load model
r = te.load('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
# file for export
f_antimony = tempfile.NamedTemporaryFile(suffix=".txt")

# export current model state
r.exportToAntimony(f_antimony.name)

# to export the initial state when the model was loaded
# set the current argument to False
r.exportToAntimony(f_antimony.name, current=False)

# The string representations of the current model are available via
str_antimony = r.getCurrentAntimony()

# and of the initial state when the model was loaded via
str_antimony = r.getAntimony()
print(str_antimony)

```

```

// Created by libAntimony v2.9.4
// Compartments and Species:
species S1, S2;

// Reactions:
_J0: S1 -> S2; k1*S1;

// Species initializations:
S1 = 10;
S2 = ;

// Variable initializations:
k1 = 0.1;

// Other declarations:
const k1;

```

Tellurium also has functions for exporting the current model state to CellML. These functionalities rely on using Antimony to perform the conversion.

```
import tellurium as te
import tempfile

# load model
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
# file for export
f_cellml = tempfile.NamedTemporaryFile(suffix=".cellml")

# export current model state
r.exportToCellML(f_cellml.name)

# to export the initial state when the model was loaded
# set the current argument to False
r.exportToCellML(f_cellml.name, current=False)

# The string representations of the current model are available via
str_cellml = r.getCurrentCellML()

# and of the initial state when the model was loaded via
str_cellml = r.getCellML()
print(str_cellml)
```

```
<?xml version="1.0"?>
<model xmlns:cellml="http://www.cellml.org/cellml/1.1#" xmlns="http://www.cellml.org/
↪cellml/1.1#" name="__main">
<component name="__main">
<variable initial_value="10" name="S1" units="dimensionless"/>
<variable name="S2" units="dimensionless"/>
<variable initial_value="0.1" name="k1" units="dimensionless"/>
<variable name="_J0" units="dimensionless"/>
<math xmlns="http://www.w3.org/1998/Math/MathML">
<apply>
<eq/>
<ci>_J0</ci>
<apply>
<times/>
<ci>k1</ci>
<ci>S1</ci>
</apply>
</apply>
</math>
<variable name="time" units="dimensionless"/>
<math xmlns="http://www.w3.org/1998/Math/MathML">
<apply>
<eq/>
<apply>
<diff/>
<bvar>
<ci>time</ci>
</bvar>
<ci>S1</ci>
</apply>
<apply>
<minus/>
<ci>_J0</ci>
```

(continues on next page)

(continued from previous page)

```

</apply>
</apply>
</math>
<math xmlns="http://www.w3.org/1998/Math/MathML">
<apply>
<eq/>
<apply>
<diff/>
<bvar>
<ci>time</ci>
</bvar>
<ci>S2</ci>
</apply>
<ci>_J0</ci>
</apply>
</math>
</component>
<group>
<relationship_ref relationship="encapsulation"/>
<component_ref component="__main"/>
</group>
</model>

```

To export the current model state to MATLAB, use `getCurrentMatlab`.

```

import tellurium as te
import tempfile

# load model
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
# file for export
f_matlab = tempfile.NamedTemporaryFile(suffix=".m")

# export current model state
r.exportToMatlab(f_matlab.name)

# to export the initial state when the model was loaded
# set the current argument to False
r.exportToMatlab(f_matlab.name, current=False)

# The string representations of the current model are available via
str_matlab = r.getCurrentMatlab()

# and of the initial state when the model was loaded via
str_matlab = r.getMatlab()
print(str_matlab)

```

```

% How to use:
%
% __main takes 3 inputs and returns 3 outputs.
%
% [t x rInfo] = __main(tspan,solver,options)
% INPUTS:
% tspan - the time vector for the simulation. It can contain every time point,
% or just the start and end (e.g. [0 1 2 3] or [0 100]).
% solver - the function handle for the odeN solver you wish to use (e.g. @ode23s).

```

(continues on next page)

(continued from previous page)

```

% options - this is the options structure returned from the MATLAB odeset
% function used for setting tolerances and other parameters for the solver.
%
% OUTPUTS:
% t - the time vector that corresponds with the solution. If tspan only contains
% the start and end times, t will contain points spaced out by the solver.
% x - the simulation results.
% rInfo - a structure containing information about the model. The fields
% within rInfo are:
%     stoich - the stoichiometry matrix of the model
%     floatingSpecies - a cell array containing floating species name, initial
%     value, and indicator of the units being in concentration or amount
%     compartments - a cell array containing compartment names and volumes
%     params - a cell array containing parameter names and values
%     boundarySpecies - a cell array containing boundary species name, initial
%     value, and indicator of the units being in concentration or amount
%     rateRules - a cell array containing the names of variables used in a rate rule
%
% Sample function call:
%     options = odeset('RelTol',1e-12,'AbsTol',1e-9);
%     [t x rInfo] = __main(linspace(0,100,100),@ode23s,options);
%
function [t x rInfo] = __main(tspan,solver,options)
    % initial conditions
    [x rInfo] = model();

    % initial assignments

    % assignment rules

    % run simulation
    [t x] = feval(solver,@model,tspan,x,options);

    % assignment rules

function [xdot rInfo] = model(time,x)
% x(1)      S1
% x(2)      S2

% List of Compartments
vol__default_compartment = 1;           %default_compartment

% Global Parameters
rInfo.g_p1 = 0.1;           % k1

if (nargin == 0)

    % set initial conditions
    xdot(1) = 10*vol__default_compartment;           % S1 = S1 [Concentration]
    xdot(2) = 0*vol__default_compartment;           % S2 = S2 [Concentration]

    % reaction info structure
    rInfo.stoich = [
        -1
        1
    ];

```

(continues on next page)

(continued from previous page)

```

    rInfo.floatingSpecies = {                                % Each row: [Species Name, Initial Value,
↪isAmount (1 for amount, 0 for concentration)]
        'S1' , 10, 0
        'S2' , 0, 0
    };

    rInfo.compartments = {                                    % Each row: [Compartment Name, Value]
        'default_compartment' , 1
    };

    rInfo.params = {                                         % Each row: [Parameter Name, Value]
        'k1' , 0.1
    };

    rInfo.boundarySpecies = {                                % Each row: [Species Name, Initial Value,
↪isAmount (1 for amount, 0 for concentration)]
    };

    rInfo.rateRules = {                                       % List of variables involved in a rate rule
    };

else

    % calculate rates of change
    R0 = rInfo.g_p1*(x(1));

    xdot = [
        - R0
        + R0
    ];
end;

%listOfSupportedFunctions
function z = pow (x,y)
    z = x^y;

function z = sqr (x)
    z = x*x;

function z = piecewise(varargin)
    numArgs = nargin;
    result = 0;
    foundResult = 0;
    for k=1:2: numArgs-1
        if varargin{k+1} == 1
            result = varargin{k};
            foundResult = 1;
            break;
        end
    end
    if foundResult == 0
        result = varargin{numArgs};
    end
    z = result;

```

(continues on next page)

(continued from previous page)

```
function z = gt(a,b)
    if a > b
        z = 1;
    else
        z = 0;
    end

function z = lt(a,b)
    if a < b
        z = 1;
    else
        z = 0;
    end

function z = geq(a,b)
    if a >= b
        z = 1;
    else
        z = 0;
    end

function z = leq(a,b)
    if a <= b
        z = 1;
    else
        z = 0;
    end

function z = neq(a,b)
    if a ~= b
        z = 1;
    else
        z = 0;
    end

function z = and(varargin)
    result = 1;
    for k=1:nargin
        if varargin{k} ~= 1
            result = 0;
            break;
        end
    end
    z = result;

function z = or(varargin)
    result = 0;
    for k=1:nargin
        if varargin{k} ~= 0
```

(continues on next page)

(continued from previous page)

```

        result = 1;
        break;
    end
end
z = result;

function z = xor(varargin)
    foundZero = 0;
    foundOne = 0;
    for k = 1:nargin
        if varargin{k} == 0
            foundZero = 1;
        else
            foundOne = 1;
        end
    end
    if foundZero && foundOne
        z = 1;
    else
        z = 0;
    end
end

function z = not(a)
    if a == 1
        z = 0;
    else
        z = 1;
    end
end

function z = root(a,b)
    z = a^(1/b);
end

```

The above examples rely on Antimony as an intermediary between formats. You can use this functionality directly using e.g. `antimony.getCellMLString`. A comprehensive set of functions can be found in the [Antimony API documentation](#).

```

import antimony
antimony.loadAntimonyString(''S1 -> S2; k1*S1; k1 = 0.1; S1 = 10'')
ant_str = antimony.getCellMLString(antimony.getMainModuleName())
print(ant_str)

```

```

<?xml version="1.0"?>
<model xmlns:cellml="http://www.cellml.org/cellml/1.1#" xmlns="http://www.cellml.org/
↪cellml/1.1#" name="__main">
<component name="__main">
<variable initial_value="10" name="S1" units="dimensionless"/>
<variable name="S2" units="dimensionless"/>
<variable initial_value="0.1" name="k1" units="dimensionless"/>
<variable name="_J0" units="dimensionless"/>
<math xmlns="http://www.w3.org/1998/Math/MathML">
<apply>
<eq/>

```

(continues on next page)

(continued from previous page)

```

<ci>_J0</ci>
<apply>
<times/>
<ci>k1</ci>
<ci>S1</ci>
</apply>
</apply>
</math>
<variable name="time" units="dimensionless"/>
<math xmlns="http://www.w3.org/1998/Math/MathML">
<apply>
<eq/>
<apply>
<diff/>
<bvar>
<ci>time</ci>
</bvar>
<ci>S1</ci>
</apply>
<apply>
<minus/>
<ci>_J0</ci>
</apply>
</apply>
</math>
<math xmlns="http://www.w3.org/1998/Math/MathML">
<apply>
<eq/>
<apply>
<diff/>
<bvar>
<ci>time</ci>
</bvar>
<ci>S2</ci>
</apply>
<ci>_J0</ci>
</apply>
</math>
</component>
<group>
<relationship_ref relationship="encapsulation"/>
<component_ref component="__main"/>
</group>
</model>

```

7.6 Stochastic Simulation

Use these routines to carry out Gillespie style stochastic simulations.

class tellurium.tellurium.**ExtendedRoadRunner** (*args, **kwargs)

getSeed (integratorName='gillespie')

Current seed used by the integrator with integratorName. Defaults to the seed of the gillespie integrator.

Parameters `integratorName` (*str*) – name of the integrator for which the seed should be returned

Returns current seed

Return type float

gillespie (**args, **kwargs*)

Run a Gillespie stochastic simulation.

Sets the integrator to gillespie and performs simulation.

```
rr = te.loada ('S1 -> S2; k1*S1; k1 = 0.1; S1 = 40')
# Simulate from time zero to 40 time units using variable step sizes (classic_
↳Gillespie)
result = rr.gillespie (0, 40)
# Simulate on a grid with 10 points from start 0 to end time 40
rr.reset()
result = rr.gillespie (0, 40, 10)
# Simulate from time zero to 40 time units using variable step sizes with_
↳given selection list
# This means that the first column will be time and the second column species_
↳S1
rr.reset()
result = rr.gillespie (0, 40, selections=['time', 'S1'])
# Simulate on a grid with 20 points from time zero to 40 time units
# using the given selection list
rr.reset()
result = rr.gillespie (0, 40, 20, ['time', 'S1'])
rr.plot(result)
```

Parameters

- **seed** (*int*) – seed for gillespie
- **args** – parameters for simulate
- **kwargs** – parameters for simulate

Returns simulation results

setSeed (*seed, integratorName='gillespie'*)

Set seed in integrator with integratorName. Defaults to the seed of the gillespie integrator.

Raises Error if integrator does not have key 'seed'.

Parameters

- **seed** – seed to set
- **integratorName** (*str*) – name of the integrator for which the seed should be returned

Stochastic simulations can be run by changing the current integrator type to 'gillespie' or by using the `r.gillespie` function.

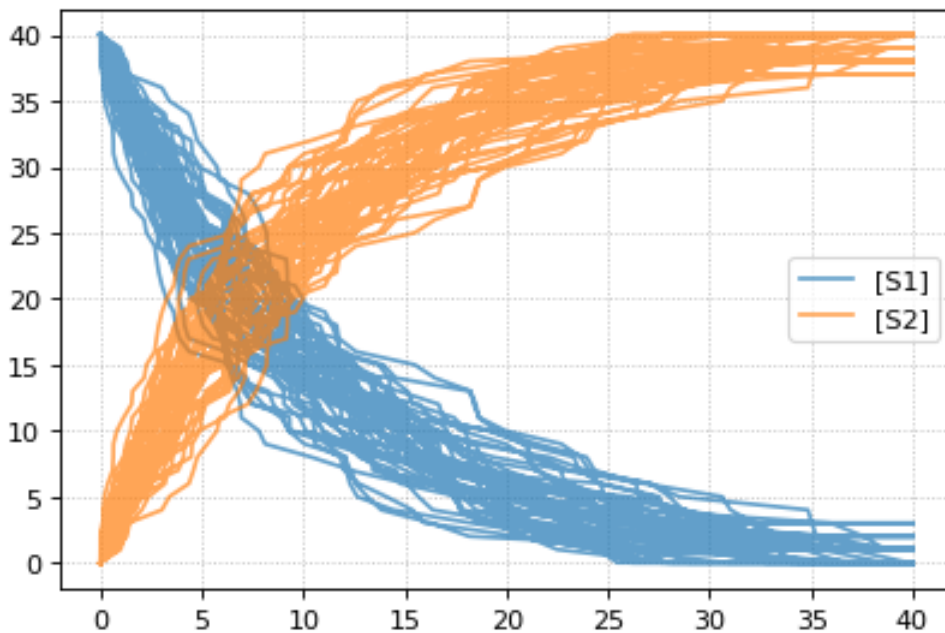
```
import tellurium as te
import numpy as np

r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 40')
r.integrator = 'gillespie'
r.integrator.seed = 1234
```

(continues on next page)

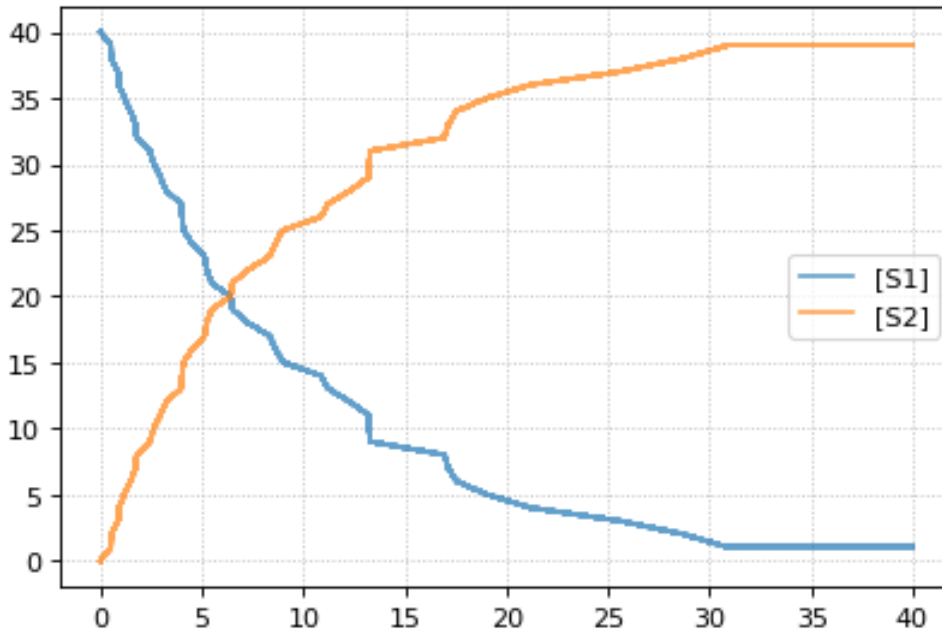
(continued from previous page)

