

---

# tellurium Documentation

*Release 2.0.5*

**sys-bio**

**Jan 05, 2018**



---

# Contents

---

<b>1</b>	<b>Installation and Front-ends</b>	<b>3</b>
1.1	Installation Options . . . . .	3
<b>2</b>	<b>Quick Start</b>	<b>7</b>
2.1	Simple Example . . . . .	7
2.2	More Complex Example . . . . .	8
<b>3</b>	<b>Walkthrough</b>	<b>11</b>
3.1	Notebook Walkthrough . . . . .	11
3.2	Notebook Troubleshooting . . . . .	11
3.3	IDE Walkthrough . . . . .	13
3.4	Additional Resources for Tellurium . . . . .	13
3.5	Learning Python . . . . .	13
<b>4</b>	<b>Usage Examples</b>	<b>15</b>
4.1	Basics . . . . .	15
4.2	Models & Model Building . . . . .	21
4.3	SED-ML . . . . .	28
4.4	COMBINE & Inline OMEX . . . . .	36
4.5	Modeling Case Studies . . . . .	51
<b>5</b>	<b>Antimony Reference</b>	<b>71</b>
5.1	Background . . . . .	71
5.2	Change Log . . . . .	72
5.3	Contents . . . . .	72
5.4	Introduction & Basics . . . . .	73
5.5	Examples . . . . .	75
5.6	Simulating Models . . . . .	80
5.7	Language Reference . . . . .	84
<b>6</b>	<b>Parallel Programming</b>	<b>103</b>
6.1	Tellurium in Distributed Environment . . . . .	103
6.2	Getting Started . . . . .	103
6.3	Your First Stochastic Simulation . . . . .	104
6.4	Parameter Estimation . . . . .	104
6.5	Parameter Scanning . . . . .	108

<b>7</b>	<b>Tellurium Methods</b>	<b>111</b>
7.1	Installing Packages . . . . .	111
7.2	Utility Methods . . . . .	113
7.3	Model Loading . . . . .	119
7.4	Interconversion Utilities . . . . .	122
7.5	Export Utilities . . . . .	124
7.6	Stochastic Simulation . . . . .	134
7.7	Distributed Stochastic Simulation . . . . .	138
7.8	Distributed Stochastic Simulation Utility . . . . .	138
7.9	Plot Distributed Stochastic Simulation Results . . . . .	138
7.10	Distributed Parameter Scanning . . . . .	139
7.11	Plotting Image of Parameter Scan . . . . .	139
7.12	Distributed Sensitivity Analysis . . . . .	139
7.13	Distributed Sensitivity Analysis Utility . . . . .	139
7.14	Math . . . . .	139
7.15	Plotting . . . . .	139
7.16	Model Reset . . . . .	145
7.17	jarnac Short-cuts . . . . .	149
7.18	Test Models . . . . .	151
7.19	Model Methods . . . . .	152
<b>8</b>	<b>API</b>	<b>157</b>
<b>9</b>	<b>Appendix</b>	<b>163</b>
9.1	Source Code Repositories . . . . .	163
9.2	License . . . . .	163
9.3	Contact . . . . .	164
9.4	Funding . . . . .	164
9.5	Acknowledgments . . . . .	164
<b>10</b>	<b>Indices and tables</b>	<b>165</b>

Model, simulate, and analyse biochemical systems using a single tool.

Contents:



### 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 [nteract 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)