```
results = []
for k in range(1, 50):
    r.reset()
    s = r.simulate(0, 40)
    results.append(s)
    r.plot(s, show=False, alpha=0.7)
te.show()
```



Setting the identical seed for all repeats results in identical traces in each simulation.

```
results = []
for k in range(1, 20):
    r.reset()
    r.setSeed(123456)
    s = r.simulate(0, 40)
    results.append(s)
    r.plot(s, show=False, loc=None, color='black', alpha=0.7)
te.show()
```

You can combine two timecourse simulations and change e.g. parameter values in between each simulation. The `gillespie` method simulates up to the given end time 10, after which you can make arbitrary changes to the model, then simulate again.

When using the `r.plot` function, you can pass the parameter `labels`, which controls the names that will be used in the figure legend, and `tag`, which ensures that traces with the same tag will be drawn with the same color (each species within each trace will be plotted in its own color, but these colors will match trace to trace).

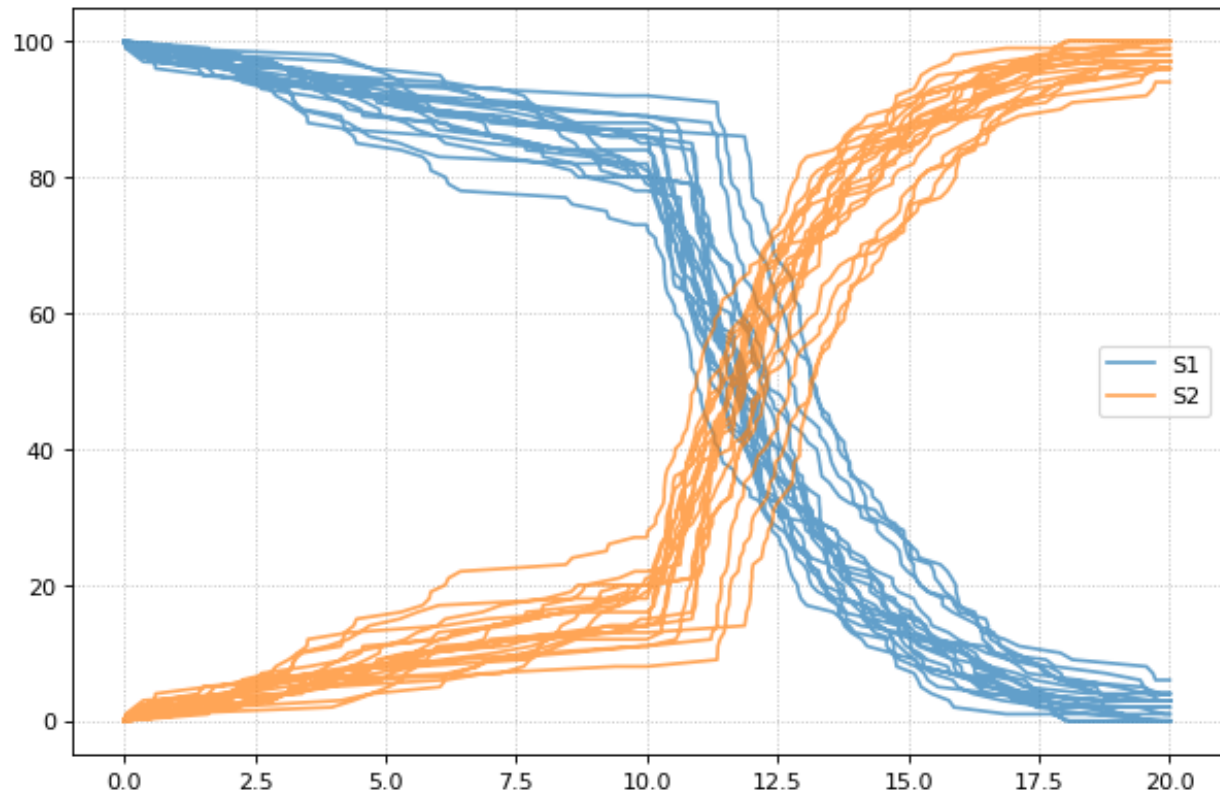
```
import tellurium as te

r = te.loada('S1 -> S2; k1*S1; k1 = 0.02; S1 = 100')
r.setSeed(1234)
for k in range(1, 20):
    r.resetToOrigin()
    res1 = r.gillespie(0, 10)
    r.plot(res1, show=False) # plot first half of data

    # change in parameter after the first half of the simulation
    # We could have also used an Event in the antimony model,
    # which are described further in the Antimony Reference section
    r.k1 = r.k1*20
    res2 = r.gillespie(10, 20)

    r.plot(res2, show=False) # plot second half of data

te.show()
```



7.7 Math

Use these routines to perform various calculations.

`tellurium.getEigenvalues(m)`

Eigenvalues of matrix.

Convenience method for computing the eigenvalues of a matrix `m`. Uses numpy `eig` to compute the eigenvalues.

Parameters `m` – numpy array

Returns numpy array containing eigenvalues

`tellurium.rank(A, atol=1e-13, rtol=0)`

Estimate the rank (i.e. the dimension of the column space) of a matrix.

The algorithm used by this function is based on the singular value decomposition of `A`.

A [ndarray] `A` should be at most 2-D. A 1-D array with length `n` will be treated as a 2-D with shape `(1, n)`

atol [float] The absolute tolerance for a zero singular value. Singular values smaller than `atol` are considered to be zero.

rtol [float] The relative tolerance. Singular values less than `rtol*smax` are considered to be zero, where `smax` is the largest singular value.

If both `atol` and `rtol` are positive, the combined tolerance is the maximum of the two; that is:

```
tol = max(atol, rtol * smax)
```

Singular values smaller than `tol` are considered to be zero.

r [int] The estimated rank of the matrix.

numpy.linalg.matrix_rank `matrix_rank` is basically the same as this function, but it does not provide the option of the absolute tolerance.

`tellurium.nullspace(A, atol=1e-13, rtol=0)`

Compute an approximate basis for the nullspace of A.

The algorithm used by this function is based on the singular value decomposition of A.

A [ndarray] A should be at most 2-D. A 1-D array with length k will be treated as a 2-D with shape (1, k)

atol [float] The absolute tolerance for a zero singular value. Singular values smaller than *atol* are considered to be zero.

rtol [float] The relative tolerance. Singular values less than *rtol***smax* are considered to be zero, where *smax* is the largest singular value.

If both *atol* and *rtol* are positive, the combined tolerance is the maximum of the two; that is:

```
tol = max(atol, rtol * smax)
```

Singular values smaller than *tol* are considered to be zero.

ns [ndarray] If A is an array with shape (m, k), then *ns* will be an array with shape (k, n), where n is the estimated dimension of the nullspace of A. The columns of *ns* are a basis for the nullspace; each element in `numpy.dot(A, ns)` will be approximately zero.

`tellurium.rref(A)`

Compute the reduced row echelon for the matrix A. Returns returns a tuple of two elements. The first is the reduced row echelon form, and the second is a list of indices of the pivot columns.

7.8 ODE Extraction Methods

Routines to extract ODEs.

`tellurium.getODEsFromSBMLFile(fileName)`

Given a SBML file name, this function returns the model as a string of rules and ODEs

```
>>> te.getODEsFromSBMLFile ('mymodel.xml')
```

`tellurium.getODEsFromSBMLString(sbmlStr)`

Given a SBML string this function returns the model as a string of rules and ODEs

```
>>> te.getODEsFromSBMLString (sbmlStr)
```

`tellurium.getODEsFromModel(roadrunnerModel)`

Given a roadrunner instance this function returns a string of rules and ODEs

```
>>> r = te.loada ('S1 -> S2; k1*S1; k1=1')
>>> te.getODEsFromModel (r)
```

7.9 Plotting

Tellurium has a plotting engine which can target either Plotly (when used in a notebook environment) or Matplotlib. To specify which engine to use, use `te.setDefaultPlottingEngine()`.

`tellurium.plot(x, y, show=True, **kwargs)`
Create a 2D scatter plot.

Parameters

- **x** – A numpy array describing the X datapoints. Should have the same number of rows as y.
- **y** – A numpy array describing the Y datapoints. Should have the same number of rows as x.
- **tag** – A tag so that all traces of the same type are plotted using same color/label (for e.g. multiple stochastic traces).
- **tags** – Like tag, but for multiple traces.
- **name** – The name of the trace.
- **label** – The name of the trace.
- **names** – Like name, but for multiple traces to appear in the legend.
- **labels** – The name of the trace.
- **alpha** – Floating point representing the opacity ranging from 0 (transparent) to 1 (opaque).
- **show** – `show=True` (default) shows the plot, use `show=False` to plot multiple simulations in one plot
- **showlegend** – Whether to show the legend or not.
- **mode** – Can be set to ‘markers’ to generate scatter plots, or ‘dash’ for dashed lines.

```
import numpy as np, tellurium as te
result = np.array([[1,2,3,4], [7.2,6.5,8.8,10.5], [9.8, 6.5, 4.3,3.0]])
te.plot(result[:,0], result[:,1], name='Second column', show=False)
te.plot(result[:,0], result[:,2], name='Third column', show=False)
te.show(reset=False) # does not reset the plot after showing plot
te.plot(result[:,0], result[:,3], name='Fourth column', show=False)
te.show()
```

NOTE: When loading a model with `r = te.loada(“antimony_string”)` and calling `r.plot()`, it is the below `tellurium.ExtendedRoadRunner.plot()` method below that is called not `te.plot()`.

`tellurium.plotArray(result, loc='upper right', show=True, resetColorCycle=True, xlabel=None, ylabel=None, title=None, xlim=None, ylim=None, xscale='linear', yscale='linear', grid=False, labels=None, **kwargs)`

Plot an array.

Parameters

- **result** – Array to plot, first column of the array must be the x-axis and remaining columns the y-axis
- **loc** (*str*) – Location of legend box. Valid strings ‘best’ | ‘upper right’ | ‘upper left’ | ‘lower left’ | ‘lower right’ | ‘right’ | ‘center left’ | ‘center right’ | ‘lower center’ | ‘upper center’ | ‘center’ |
- **color** (*str*) – ‘red’, ‘blue’, etc. to use the same color for every curve
- **labels** – A list of labels for the legend, include as many labels as there are curves to plot
- **xlabel** (*str*) – x-axis label
- **ylabel** (*str*) – y-axis label
- **title** (*str*) – Add plot title

- **xlim** – Limits on x-axis (tuple [start, end])
- **ylim** – Limits on y-axis
- **yscale** – ‘linear’ or ‘log’ scale for x-axis
- **yscale** – ‘linear’ or ‘log’ scale for y-axis
- **grid** (*bool*) – Show grid
- **show** – show=True (default) shows the plot, use show=False to plot multiple simulations in one plot
- **resetColorCycle** (*bool*) – If true, resets color cycle on given figure (works with show=False to plot multiple simulations on a single plot)
- **kwargs** – Additional matplotlib keywords like linewidth, linestyle...

```
import numpy as np, tellurium as te
result = np.array([[1,2,3], [7.2,6.5,8.8], [9.8, 6.5, 4.3]])
te.plotArray(result, title="My graph", xlim=(1, 5)), labels=["Label 1", "Label 2
↪"],
               yscale='log', linestyle='dashed')
```

The function `tellurium.plotArray` assumes that the first column in the array is the x-axis and the second and subsequent columns represent curves on the y-axis.

```
class tellurium.tellurium.ExtendedRoadRunner (*args, **kwargs)
```

```
draw (**kwargs)
```

Draws an SBMLDiagram of the current model.

To set the width of the output plot provide the ‘width’ argument. Species are drawn as white circles (boundary species shaded in blue), reactions as grey squares. Currently only the drawing of medium-size networks is supported.

```
plot (result=None, show=True, xtitle=None, ytitle=None, title=None, linewidth=2, xlim=None,
      ylim=None, logx=False, logy=False, xscale='linear', yscale='linear', grid=False, ordi-
      nates=None, tag=None, labels=None, figsize=(6, 4), savefig=None, dpi=80, alpha=1.0,
      **kwargs)
```

Plot roadrunner simulation data.

Plot is called with simulation data to plot as the first argument. If no data is provided the data currently held by roadrunner generated in the last simulation is used. The first column is considered the x axis and all remaining columns the y axis. If the result array has no names, then the current `r.selections` are used for naming. In this case the dimension of the `r.selections` has to be the same like the number of columns of the result array.

Curves are plotted in order of selection (columns in result).

In addition to the listed keywords plot supports all matplotlib.pyplot.plot keyword arguments, like color, alpha, linewidth, linestyle, marker, ...

```
sbml = te.getTestModel('feedback.xml')
r = te.loadSBMLModel(sbml)
s = r.simulate(0, 100, 201)
r.plot(s, loc="upper right", linewidth=2.0, lineStyle='-', marker='o', ↪
      ↪markersize=2.0,
      ↪alpha=0.8, title="Feedback Oscillation", xlabel="time", ylabel=
      ↪"concentration",
      ↪xlim=[0,100], ylim=[-1, 4])
```

Parameters

- **result** – results data to plot (numpy array)
- **show** (*bool*) – show=True (default) shows the plot, use show=False to plot multiple simulations in one plot
- **xtitle** (*str*) – x-axis label
- **xlabel** (*str*) – x-axis label (same as xtitle)
- **ytitle** (*str*) – y-axis label
- **ylabel** (*str*) – y-axis label (same as ytitle)
- **title** (*str*) – plot title
- **linewidth** (*float*) – linewidth of the plot
- **xlim** – limits on x-axis (tuple [start, end])
- **ylim** – limits on y-axis
- **logx** (*bool*) – use log scale for x-axis
- **logy** (*bool*) – use log scale for y-axis
- **xscale** – ‘linear’ or ‘log’ scale for x-axis
- **yscale** – ‘linear’ or ‘log’ scale for y-axis
- **grid** (*bool*) – show grid
- **ordinates** – If supplied, only these selections will be plotted (see RoadRunner selections)
- **tag** – If supplied, all traces with the same tag will be plotted with the same color/style
- **labels** – ‘id’ to use species IDs
- **figsize** – If supplied, customize the size of the figure (width,height)
- **savefig** (*str*) – If supplied, saves the figure to specified location
- **dpi** (*int*) – Change the dpi of the saved figure
- **alpha** (*float*) – Change the alpha value of the figure
- **kwargs** – additional matplotlib keywords like marker, lineStyle, ...

NOTE: When loading a model with `r = te.loada('antimony_string')` and calling `r.plot()`, it is the above `tellurium.ExtendedRoadRunner.plot()` method below that is called not `te.plot()`.

7.9.1 Add plot elements

Example showing how to embellish a graph - change title, axes labels, set axis limit. Example also uses an event to pulse S1.

```
import tellurium as te, roadrunner

r = te.loada('''
  $Xo -> S1; k1*Xo;
  S1 -> $X1; k2*S1;
```

(continues on next page)

(continued from previous page)

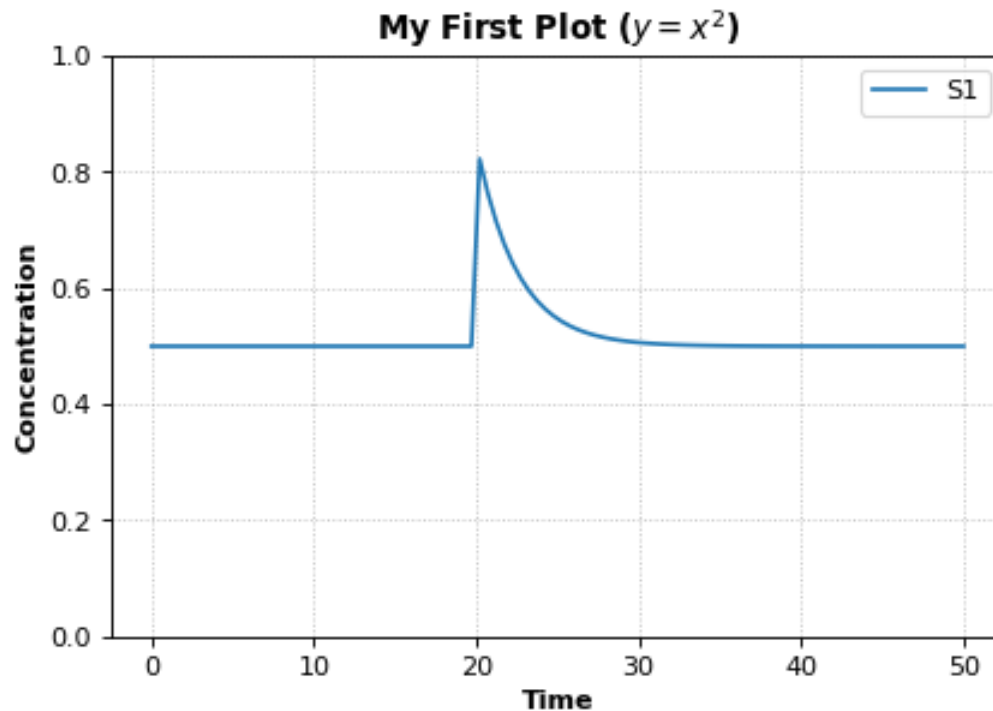
```

    k1 = 0.2; k2 = 0.4; Xo = 1; S1 = 0.5;
    at (time > 20): S1 = S1 + 0.35
'''

# Simulate the first part up to 20 time units
m = r.simulate (0, 50, 100, ["time", "S1"])

# using latex syntax to
↪render math
r.plot(m, ylim=(0.,1.), xtitle='Time', ytitle='Concentration', title='My First Plot (
↪$y = x^2$)')

```



7.9.2 Saving plots

To save a plot, use `r.plot` and the `savefig` parameter. Use `dpi` to specify image quality. Pass in the save location along with the image name.

```

import tellurium as te, os
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
result = r.simulate(0, 50, 100)
currentDir = os.getcwd() # gets the current directory
r.plot(title='My plot', xtitle='Time', ytitle='Concentration', dpi=150,
       savefig=currentDir + '\\test.png') # save image to current directory as "test.
↪png"

```

The path can be specified as a written out string. The plot can also be saved as a pdf instead of png.

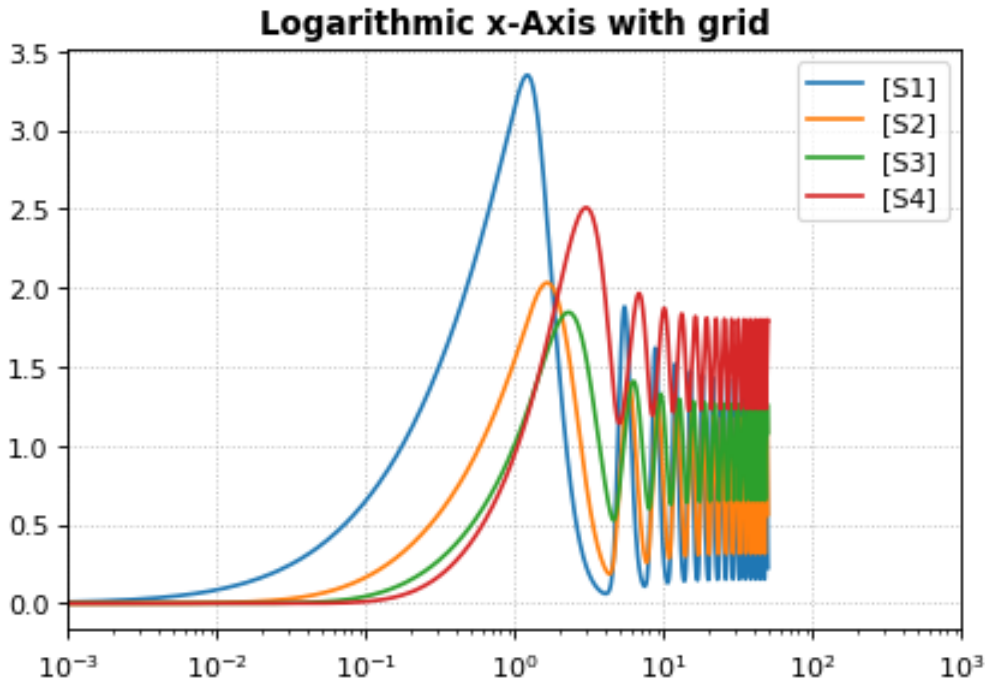
```
savefig='C:\\Tellurium-Winpython-3.6\\settings\\.spyder-py3\\test.pdf'
```

7.9.3 Logarithmic axis

The axis scale can be adapted with the `xscale` and `yscale` settings.

```
import tellurium as te

r = te.loadTestModel('feedback.xml')
r.integrator.variable_step_size = True
s = r.simulate(0, 50)
r.plot(s, logx=True, xlim=[10E-4, 10E2],
       title="Logarithmic x-Axis with grid", ylabel="concentration");
```



7.9.4 Plotting multiple simulations

All plotting is done via the `r.plot` or `te.plotArray` functions. To plot multiple curves in one figure use the `show=False` setting.

```
import tellurium as te

import numpy as np
import matplotlib.pyplot as plt

# Load a model and carry out a simulation generating 100 points
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
r.draw(width=100)

# get colormap
# Colormap instances are used to convert data values (floats) from the interval [0, 1]
cmap = plt.get_cmap('Blues')

k1_values = np.linspace(start=0.1, stop=1.5, num=15)
```

(continues on next page)

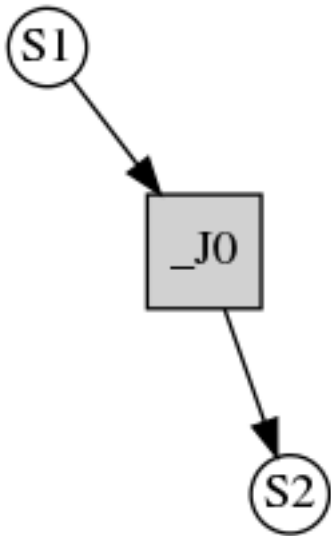
(continued from previous page)

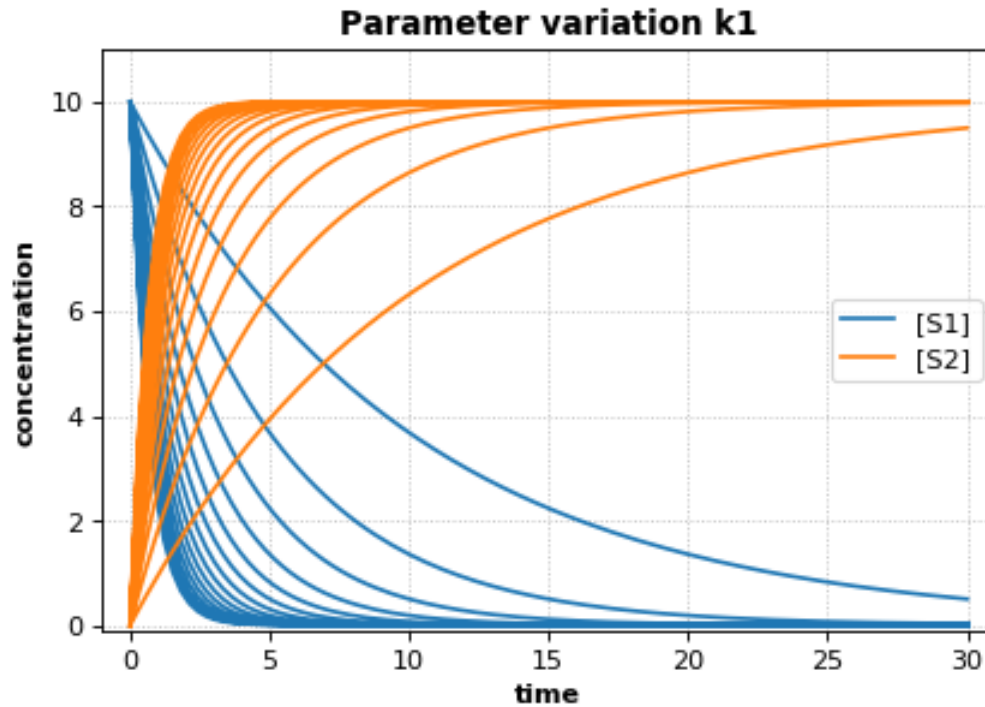
```
max_k1 = max(k1_values)
for k, value in enumerate(k1_values):
    r.reset()
    r.k1 = value
    s = r.simulate(0, 30, 100)

    color = cmap((value+max_k1)/(2*max_k1))
    # use show=False to plot multiple curves in the same figure
    r.plot(s, show=False, title="Parameter variation k1", xtitle="time", ytitle=
↪ "concentration",
        xlim=[-1, 31], ylim=[-0.1, 11])

te.show()

print('Reference Simulation: k1 = {}'.format(r.k1))
print('Parameter variation: k1 = {}'.format(k1_values))
```