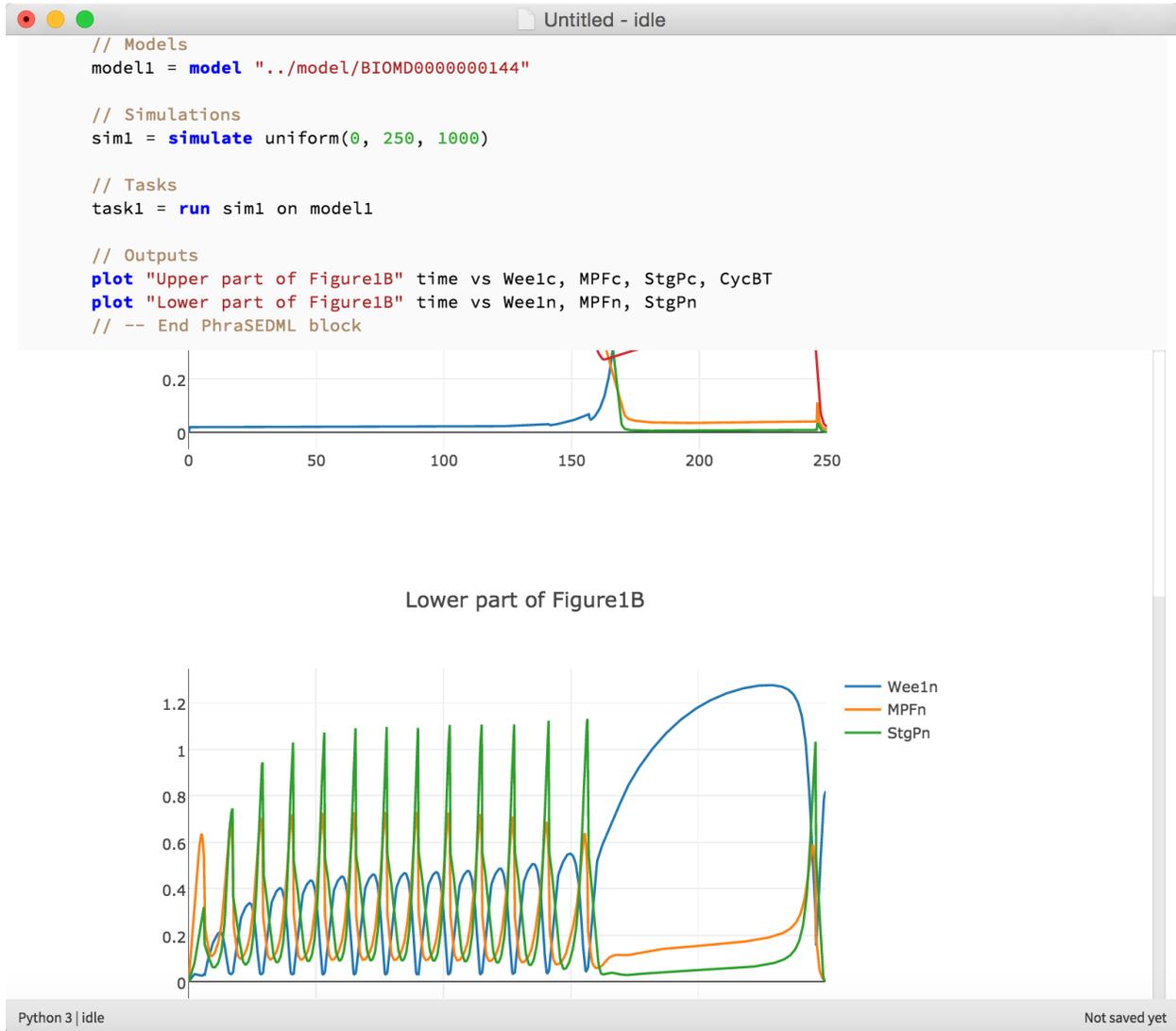


Fig. 1.1: Tellurium notebook offers an environment similar to Jupyter

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

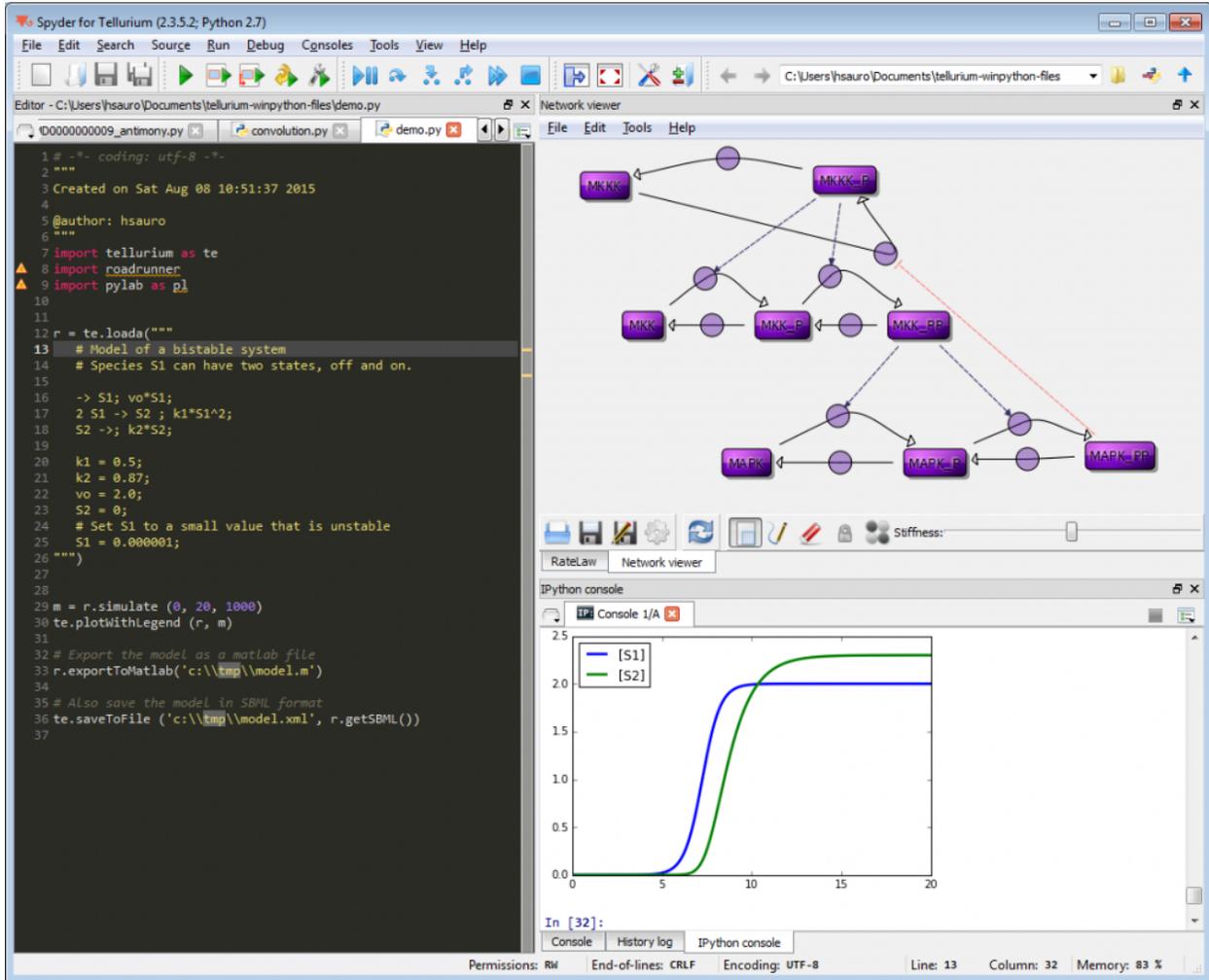


Fig. 1.2: Tellurium IDE features a programmer-centric interface similar to MATLAB