```
Reference Simulation: k1 = 1.5
Parameter variation: k1 = [0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. 1.1 1.2 1.3 1.4 1.
↪5]
```

7.9.5 Using Tags and Names

Tags can be used to coordinate the color, opacity, and legend names between several sets of data. This can be used to highlight certain features that these datasets have in common. Names allow you to give a more meaningful description of the data in the legend.

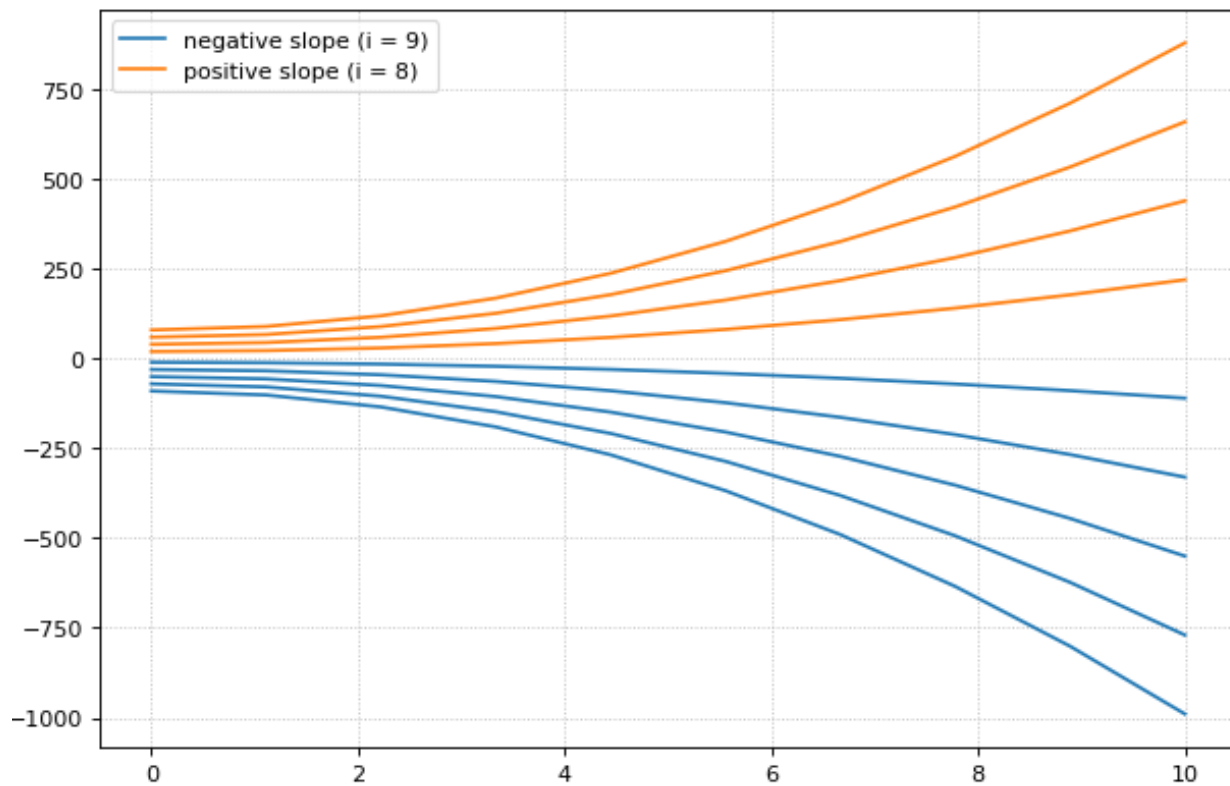
```
import tellurium as te
import numpy as np

for i in range(1, 10):
    x = np.linspace(0, 10, num = 10)
    y = i*x**2 + 10*i

    if i % 2 == 0:
        next_tag = "positive slope"
    else:
        next_tag = "negative slope"
        y = -1*y

    next_name = next_tag + " (i = " + str(i) + ")"
    te.plot(x, y, show = False, tag = next_tag, name = next_name)

te.show()
```



Note that only two items show up in the legend, one for each tag used. In this case, the name found in the legend will match the name of the last set of data plotted using that specific tag. The color and opacity for each tagged groups will also be chosen from the last dataset inputted with that given tag.

7.9.6 Subplots

`te.plotArray` can be used in conjunction with `matplotlib` functions to create subplots.

```
import tellurium as te
import numpy as np
import matplotlib.pyplot as plt

r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 20')
r.setIntegrator('gillespie')
r.integrator.seed = '1234'
kValues = np.linspace(0.1, 0.9, num=9) # generate k1 values

plt.gcf().set_size_inches(10, 10) # size of figure
plt.subplots_adjust(wspace=0.4, hspace=0.4) # adjust the space between subplots
plt.suptitle('Variation in k1 value', fontsize=16) # main title

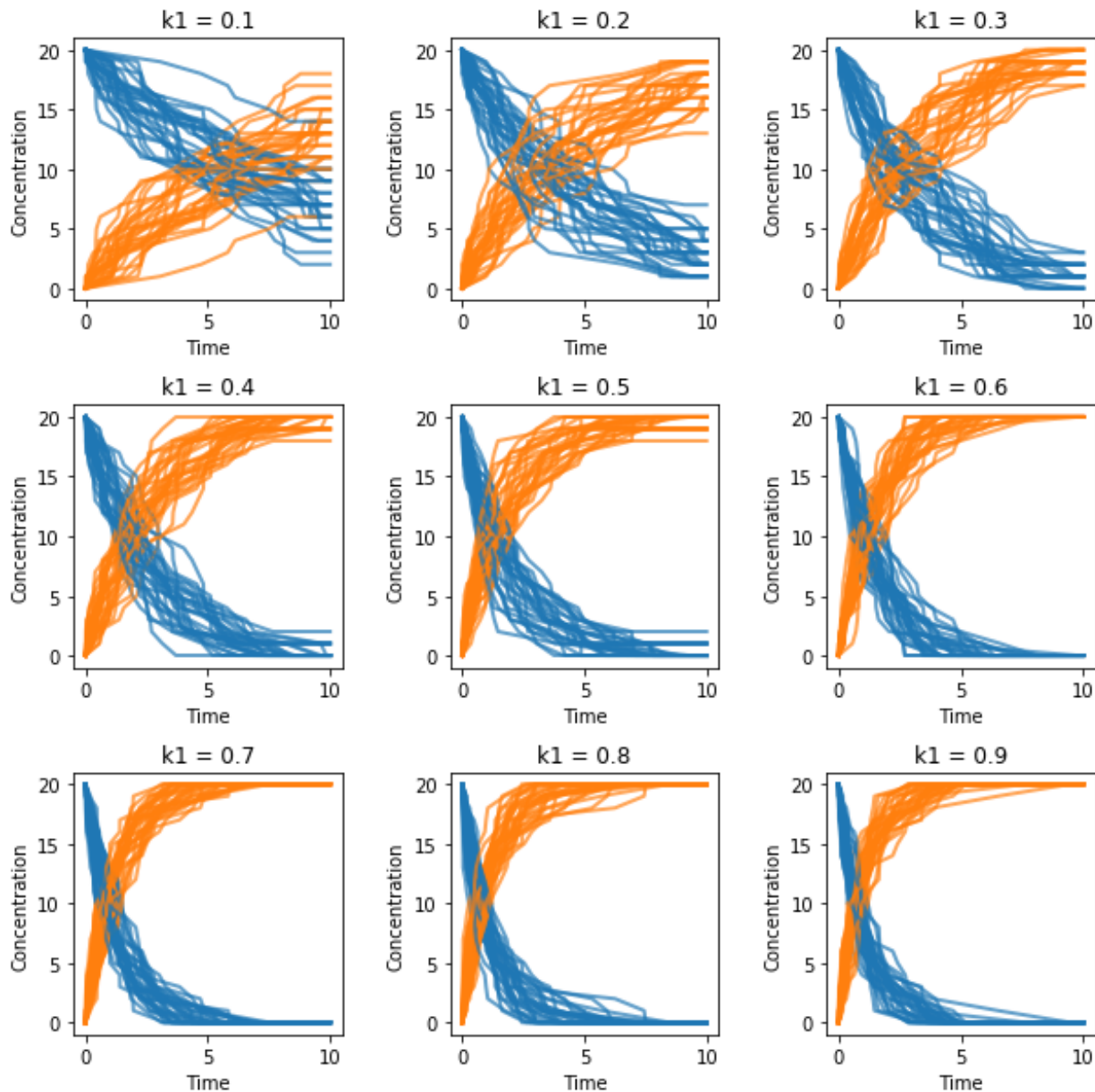
for i in range(1, len(kValues) + 1):
    r.k1 = kValues[i - 1]
    # designates number of subplots (row, col) and spot to plot next
    plt.subplot(3, 3, i)
    for j in range(1, 30):
        r.reset()
        s = r.simulate(0, 10)
```

(continues on next page)

(continued from previous page)

```
t = "k1 = " + '{:.1f}'.format(kValues[i - 1])
# plot each subplot, use show=False to save multiple traces
te.plotArray(s, show=False, title=t, xlabel='Time',
            ylabel='Concentration', alpha=0.7)
```

Variation in k1 value



7.9.7 External Plotting

For those more familiar with plotting in Python, other libraries such as `matplotlib.pyplot` offer a wider range of plotting options. To use these external libraries, extract the simulation timecourse data returned from `r.simulate`.

Data is returned in the form of a dictionary/NamedArray, so specific elements can easily be extracted using the species name as the key.

```
import tellurium as te
import matplotlib.pyplot as plt

antimonyString = '''
model feedback()
// Reactions:
J0: Nan1 + Mol -> Nan1Mol; (K1*Nan1*Mol);
J1: Nan1Mol -> Nan1 + Mol; (K_1*Nan1Mol);
J2: Nan1Mol + Nan2 -> Nan1MolNan2; (K2*Nan1Mol*Nan2)
J3: Nan1MolNan2 + GeneOff -> GeneOn; (K3*Nan1MolNan2*GeneOff);
J4: GeneOn -> Nan1MolNan2 + GeneOff; (K_3*GeneOn);

// Species initializations:
Nan1 = 0.0001692; Mol = 0.0001692/2; Nan2 = 0.0001692; Nan1Mol = 0;
Nan1MolNan2 = 0; GeneOff = 5*10^-5; GeneOn = 0;

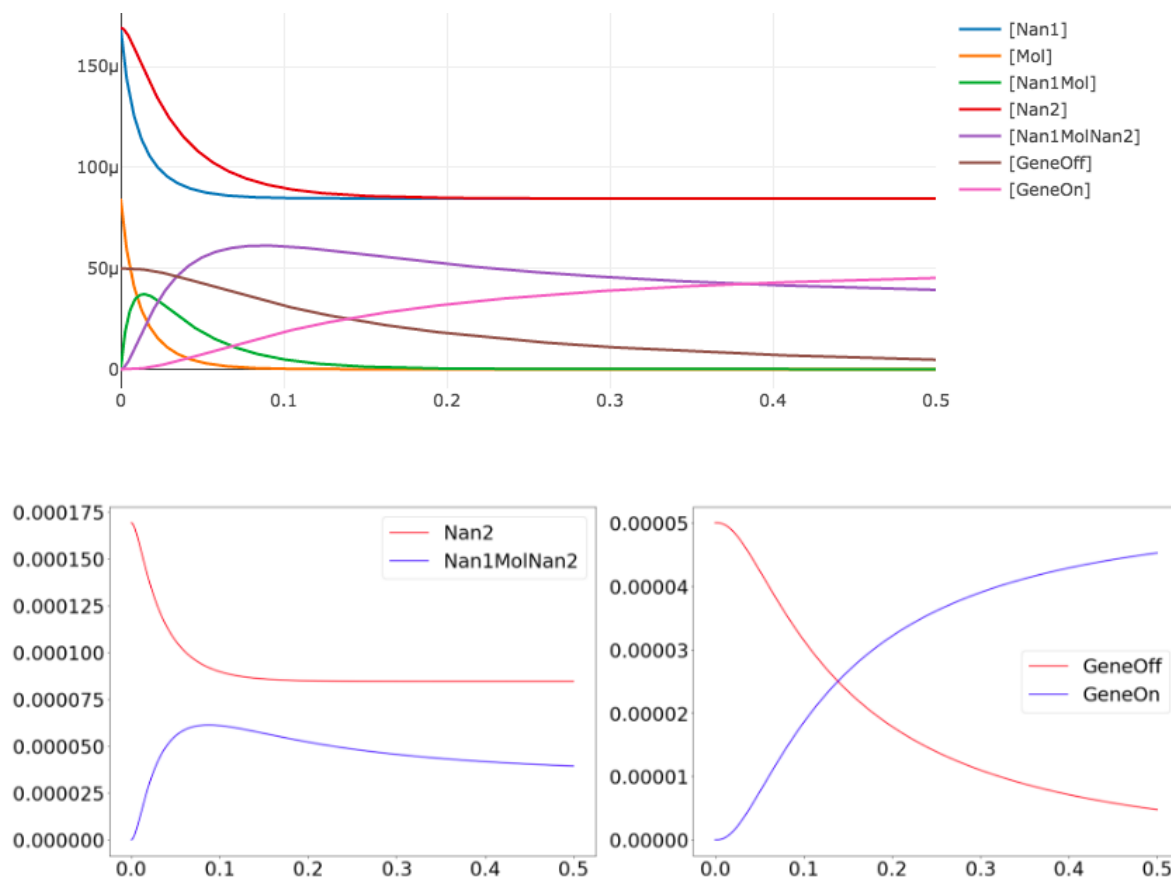
// Variable initialization:
K1 = 6.1*10^5; K_1 = 8*10^-5; K2 = 3.3*10^5; K_2 = 5.7*10^-8; K3 = 1*10^5; K_3 = 0;
end'''

r = te.loada(antimonyString)
results = r.simulate(0,0.5,1000)
r.plot()

plt.figure(figsize=(30,10));
plt.rc('font', size=30);

plt.subplot(1,2,1);
plt.plot(results['time'], results['Nan2'], 'r', results['time'], results[
↪ 'Nan1MolNan2'], 'b');
plt.legend({'Nan2', 'Nan1MolNan2'});

plt.subplot(1,2,2);
plt.plot(results['time'], results['GeneOff'], 'r', results['time'], results[
↪ 'GeneOn'], 'b');
plt.legend({'GeneOff', 'GeneOn'});
```



Note that we can extract all the time course data for a specific species such as Nan2 by calling `results['[Nan2]']`. The extract brackets `[]` around Nan2 may or may not be required depending on if the units are in terms of concentration or just a count. To check, simply print out results and you can see the names of each species.

7.9.8 Draw diagram

This example shows how to draw a network diagram, requires `graphviz`.