### 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

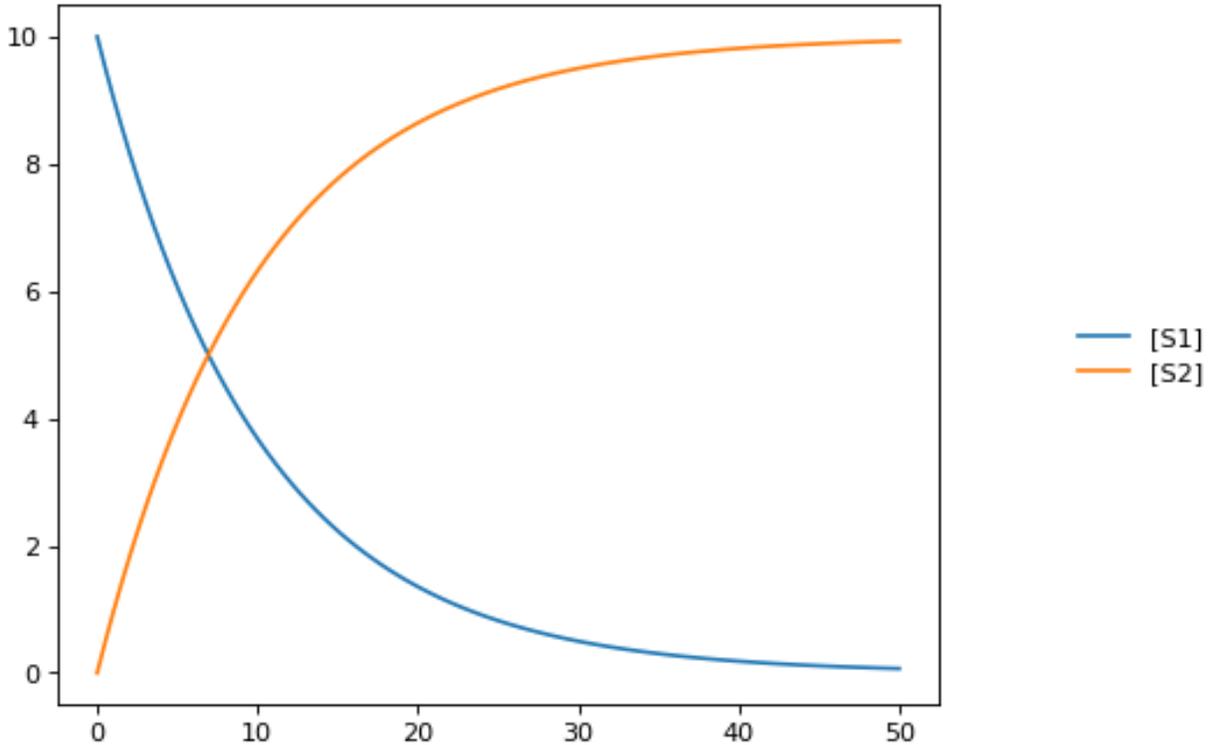
The Tellurium PyPI packages support 64-bit Python versions 2.7, 3.4, 3.5, and 3.6 for Windows, Mac, and Linux. The notebook viewer comes with Python 3.6 (64-bit) and the IDE comes with Python 2.7 (32-bit). If you need support for a Python version not already covered, please [file an issue](#).

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

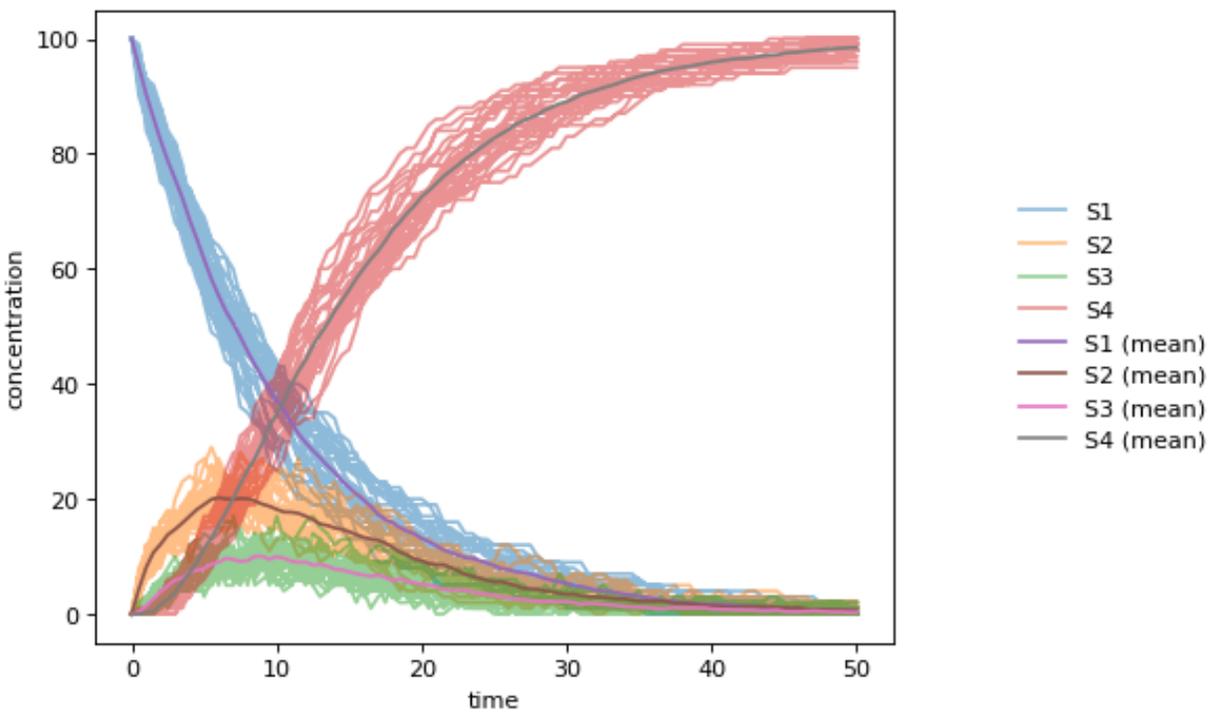
# run repeated simulation
Ncol = len(r.selections)
```

```

Nsim = 30
points = 101
s_sum = np.zeros(shape=[points, Ncol])
for k in range(Nsim):
    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).

- **TODO:** Add how to use Antimony & inline OMEX cells.
- **TODO:** Show how to search/replace with CM and search word boundaries.
- **TODO:** Show how to access example notebooks.
- **TODO:** Demo interactive plotting online with Plotly

### 3.2 Notebook Troubleshooting

#### 3.2.1 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.

#### 3.2.2 Solution

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

#### Further Reading

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

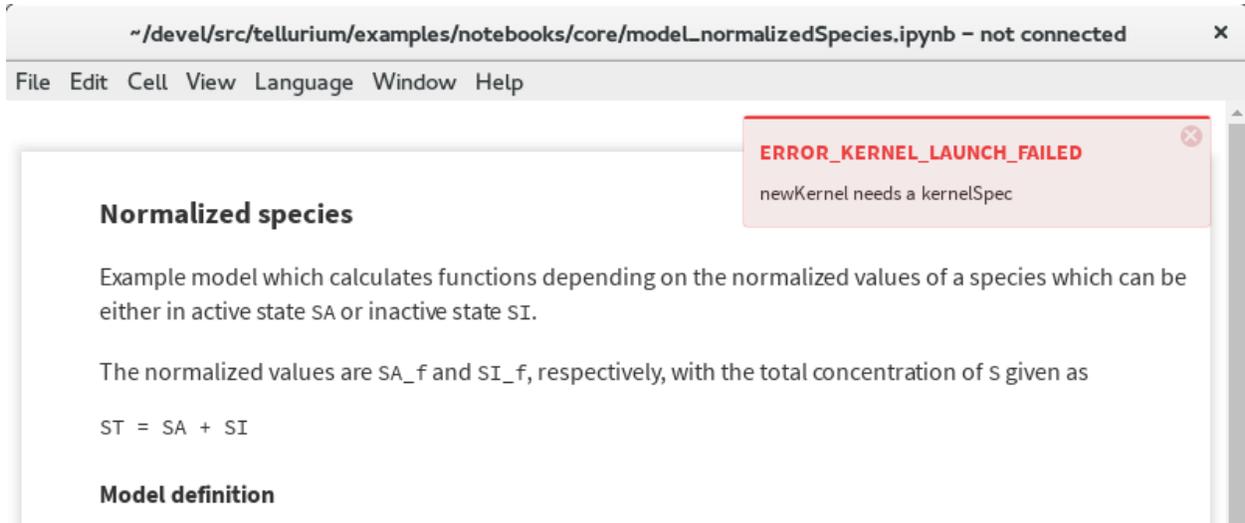


Fig. 3.1: Error message when kernel cannot be loaded

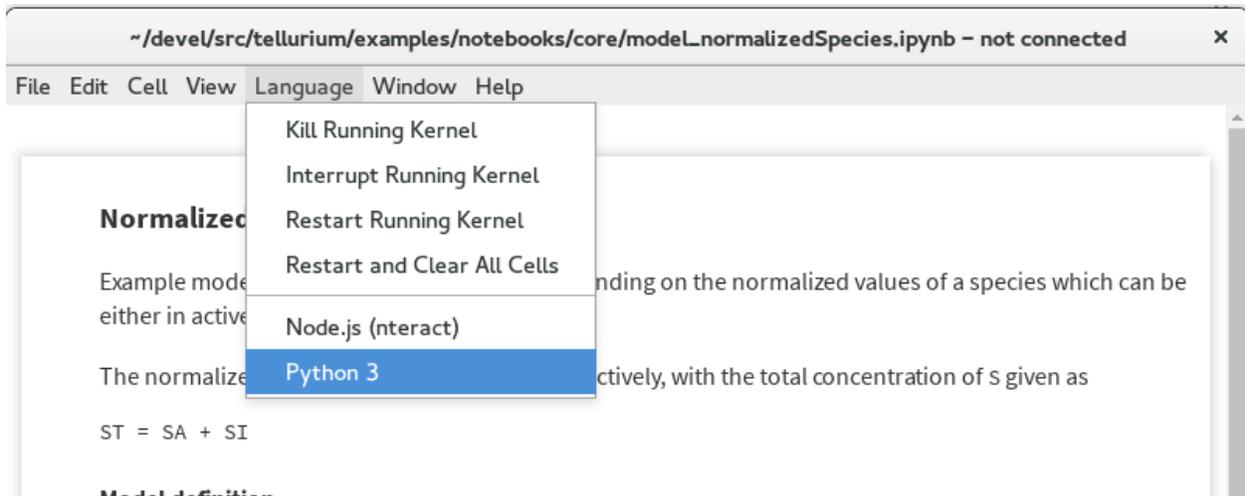


Fig. 3.2: Fix for kernel loading problem