```
import tellurium as te

r = te.loada('''
model feedback()
  // Reactions:http://localhost:8888/notebooks/core/tellurium_export.ipynb#
  J0: $X0 -> S1; (VM1 * (X0 - S1/Keq1))/(1 + X0 + S1 + S4^h);
  J1: S1 -> S2; (10 * S1 - 2 * S2) / (1 + S1 + S2);
  J2: S2 -> S3; (10 * S2 - 2 * S3) / (1 + S2 + S3);
  J3: S3 -> S4; (10 * S3 - 2 * S4) / (1 + S3 + S4);
  J4: S4 -> $X1; (V4 * S4) / (KS4 + S4);

  // Species initializations:
  S1 = 0; S2 = 0; S3 = 0;
  S4 = 0; X0 = 10; X1 = 0;

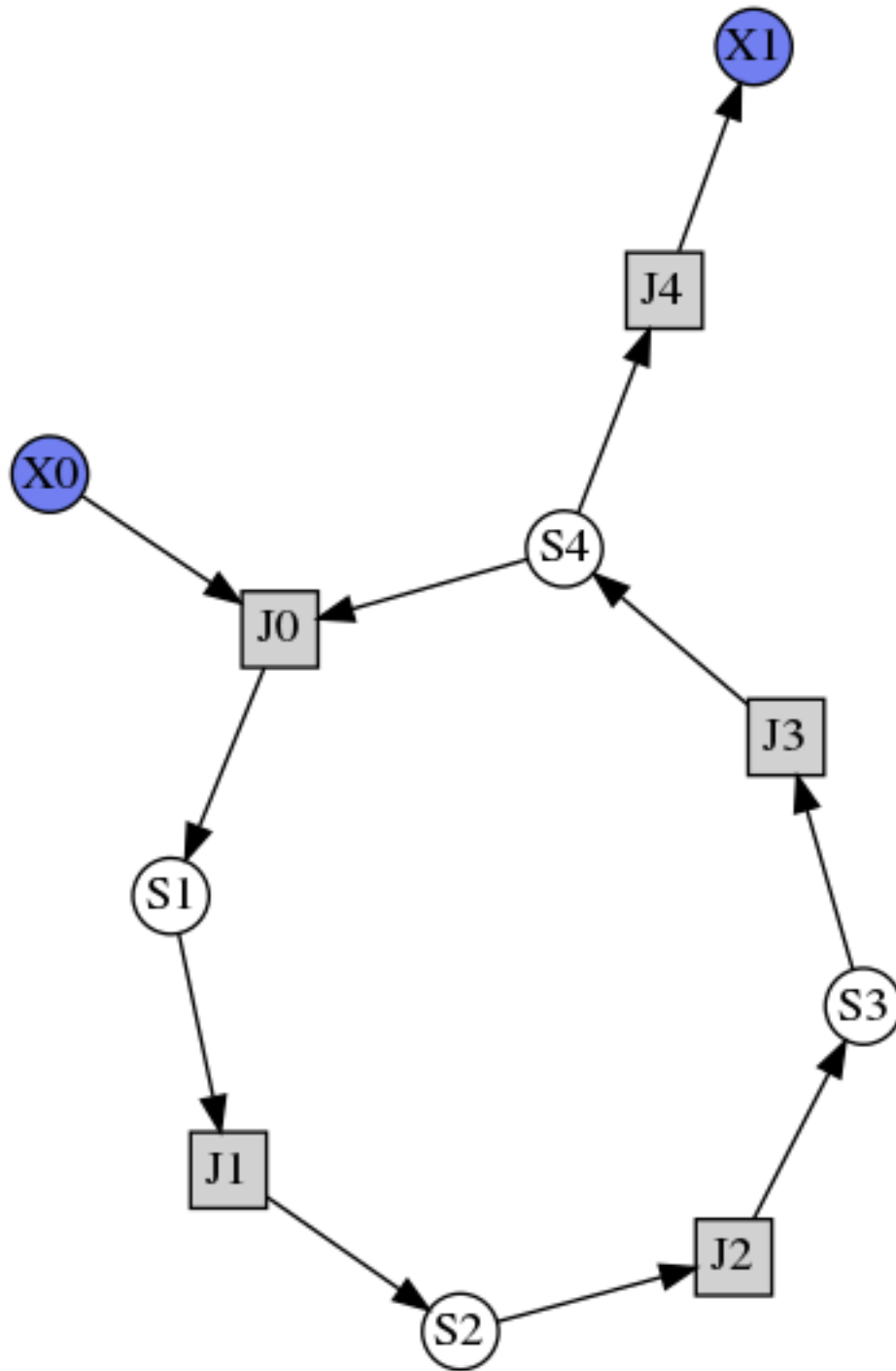
  // Variable initialization:
```

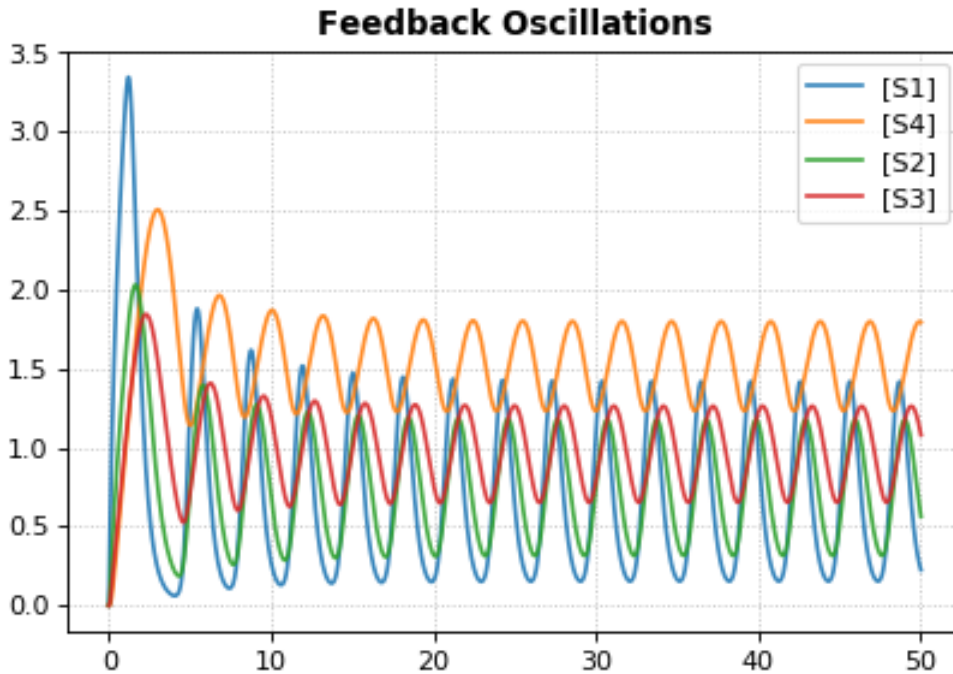
(continues on next page)

(continued from previous page)

```
    VM1 = 10; Keq1 = 10; h = 10; V4 = 2.5; KS4 = 0.5;
end'')

# simulate using variable step size
r.integrator.setValue('variable_step_size', True)
s = r.simulate(0, 50)
# draw the diagram
r.draw(width=200)
# and the plot
r.plot(s, title="Feedback Oscillations", ylabel="concentration", alpha=0.9);
```





7.9.9 Parameter Scans

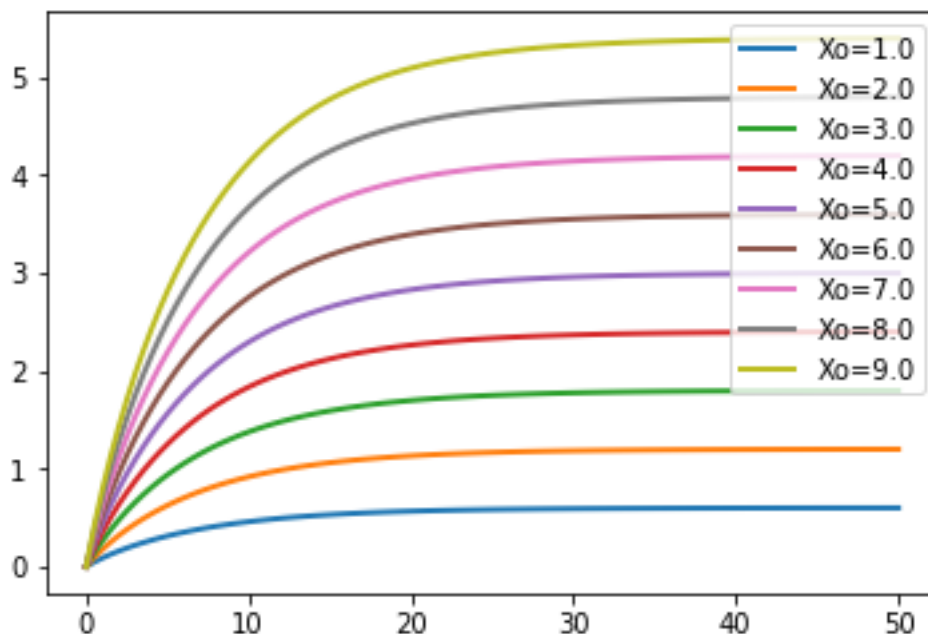
To study the consequences of varying a specific parameter value or initial concentration on a simulation, iteratively adjust the given parameter over a range of values of interest and re-run the simulation. Using the `show` parameter and `te.show` we can plot all these simulations on a single figure.

```
import tellurium as te
import roadrunner
import numpy as np

r = te.loada("""
    $Xo -> A; k1*Xo;
    A -> B; kf*A - kr*B;
    B -> ; k2*B;

    Xo = 5
    k1 = 0.1; k2 = 0.5;
    kf = 0.3; kr = 0.4
""")

for Xo in np.arange(1.0, 10, 1):
    r.reset()
    r.Xo = Xo
    m = r.simulate(0, 50, 100, ['time', 'A'])
    te.plotArray(m, show=False, labels=['Xo='+str(Xo)], resetColorCycle=False)
te.show()
```



7.9.10 Parameter Uncertainty Modeling

In most systems, some parameters are more sensitive to perturbations than others. When studying these systems, it is important to understand which parameters are highly sensitive, as errors (i.e. measurement error) introduced to these variables can create drastic differences between experimental and simulated results. To study the sensitivity of these parameters, we can sweep over a range of values as we did in the parameter scan example above. These ranges represent our uncertainty in the value of the parameter, and those parameters that create highly variable results in some measure of an output variable are deemed to be sensitive.

```
import numpy as np
import tellurium as te
import roadrunner
import antimony
import matplotlib.pyplot as plt
import math

antimonyString = '''
model feedback()
  // Reactions:
  J0: Nan1 + Mol -> Nan1Mol; (K1*Nan1*Mol);
  J1: Nan1Mol -> Nan1 + Mol; (K_1*Nan1Mol);
  J2: Nan1Mol + Nan2 -> Nan1MolNan2; (K2*Nan1Mol*Nan2)
  J3: Nan1MolNan2 + GeneOff -> GeneOn; (K3*Nan1MolNan2*GeneOff);
  J4: GeneOn -> Nan1MolNan2 + GeneOff; (K_3*GeneOn);

  // Species initializations:
  Nan1 = 0.0001692; Mol = 0.0001692/2; Nan2 = 0.0001692; Nan1Mol = 0;
  Nan1MolNan2 = 0; GeneOff = 5*10^-5; GeneOn = 0;

  // Variable initialization:
  K1 = 6.1*10^5; K_1 = 8*10^-5; K2 = 3.3*10^5; K_2 = 5.7*10^-8; K3 = 1*10^5; K_3 = 0;
end'''
```

(continues on next page)

(continued from previous page)

```

r = te.loada (model.antimonyString)

def plot_param_uncertainty(model, startVal, name, num_sims):
    stdDev = 0.6

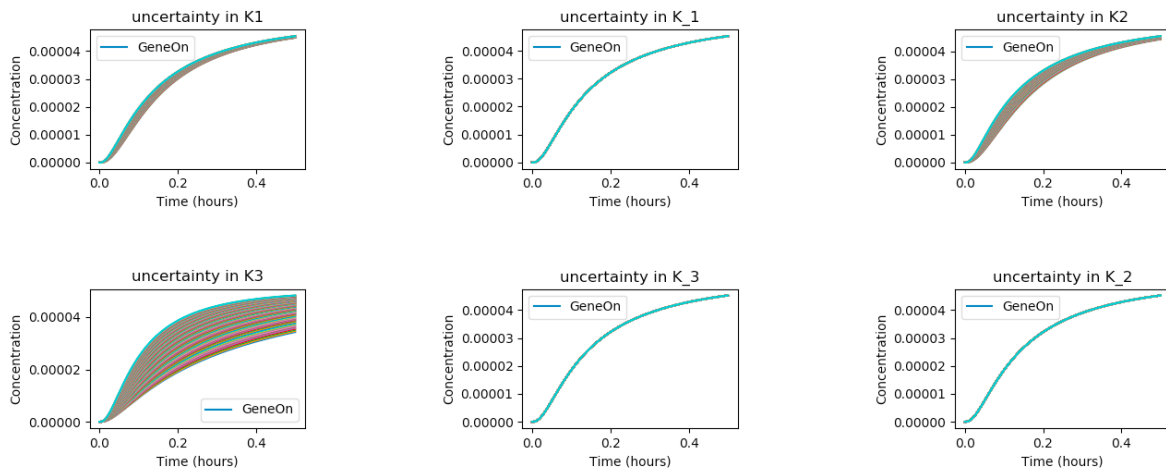
    # assumes initial parameter estimate as mean and iterates 60% above and below.
    vals = np.linspace((1-stdDev)*startVal, (1+stdDev)*startVal, 100)
    for val in vals:
        r.resetToOrigin()
        exec("r.%s = %f" % (name, val))
        result = r.simulate(0,0.5,1000, selections = ['time', 'GeneOn'])
        plt.plot(result[:,0],result[:,1])
        plt.title("uncertainty in " + name)
    plt.legend(["GeneOn"])
    plt.xlabel("Time (hours)")
    plt.ylabel("Concentration")

startVals = r.getGlobalParameterValues();
names = list(enumerate([x for x in r.getGlobalParameterIds() if ("K" in x or "k" in x)]));

n = len(names) + 1;
dim = math.ceil(math.sqrt(n))
for i,next_param in enumerate(names):
    plt.subplot(dim,dim,i+1)
    plot_param_uncertainty(r, startVals[next_param[0]], next_param[1], 100)

plt.tight_layout()
plt.show()

```



In the above code, the `exec` command is used to set the model parameters to their given value (i.e. `r.K1 = 1.5`) and the code sweeps through all the given parameters of interests (names). Above, we see that the K3 parameter produces the widest distribution of outcomes, and is thus the most sensitive under the given model, taking into account its assumptions and approximate parameter values. On the other hand, variations in K_1, K1, and K_2 seem to have very little effect on the outcome of the system.

7.10 Model Reset

The `reset` function of a `RoadRunner` instance reset the system variables (usually species concentrations) to their respective initial values. `resetAll` resets variables to their CURRENT initial as well as resets parameters. `resetToOrigin` completely resets the model.

```
import tellurium as te

r = te.loada ('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
r.integrator.setValue('variable_step_size', True)

# simulate model
sim1 = r.simulate(0, 5)
print('*** sim1 ***')
r.plot(sim1)

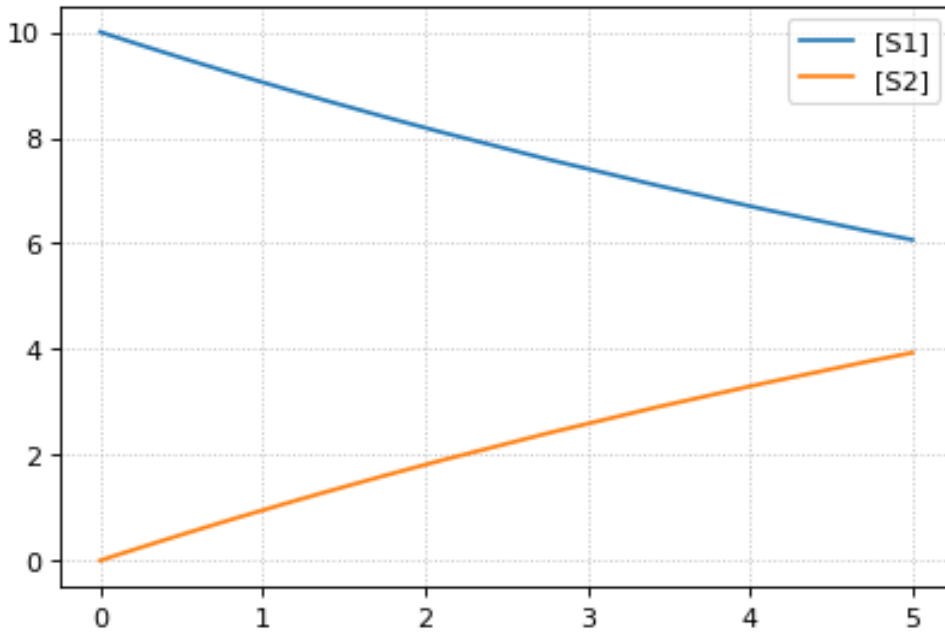
# continue from end concentration of sim1
r.k1 = 2.0
sim2 = r.simulate(0, 5)
print('-- sim2 --')
print('continue simulation from final concentrations with changed parameter')
r.plot(sim2)

# Reset initial concentrations, does not affect the changed parameter
r.reset()
sim3 = r.simulate(0, 5)
print('-- sim3 --')
print('reset initial concentrations but keep changed parameter')
r.plot(sim3)

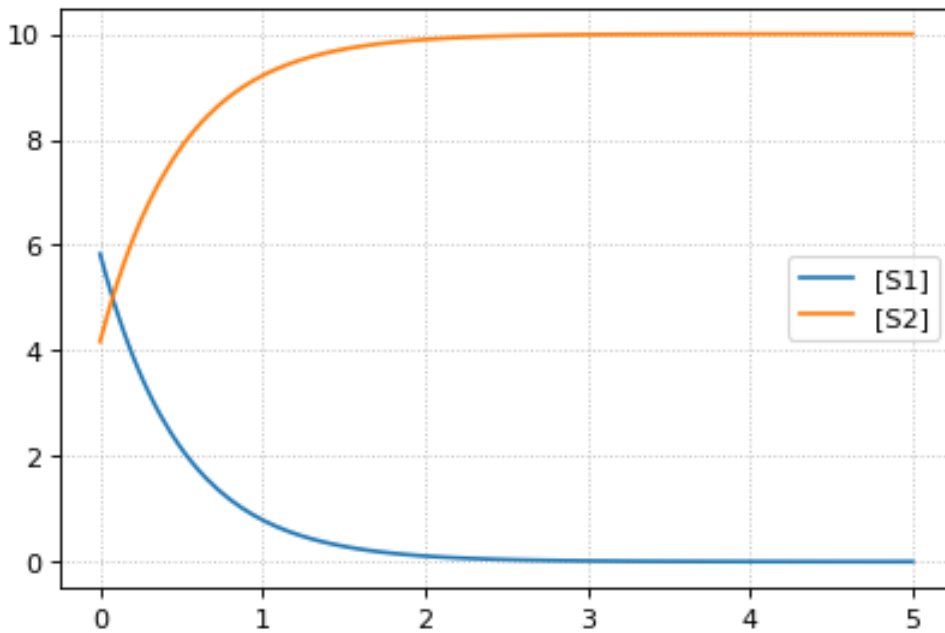
# Change CURRENT initial of k1, resetAll clears parameter but
# resets to CURRENT initial
r.setValue('init(k1)', 0.3)
r.resetAll()
sim4 = r.simulate(0, 5)
print('-- sim4 --')
print('reset to CURRENT initial of k1, reset to initial parameters')
print('k1 = ' + str(r.k1))
r.plot(sim4)

# Reset model to the state it was loaded
r.resetToOrigin()
sim5 = r.simulate(0, 5)
print('-- sim5 --')
print('reset all to origin')
r.plot(sim5);
```

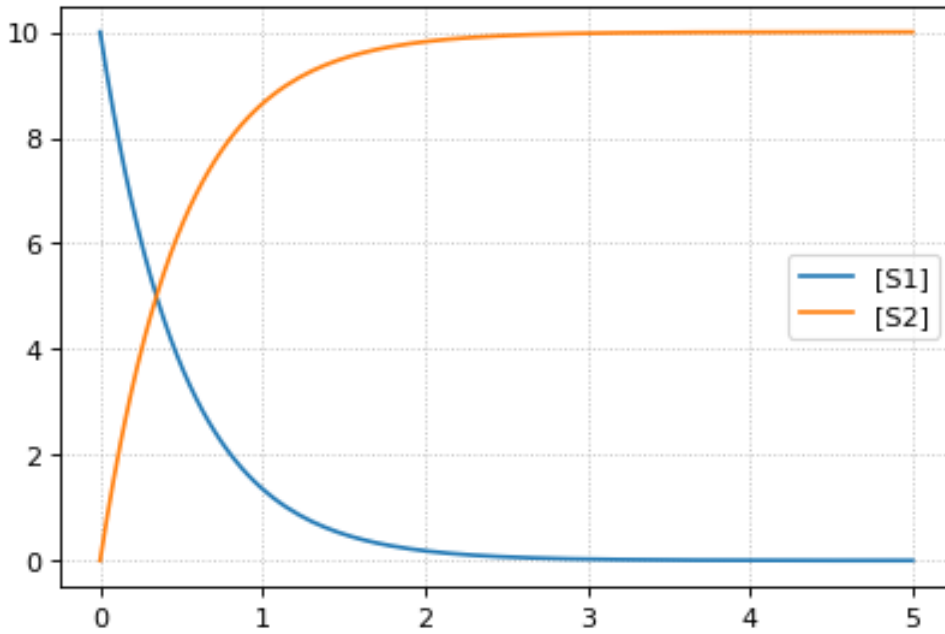
*** sim1 ***



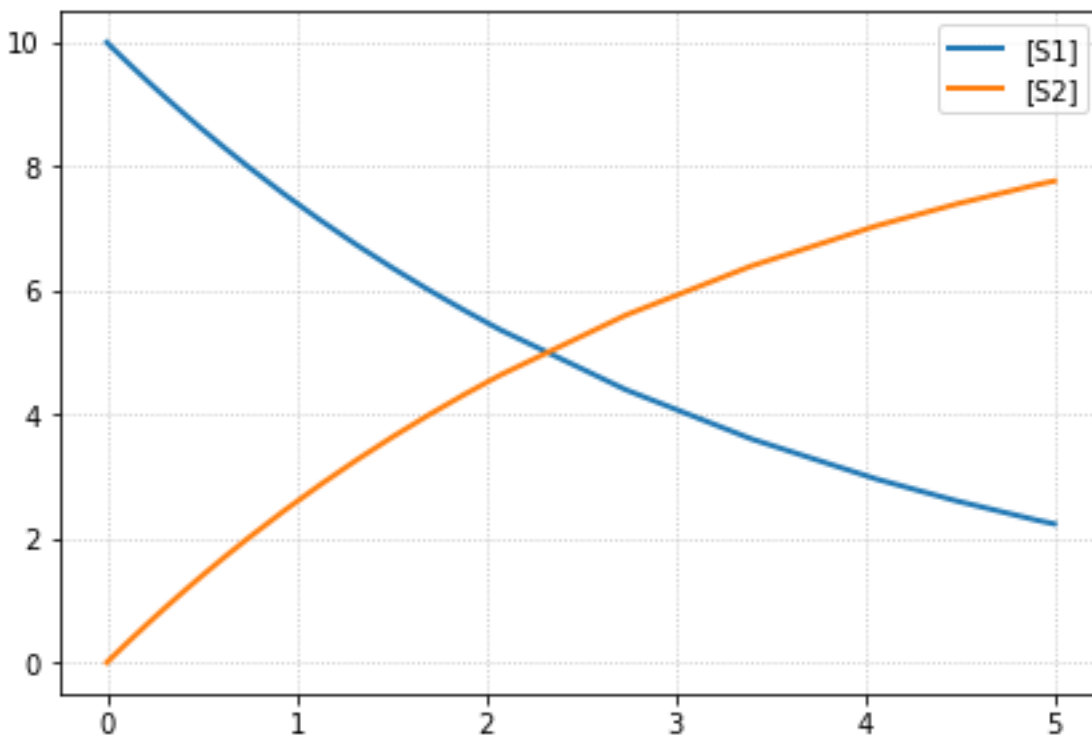
```
-- sim2 --  
continue simulation from final concentrations with changed parameter
```



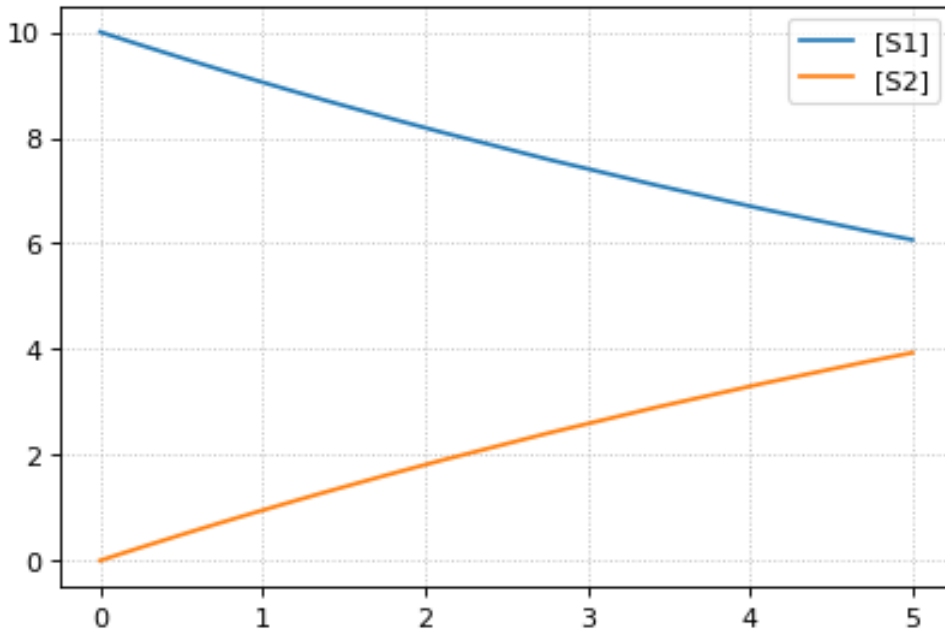
```
-- sim3 --  
reset initial concentrations but keep changed parameter
```



```
-- sim4 --  
reset to CURRENT initial of k1, reset to initial parameters  
k1 = 0.3
```



```
-- sim5 --  
reset all to origin
```



7.11 jarnac Short-cuts

Routines to support the Jarnac compatibility layer

```
class tellurium.tellurium.ExtendedRoadRunner (*args, **kwargs)
```

bs()

ExecutableModel.getBoundarySpeciesIds()

Returns a vector of boundary species Ids.

Parameters **index** (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns a list of boundary species ids.

dv()

RoadRunner::getRatesOfChange()

Returns the rates of change of all floating species. The order of species is given by the order of Ids returned by getFloatingSpeciesIds()

Returns a named array of floating species rates of change.

Return type *numpy.ndarray*

fjac()

RoadRunner.getFullJacobian()

Compute the full Jacobian at the current operating point.

This is the Jacobian of ONLY the floating species.

fs()

ExecutableModel.getFloatingSpeciesIds()

Return a list of floating species sbml ids.

ps()
ExecutableModel.getGlobalParameterIds([index])

Return a list of global parameter ids.

Returns a list of global parameter ids.

rs()
ExecutableModel.getReactionIds()

Returns a vector of reaction Ids.

Parameters **index** (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns a list of reaction ids.

rv()
ExecutableModel.getReactionRates([index])

Returns a vector of reaction rates (reaction velocity) for the current state of the model. The order of reaction rates is given by the order of Ids returned by getReactionIds()

Parameters **index** (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of reaction rates.

Return type *numpy.ndarray*

sm()
RoadRunner.getFullStoichiometryMatrix()

Get the stoichiometry matrix that corresponds to the full model, even if it was converted via conservation conversion.

sv()
ExecutableModel.getFloatingSpeciesConcentrations([index])

Returns a vector of floating species concentrations. The order of species is given by the order of Ids returned by getFloatingSpeciesIds()

Parameters **index** (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of floating species concentrations.

Return type *numpy.ndarray*

vs()
ExecutableModel.getCompartmentIds([index])

Returns a vector of compartment identifier symbols.

Parameters **index** (*None or numpy.ndarray*) – A array of compartment indices indicating which compartment ids to return.

Returns a list of compartment ids.

7.12 Test Models

RoadRunner has built into it a number of predefined models that can be use to easily try and test tellurium.

`tellurium.loadTestModel(string)`
 Loads particular test model into roadrunner.