### 3.3 IDE Walkthrough

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

- [Official Spyder documentation](#)
- 

### 3.4 Additional Resources for Tellurium

- [Suggest a new feature for Tellurium with UserVoice.](#)
- [Herbert Sauro's modeling textbook, which uses Tellurium](#)
- [YouTube video tutorials \(made prior to Tellurium notebook\).](#)

### 3.5 Learning Python

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



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.

## 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;
  species S1, S2;

  S1 = 10.0;
```

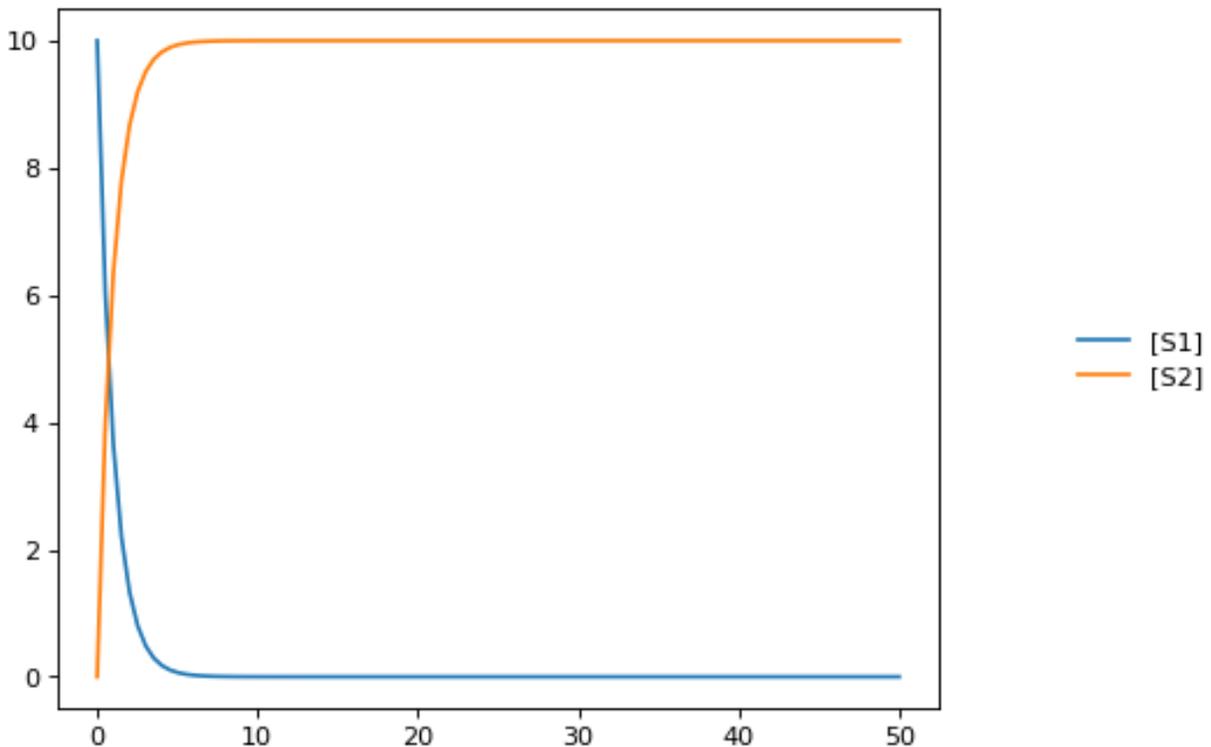
```
S2 = 0.0;
S1 in C1; S2 in C1;
J1: S1 -> S2; k1*S1;

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 'cvode', '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.