```
rr = te.loadTestModel('feedback.xml')
```

Returns RoadRunner instance with test model loaded

`tellurium.getTestModel(string)`
 SBML of given test model as a string.

```
# load test model as SBML
sbml = te.getTestModel('feedback.xml')
r = te.loadSBMLModel(sbml)
# simulate
r.simulate(0, 100, 20)
```

Returns SBML string of test model

`tellurium.listTestModels()`
 List roadrunner SBML test models.

```
print(te.listTestModels())
```

Returns list of test model paths

7.12.1 Test models

```
import tellurium as te

# To get the builtin models use listTestModels
print(te.listTestModels())
```

```
['EcoliCore.xml', 'linearPathwayClosed.xml', 'test_1.xml', 'linearPathwayOpen.xml',
↪ 'feedback.xml']
```

Load test model

```
# To load one of the test models use loadTestModel:
# r = te.loadTestModel('feedback.xml')
# result = r.simulate(0, 10, 100)
# r.plot(result)

# If you need to obtain the SBML for the test model, use getTestModel
sbml = te.getTestModel('feedback.xml')

# To look at one of the test model in Antimony form:
ant = te.sbmlToAntimony(te.getTestModel('feedback.xml'))
print(ant)
```

```
// Created by libAntimony v2.9.4
model *feedback()

// Compartments and Species:
compartment compartment_;
species S1 in compartment_, S2 in compartment_, S3 in compartment_, S4 in_
compartment_;
species $X0 in compartment_, $X1 in compartment_;

// Reactions:
J0: $X0 => S1; J0_VM1*(X0 - S1/J0_Keq1)/(1 + X0 + S1 + S4^J0_h);
J1: S1 => S2; (10*S1 - 2*S2)/(1 + S1 + S2);
J2: S2 => S3; (10*S2 - 2*S3)/(1 + S2 + S3);
J3: S3 => S4; (10*S3 - 2*S4)/(1 + S3 + S4);
J4: S4 => $X1; J4_V4*S4/(J4_KS4 + S4);

// Species initializations:
S1 = 0;
S2 = 0;
S3 = 0;
S4 = 0;
X0 = 10;
X1 = 0;

// Compartment initializations:
compartment_ = 1;

// Variable initializations:
J0_VM1 = 10;
J0_Keq1 = 10;
J0_h = 10;
J4_V4 = 2.5;
J4_KS4 = 0.5;

// Other declarations:
const compartment_, J0_VM1, J0_Keq1, J0_h, J4_V4, J4_KS4;
end
```

7.13 Running external tools

Routines to run external tools.

`tellurium.runTool` (*toolFileName*)

Call an external application called *toolFileName*. Note that .exe extension may be omitted for windows applications.

Include any arguments in *arguments* parameter.

Example: `returnString = te.runTool(['myplugin', 'arg1', 'arg2'])`

If the external tool writes to stdout, this will be captured and returned.

Parameters `toolFileName` – argument to external tool

Returns String return by external tool, if any.

7.14 Model Methods

Routines flattened from model, saves typing and easier for finding the methods

class tellurium.tellurium.**ExtendedRoadRunner** (*args, **kwargs)

getBoundarySpeciesConcentrations ([*index*])

Returns a vector of boundary species concentrations. The order of species is given by the order of Ids returned by getBoundarySpeciesIds()

Parameters *index* (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of the boundary species concentrations.

Return type *numpy.ndarray*.

given by the order of Ids returned by getBoundarySpeciesIds()

Parameters *index* (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of the boundary species concentrations.

Return type *numpy.ndarray*.

getBoundarySpeciesIds ()

Returns a vector of boundary species Ids.

Parameters *index* (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns a list of boundary species ids.

getCompartmentIds ([*index*])

Returns a vector of compartment identifier symbols.

Parameters *index* (*None or numpy.ndarray*) – A array of compartment indices indicating which compartment ids to return.

Returns a list of compartment ids.

getCompartmentVolumes ([*index*])

Returns a vector of compartment volumes. The order of volumes is given by the order of Ids returned by getCompartmentIds()

Parameters *index* (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of compartment volumes.

Return type *numpy.ndarray*.

getConservedMoietyValues ([*index*])

Returns a vector of conserved moiety volumes. The order of values is given by the order of Ids returned by getConservedMoietyIds()

Parameters *index* (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of conserved moiety values.

Return type *numpy.ndarray*.

getFloatingSpeciesConcentrations (*[index]*)

Returns a vector of floating species concentrations. The order of species is given by the order of Ids returned by `getFloatingSpeciesIds()`

Parameters *index* (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of floating species concentrations.

Return type *numpy.ndarray*

getFloatingSpeciesIds ()

Return a list of floating species sbml ids.

getGlobalParameterIds (*[index]*)

Return a list of global parameter ids.

Returns a list of global parameter ids.

getGlobalParameterValues (*[index]*)

Return a vector of global parameter values. The order of species is given by the order of Ids returned by `getGlobalParameterIds()`

Parameters *index* (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of global parameter values.

Return type *numpy.ndarray*.

getNumBoundarySpecies ()

Returns the number of boundary species in the model.

getNumCompartments ()

Returns the number of compartments in the model.

Return type *int*

getNumConservedMoieties ()

Returns the number of conserved moieties in the model.

Return type *int*

getNumFloatingSpecies ()

Returns the number of floating species in the model.

getNumGlobalParameters ()

Returns the number of global parameters in the model.

getNumReactions ()

Returns the number of reactions in the model.

getRatesOfChange ()

Returns the rates of change of all floating species. The order of species is given by the order of Ids returned by `getFloatingSpeciesIds()`

Returns a named array of floating species rates of change.

Return type *numpy.ndarray*

getReactionIds ()

Returns a vector of reaction Ids.

Parameters *index* (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns a list of reaction ids.

getReactionRates (*[index]*)

Returns a vector of reaction rates (reaction velocity) for the current state of the model. The order of reaction rates is given by the order of Ids returned by `getReactionIds()`

Parameters **index** (*numpy.ndarray*) – (optional) an index array indicating which items to return.

Returns an array of reaction rates.

Return type `numpy.ndarray`

Main tellurium entry point.

The module tellurium provides support routines. As part of this module an ExtendedRoadRunner class is defined which provides helper methods for model export, plotting or the Jarnac compatibility layer.

```
tellurium.tellurium.DumpJSONInfo()
```

Tellurium dist info. Goes into COMBINE archive.

```
tellurium.tellurium.VersionDict()
```

Return dict of version strings.

```
tellurium.tellurium.addFileToCombineArchive(archive_path, file_name, entry_location,  
                                             file_format, master, out_archive_path)
```

Add a file to an existing COMBINE archive on disk and save the result as a new archive.

Parameters

- **archive_path** – The path to the archive.
- **file_name** – The name of the file to add.
- **entry_location** – The location to store the entry in the archive.
- **file_format** – The format of the file. Can use `tecombine.KnownFormats.lookupFormat` for common formats.
- **master** – Whether the file should be marked master.
- **out_archive_path** – The path to the output archive.

```
tellurium.tellurium.addFilesToCombineArchive(archive_path, file_names, entry_locations,  
                                             file_formats, master_attributes,  
                                             out_archive_path)
```

Add multiple files to an existing COMBINE archive on disk and save the result as a new archive.

Parameters

- **archive_path** – The path to the archive.
- **file_names** – List of extra files to add.

- **entry_locations** – List of destination locations for the files in the output archive.
- **file_format** – List of formats for the resp. files.
- **master_attributes** – List of true/false values for the resp. master attributes of the files.
- **out_archive_path** – The path to the output archive.

`tellurium.tellurium.antimonyToCellML(ant)`

Convert Antimony to CellML string.

Parameters *ant* (*str* | *file*) – Antimony string or file

Returns CellML

Return type *str*

`tellurium.tellurium.antimonyToSBML(ant)`

Convert Antimony to SBML string.

Parameters *ant* (*str* | *file*) – Antimony string or file

Returns SBML

Return type *str*

`tellurium.tellurium.cellmlToAntimony(cellml)`

Convert CellML to antimony string.

Parameters *cellml* (*str* | *file*) – CellML string or file

Returns antimony

Return type *str*

`tellurium.tellurium.cellmlToSBML(cellml)`

Convert CellML to SBML string.

Parameters *cellml* (*str* | *file*) – CellML string or file

Returns SBML

Return type *str*

`tellurium.tellurium.colorCycle(color, polyNumber)`

Adjusts contents of self.color as needed for plotting methods.

`tellurium.tellurium.convertAndExecuteCombineArchive(location)`

Read and execute a COMBINE archive.

Parameters *location* – Filesystem path to the archive.

`tellurium.tellurium.convertCombineArchive(location)`

Read a COMBINE archive and convert its contents to an inline Omex.

Parameters *location* – Filesystem path to the archive.

`tellurium.tellurium.createCombineArchive(archive_path, file_names, entry_locations,
 file_formats, master_attributes, description=None)`

Create a new COMBINE archive containing the provided entries and locations.

Parameters

- **archive_path** – The path to the archive.
- **file_names** – List of extra files to add.

- **entry_locations** – List of destination locations for the files in the output archive.
- **file_format** – List of formats for the resp. files.
- **master_attributes** – List of true/false values for the resp. master attributes of the files.
- **out_archive_path** – The path to the output archive.
- **description** – A libcombine description structure to be assigned to the combine archive, if desired.

`tellurium.tellurium.executeInlineOmex (inline_omex, comp=False)`
Execute inline phrasedml and antimony.

Parameters `inline_omex` – String containing inline phrasedml and antimony.

`tellurium.tellurium.executeInlineOmexFromFile (filepath)`
Execute inline OMEX with simulations described in phrasedml and models described in antimony.

Parameters `filepath` – Path to file containing inline phrasedml and antimony.

`tellurium.tellurium.exportInlineOmex (inline_omex, export_location)`
Export an inline OMEX string to a COMBINE archive.

Parameters

- **inline_omex** – String containing inline OMEX describing models and simulations.
- **export_location** – Filepath of Combine archive to create.

`tellurium.tellurium.extractFileFromCombineArchive (archive_path, entry_location)`
Extract a single file from a COMBINE archive and return it as a string.

`tellurium.tellurium.getDefaultPlottingEngine ()`
Get the default plotting engine. Options are 'matplotlib' or 'plotly'. :return:

`tellurium.tellurium.getEigenvalues (m)`
Eigenvalues of matrix.

Convenience method for computing the eigenvalues of a matrix `m` Uses numpy eig to compute the eigenvalues.

Parameters `m` – numpy array

Returns numpy array containing eigenvalues

`tellurium.tellurium.getLastReport ()`
Get the last report generated by SED-ML.

`tellurium.tellurium.getTelluriumVersion ()`
Version number of tellurium.

Returns version

Return type str

`tellurium.tellurium.getTestModel (string)`
SBML of given test model as a string.

```
# load test model as SBML
sbml = te.getTestModel('feedback.xml')
r = te.loadSBMLModel(sbml)
# simulate
r.simulate(0, 100, 20)
```

Returns SBML string of test model

`tellurium.tellurium.getVersionInfo()`

Returns version information for tellurium included packages.

Returns list of tuples (package, version)

`tellurium.tellurium.inIPython()`

Checks if tellurium is used in IPython.

Returns true if tellurium is being using in an IPython environment, false otherwise. :return: boolean

`tellurium.tellurium.listTestModels()`

List roadrunner SBML test models.

```
print(te.listTestModels())
```

Returns list of test model paths

`tellurium.tellurium.loadAntimonyModel(ant)`

Load Antimony model with tellurium.

See also: `loada()`

```
r = te.loadAntimonyModel('S1 -> S2; k1*S1; k1=0.1; S1=10.0; S2 = 0.0')
```

Parameters `ant` (*str* / *file*) – Antimony model

Returns RoadRunner instance with model loaded

Return type `roadrunner.ExtendedRoadRunner`

`tellurium.tellurium.loadCellMLModel(cellml)`

Load CellML model with tellurium.

Parameters `cellml` (*str* / *file*) – CellML model

Returns RoadRunner instance with model loaded

Return type `roadrunner.ExtendedRoadRunner`

`tellurium.tellurium.loadSBMLModel(sbml)`

Load SBML model from a string or file.

Parameters `sbml` (*str* / *file*) – SBML model

Returns RoadRunner instance with model loaded

Return type `roadrunner.ExtendedRoadRunner`

`tellurium.tellurium.loadTestModel(string)`

Loads particular test model into roadrunner.