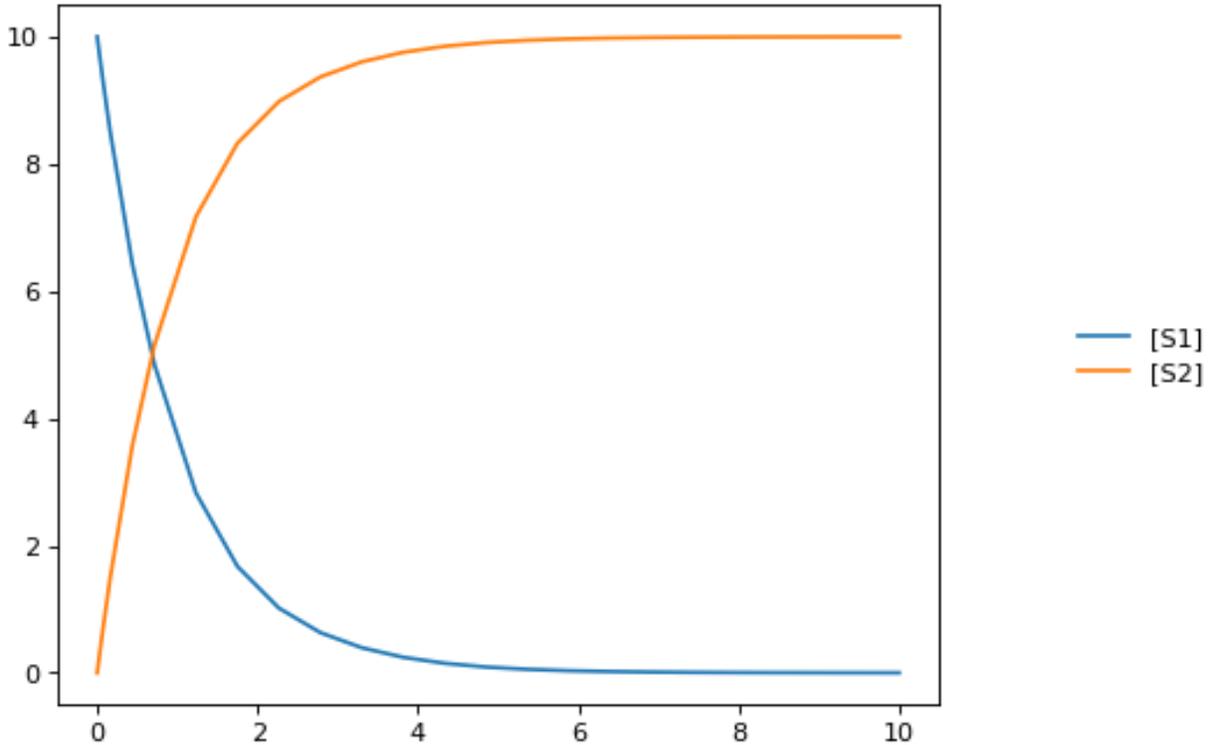
```
# what is the current integrator?
print('The current integrator is:')
print(r.integrator)

# 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
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 either given as argument to `r.simulate`.

```

print('Floating species in model:')
print(r.getFloatingSpeciesIds())
# 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.{} == {}'.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.{} == {}'.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

```

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

## 4.2 Models & Model Building

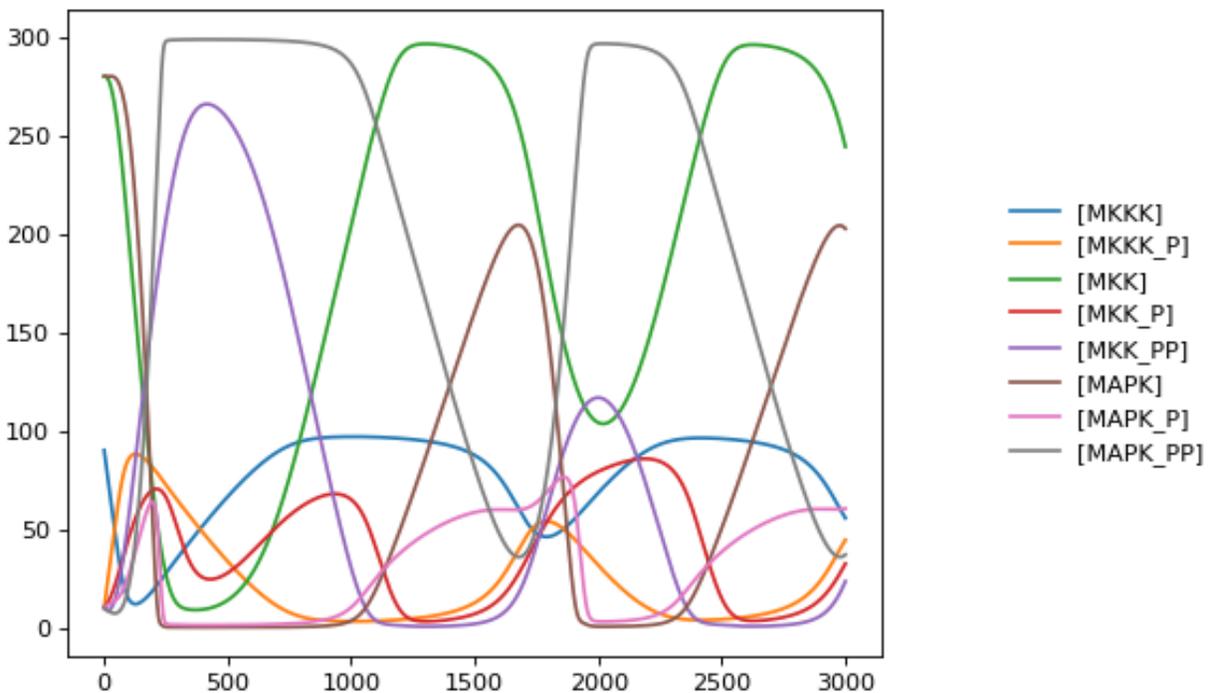
In this section model the creation of example models is shown.

### 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("http://biomodels.caltech.edu/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(''
  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 v2.9.3
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

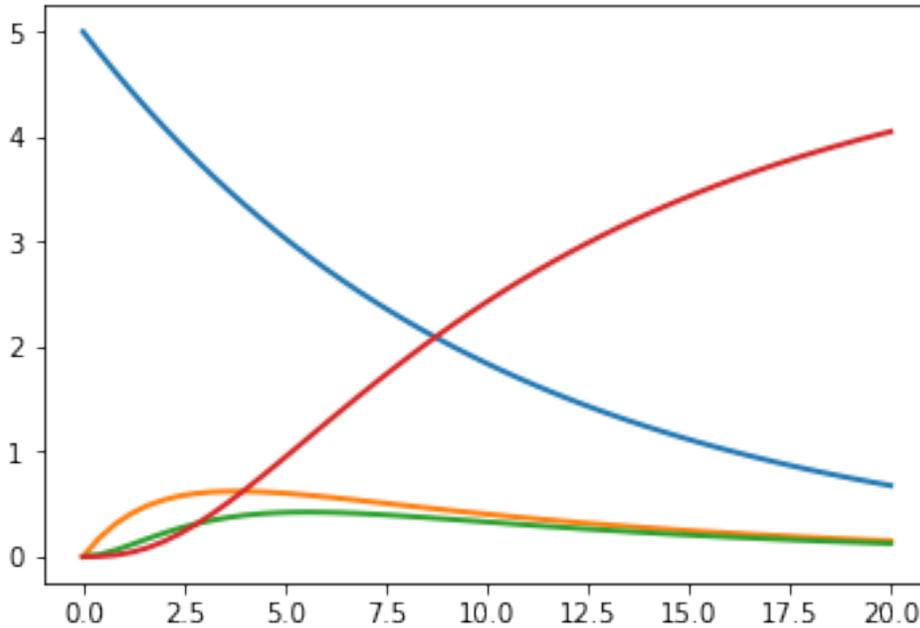
Model creation and simulation of a simple irreversible chain of reactions  $S1 \rightarrow S2 \rightarrow S3 \rightarrow S4$ .