```
rr = te.loadTestModel('feedback.xml')
```

Returns RoadRunner instance with test model loaded

`tellurium.tellurium.loada(ant)`

Load model from Antimony string.

See also: `loadAntimonyModel()`

```
r = te.loada('S1 -> S2; k1*S1; k1=0.1; S1=10.0; S2 = 0.0')
```

Parameters `ant` (*str* | *file*) – Antimony model

Returns RoadRunner instance with model loaded

Return type roadrunner.ExtendedRoadRunner

`tellurium.tellurium.loads` (*ant*)

Load SBML model with tellurium

See also: `loadSBMLModel()`

Parameters `ant` (*str* | *file*) – SBML model

Returns RoadRunner instance with model loaded

Return type roadrunner.ExtendedRoadRunner

`tellurium.tellurium.model` (*model_name*)

Retrieve a model which has already been loaded.

Parameters `model_name` (*str*) – the name of the model

`tellurium.tellurium.noticesOff` ()

Switch off the generation of notices to the user. Call this to stop roadrunner from printing warning message to the console.

See also `noticesOn()`

`tellurium.tellurium.noticesOn` ()

Switch on notice generation to the user.

See also `noticesOff()`

`tellurium.tellurium.plotArray` (*result*, *loc*='upper right', *show*=True, *resetColorCycle*=True, *xlabel*=None, *ylabel*=None, *title*=None, *xlim*=None, *ylim*=None, *xscale*='linear', *yscale*='linear', *grid*=False, *labels*=None, ***kwargs*)

Plot an array.

Parameters

- **result** – Array to plot, first column of the array must be the x-axis and remaining columns the y-axis
- **loc** (*str*) – Location of legend box. Valid strings 'best' | 'upper right' | 'upper left' | 'lower left' | 'lower right' | 'right' | 'center left' | 'center right' | 'lower center' | 'upper center' | 'center' |
- **color** (*str*) – 'red', 'blue', etc. to use the same color for every curve
- **labels** – A list of labels for the legend, include as many labels as there are curves to plot
- **xlabel** (*str*) – x-axis label
- **ylabel** (*str*) – y-axis label
- **title** (*str*) – Add plot title
- **xlim** – Limits on x-axis (tuple [start, end])
- **ylim** – Limits on y-axis
- **xscale** – 'linear' or 'log' scale for x-axis

- **yscale** – ‘linear’ or ‘log’ scale for y-axis
- **grid** (*bool*) – Show grid
- **show** – show=True (default) shows the plot, use show=False to plot multiple simulations in one plot
- **resetColorCycle** (*bool*) – If true, resets color cycle on given figure (works with show=False to plot multiple simulations on a single plot)
- **kwargs** – Additional matplotlib keywords like linewidth, linestyle...

```
import numpy as np, tellurium as te
result = np.array([[1,2,3], [7.2,6.5,8.8], [9.8, 6.5, 4.3]])
te.plotArray(result, title="My graph", xlim=(1, 5), labels=["Label 1", "Label 2
↪"],
              yscale='log', linestyle='dashed')
```

`tellurium.tellurium.plotWithLegend(r, result=None, loc='upper right', show=True)`

Plot an array and include a legend. The first argument must be a roadrunner variable. The second argument must be an array containing data to plot. The first column of the array will be the x-axis and remaining columns the y-axis. Returns a handle to the plotting object.

`plotWithLegend(r)`

`tellurium.tellurium.printVersionInfo()`

Prints version information for tellurium included packages.

see also: `getVersionInfo()`

`tellurium.tellurium.sbmlToAntimony(sbml)`

Convert SBML to antimony string.

Parameters *sbml* (*str* | *file*) – SBML string or file

Returns Antimony

Return type *str*

`tellurium.tellurium.sbmlToCellML(sbml)`

Convert SBML to CellML string.

Parameters *sbml* (*str* | *file*) – SBML string or file

Returns CellML

Return type *str*

`tellurium.tellurium.setDefaultPlottingEngine(engine)`

Set the default plotting engine. Overrides current value.

Parameters *engine* – A string describing which plotting engine to use. Valid values are ‘matplotlib’ and ‘plotly’.

`tellurium.tellurium.setLastReport(report)`

Used by SED-ML to save the last report created (for validation).

`tellurium.tellurium.setSavePlotsToPDF(value)`

Sets whether plots should be saved to PDF.

9.1 Source Code Repositories

Tellurium is a collection of Python packages developed inside and outside our group, including simulators, libraries for reading and writing standards like SBML and SED-ML, and various utilities (e.g. [sbml2matlab](#)). Tellurium itself is a Python module that provides integration between these various subpackages and its source code is [hosted on GitHub](#). A list of constituent packages and repositories is given below:

- [tellurium](#): This project.
- [libroadrunner](#): SBML ODE / stochastic simulator.
- [antimony](#): A human-readable representation of SBML.
- [phrasedml](#): A human-readable representation of SED-ML.
- [libcombine](#): A library for reading/writing COMBINE archives.
- [sbml2matlab](#): A utility for converting SBML models to MATLAB ODE simulations.
- [simplesbml](#): A utility for creating SBML models without the complexity of libSBML.
- [libsbml](#): A library for reading/writing [SBML](#).
- [libsedml](#): A library for reading/writing SED-ML.

9.2 License

The [Tellurium source code](#) is licensed under the Apache License 2.0. Tellurium uses third-party dependencies which may be licensed under different terms. Consult the documentation for the respective third-party packages for more details.

9.3 Contact

- For questions about Tellurium, please visit our [tellurium-discuss Google group](#).
- For new feature requests or to report bugs, see [Tellurium on GitHub](#). and create an issue.

9.4 Funding

The Tellurium project is funded by generous support from **NIH/NIGMS** grant GM081070. The content is solely the responsibility of the authors and does not necessarily represent the views of the National Institutes of Health.

9.5 Acknowledgments

Tellurium relies on many great open-source projects, including the [Spyder IDE](#), [nteract](#), [Python](#), [numpy](#), [Jupyter](#), [CVODE](#), [NLEQ](#), [AUTO2000](#), [LAPACK](#), [LLVM](#), and [the POCO libraries](#). Tellurium also relies on open source contributions from Frank Bergmann ([libStruct](#), [libSEDML](#)), Mike Hucka, Sarah Keating ([libSBML](#)), and Pierre Raybaut.

CHAPTER 10

Indices and tables

- `genindex`
- `modindex`
- `search`

A

`addFilesToCombineArchive()` (in module *tellurium.tellurium*), 187
`addFileToCombineArchive()` (in module *tellurium.tellurium*), 187
`antimonyToCellML()` (in module *tellurium*), 142
`antimonyToCellML()` (in module *tellurium.tellurium*), 188
`antimonyToSBML()` (in module *tellurium*), 142
`antimonyToSBML()` (in module *tellurium.tellurium*), 188

B

`bs()` (*tellurium.tellurium.ExtendedRoadRunner* method), 179

C

`cellmlToAntimony()` (in module *tellurium*), 143
`cellmlToAntimony()` (in module *tellurium.tellurium*), 188
`cellmlToSBML()` (in module *tellurium*), 143
`cellmlToSBML()` (in module *tellurium.tellurium*), 188
`colorCycle()` (in module *tellurium.tellurium*), 188
`convertAndExecuteCombineArchive()` (in module *tellurium.tellurium*), 188
`convertCombineArchive()` (in module *tellurium*), 54
`convertCombineArchive()` (in module *tellurium.tellurium*), 188
`createCombineArchive()` (in module *tellurium.tellurium*), 188

D

`draw()` (*tellurium.tellurium.ExtendedRoadRunner* method), 161
`DumpJSONInfo()` (in module *tellurium.tellurium*), 187
`dv()` (*tellurium.tellurium.ExtendedRoadRunner* method), 179

E

`executeInlineOmex()` (in module *tellurium*), 54
`executeInlineOmex()` (in module *tellurium.tellurium*), 189
`executeInlineOmexFromFile()` (in module *tellurium.tellurium*), 189
`exportInlineOmex()` (in module *tellurium*), 54
`exportInlineOmex()` (in module *tellurium.tellurium*), 189
`exportToAntimony()` (*tellurium.tellurium.ExtendedRoadRunner* method), 144
`exportToCellML()` (*tellurium.tellurium.ExtendedRoadRunner* method), 145
`exportToMatlab()` (*tellurium.tellurium.ExtendedRoadRunner* method), 145
`exportToSBML()` (*tellurium.tellurium.ExtendedRoadRunner* method), 145
`ExtendedRoadRunner` (class in *tellurium.tellurium*), 144, 154, 161, 179, 183
`extractFileFromCombineArchive()` (in module *tellurium*), 54
`extractFileFromCombineArchive()` (in module *tellurium.tellurium*), 189

F

`fjac()` (*tellurium.tellurium.ExtendedRoadRunner* method), 179
`fs()` (*tellurium.tellurium.ExtendedRoadRunner* method), 179

G

`getAntimony()` (*tellurium.tellurium.ExtendedRoadRunner* method), 145
`getBoundarySpeciesConcentrations()`

- (*tellurium.tellurium.ExtendedRoadRunner* method), 183
- getBoundarySpeciesIds()* (*tellurium.tellurium.ExtendedRoadRunner* method), 183
- getCellML()* (*tellurium.tellurium.ExtendedRoadRunner* method), 145
- getCompartmentIds()* (*tellurium.tellurium.ExtendedRoadRunner* method), 183
- getCompartmentVolumes()* (*tellurium.tellurium.ExtendedRoadRunner* method), 183
- getConservedMoietiesValues()* (*tellurium.tellurium.ExtendedRoadRunner* method), 183
- getCurrentAntimony()* (*tellurium.tellurium.ExtendedRoadRunner* method), 145
- getCurrentCellML()* (*tellurium.tellurium.ExtendedRoadRunner* method), 145
- getCurrentMatlab()* (*tellurium.tellurium.ExtendedRoadRunner* method), 145
- getDefaultPlottingEngine()* (*in module tellurium*), 189
- getEigenvalues()* (*in module tellurium*), 158
- getEigenvalues()* (*in module tellurium.tellurium*), 189
- getFloatingSpeciesConcentrations()* (*tellurium.tellurium.ExtendedRoadRunner* method), 183
- getFloatingSpeciesIds()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getGlobalParameterIds()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getGlobalParameterValues()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getLastReport()* (*in module tellurium.tellurium*), 189
- getMatlab()* (*tellurium.tellurium.ExtendedRoadRunner* method), 146
- getNumBoundarySpecies()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getNumCompartments()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getNumConservedMoieties()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getNumFloatingSpecies()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getNumGlobalParameters()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getNumReactions()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getODEsFromModel()* (*in module tellurium*), 159
- getODEsFromSBMLFile()* (*in module tellurium*), 159
- getODEsFromSBMLString()* (*in module tellurium*), 159
- getRatesOfChange()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getReactionIds()* (*tellurium.tellurium.ExtendedRoadRunner* method), 184
- getReactionRates()* (*tellurium.tellurium.ExtendedRoadRunner* method), 185
- getSeed()* (*tellurium.tellurium.ExtendedRoadRunner* method), 154
- getTelluriumVersion()* (*in module tellurium*), 134
- getTelluriumVersion()* (*in module tellurium.tellurium*), 189
- getTestModel()* (*in module tellurium*), 181
- getTestModel()* (*in module tellurium.tellurium*), 189
- getVersionInfo()* (*in module tellurium*), 134
- getVersionInfo()* (*in module tellurium.tellurium*), 190
- gillespie()* (*tellurium.tellurium.ExtendedRoadRunner* method), 155
- I**
- inIPython()* (*in module tellurium.tellurium*), 190
- installPackage()* (*in module tellurium*), 133
- L**
- listTestModels()* (*in module tellurium*), 181
- listTestModels()* (*in module tellurium.tellurium*), 190
- loada()* (*in module tellurium*), 140
- loada()* (*in module tellurium.tellurium*), 190
- loadAntimonyModel()* (*in module tellurium*), 140
- loadAntimonyModel()* (*in module tellurium.tellurium*), 190
- loadCellMLModel()* (*in module tellurium*), 140
- loadCellMLModel()* (*in module tellurium.tellurium*), 190

loads() (in module tellurium.tellurium), 191
 loadSBMLModel() (in module tellurium), 140
 loadSBMLModel() (in module tellurium.tellurium), 190
 loadTestModel() (in module tellurium), 180
 loadTestModel() (in module tellurium.tellurium), 190

M

model() (in module tellurium.tellurium), 191

N

noticesOff() (in module tellurium), 134
 noticesOff() (in module tellurium.tellurium), 191
 noticesOn() (in module tellurium), 134
 noticesOn() (in module tellurium.tellurium), 191
 nullspace() (in module tellurium), 159

P

plot() (in module tellurium), 159
 plot() (tellurium.tellurium.ExtendedRoadRunner method), 161
 plotArray() (in module tellurium), 160
 plotArray() (in module tellurium.tellurium), 191
 plotWithLegend() (in module tellurium.tellurium), 192
 printVersionInfo() (in module tellurium), 134
 printVersionInfo() (in module tellurium.tellurium), 192
 ps() (tellurium.tellurium.ExtendedRoadRunner method), 180

R

rank() (in module tellurium), 158
 readFromFile() (in module tellurium), 134
 rref() (in module tellurium), 159
 rs() (tellurium.tellurium.ExtendedRoadRunner method), 180
 runTool() (in module tellurium), 182
 rv() (tellurium.tellurium.ExtendedRoadRunner method), 180

S

saveToFile() (in module tellurium), 134
 sbmlToAntimony() (in module tellurium), 142
 sbmlToAntimony() (in module tellurium.tellurium), 192
 sbmlToCellML() (in module tellurium), 142
 sbmlToCellML() (in module tellurium.tellurium), 192
 searchPackage() (in module tellurium), 133
 setDefaultPlottingEngine() (in module tellurium.tellurium), 192
 setLastReport() (in module tellurium.tellurium), 192

setSavePlotsToPDF() (in module tellurium.tellurium), 192
 setSeed() (tellurium.tellurium.ExtendedRoadRunner method), 155
 sm() (tellurium.tellurium.ExtendedRoadRunner method), 180
 sv() (tellurium.tellurium.ExtendedRoadRunner method), 180

T

tellurium.tellurium (module), 187

U

uninstallPackage() (in module tellurium), 133
 upgradePackage() (in module tellurium), 133

V

VersionDict() (in module tellurium.tellurium), 187
 vs() (tellurium.tellurium.ExtendedRoadRunner method), 180