```
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 Feedback oscillations

Model oscillations via feedback

```
import tellurium as te

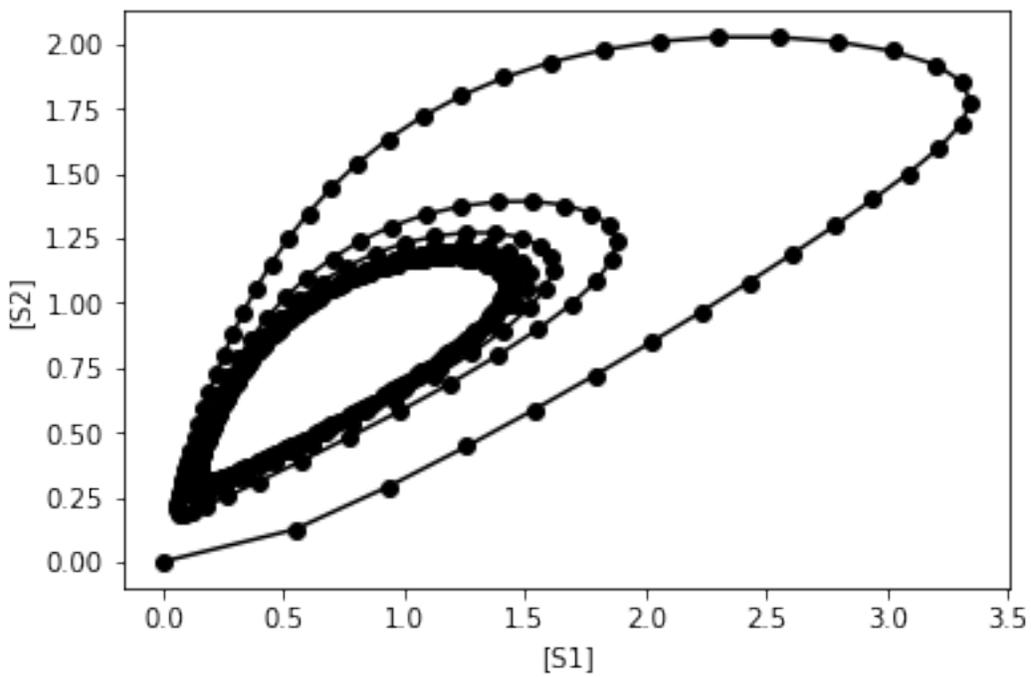
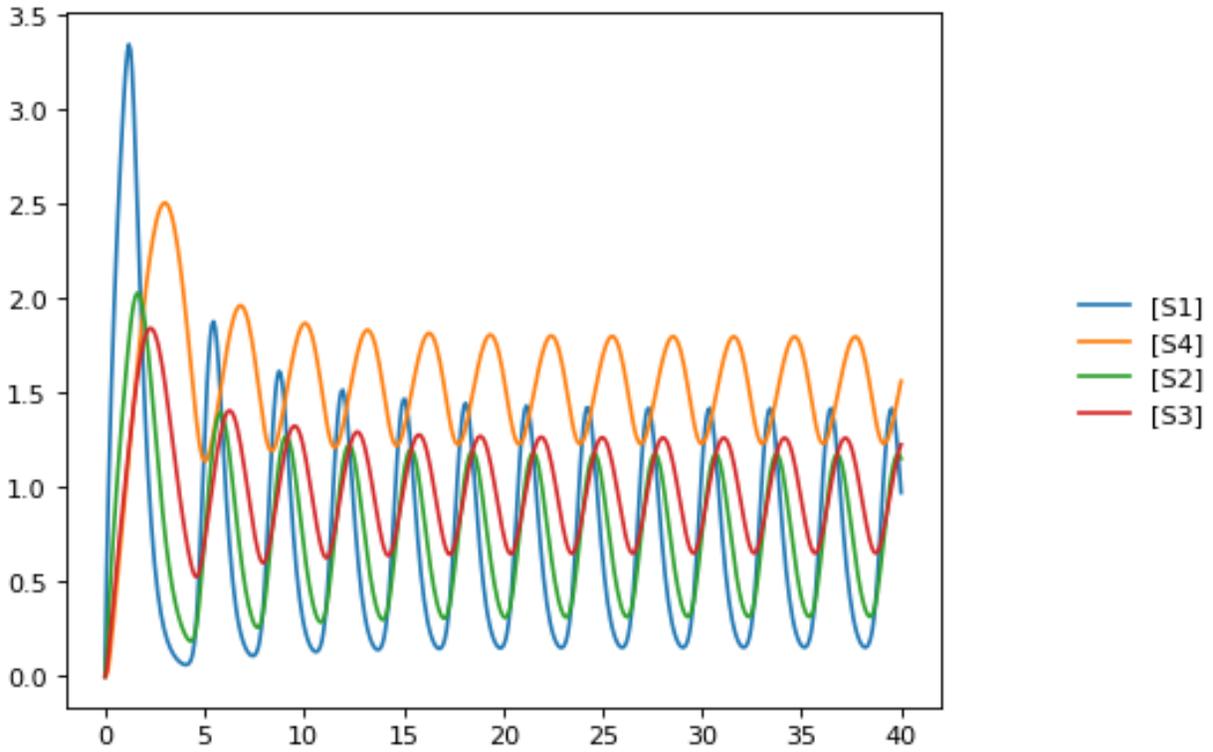
r = te.loada('''
model feedback()
  // Reactions:
  J0: $X0 -> S1; (VM1 * (X0 - S1/Keq1))/(1 + X0 + S1 + S4^h);
  J1: S1 -> S2; (10 * S1 - 2 * S2) / (1 + S1 + S2);
  J2: S2 -> S3; (10 * S2 - 2 * S3) / (1 + S2 + S3);
  J3: S3 -> S4; (10 * S3 - 2 * S4) / (1 + S3 + S4);
  J4: S4 -> $X1; (V4 * S4) / (KS4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

res = r.simulate(0, 40, 500)
r.plot()

import matplotlib.pyplot as plt
plt.plot(res["[S1]"], res["[S2]"], 'o-', color="black")
plt.xlabel("[S1]")
plt.ylabel("[S2]");
```



### 4.2.5 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.LOASBML_OPTIONS_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
  SawTooth = amplitude/2
  SawTooth' = amplitude/period
  at UpDown==0: SawTooth = 0

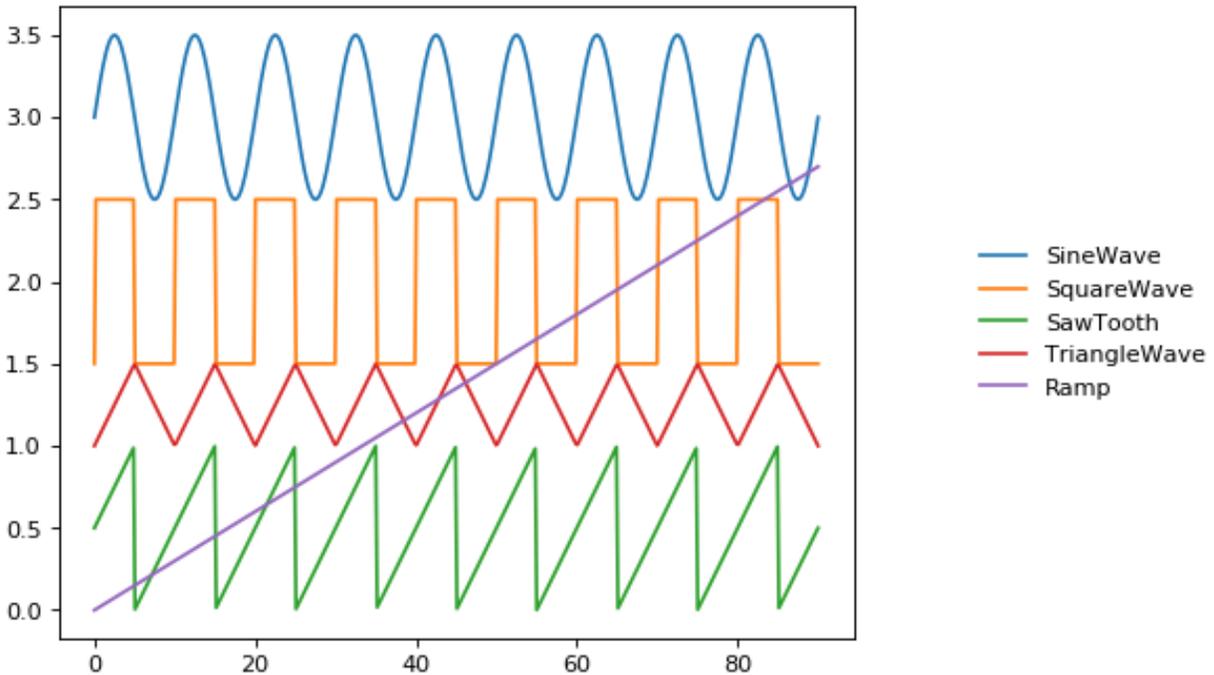
  # Simple ramp
  Ramp := 0.03*time
end
'''

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.LOASBML_OPTIONS_CONSERVED_MOIETIES, False)

```



#### 4.2.6 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

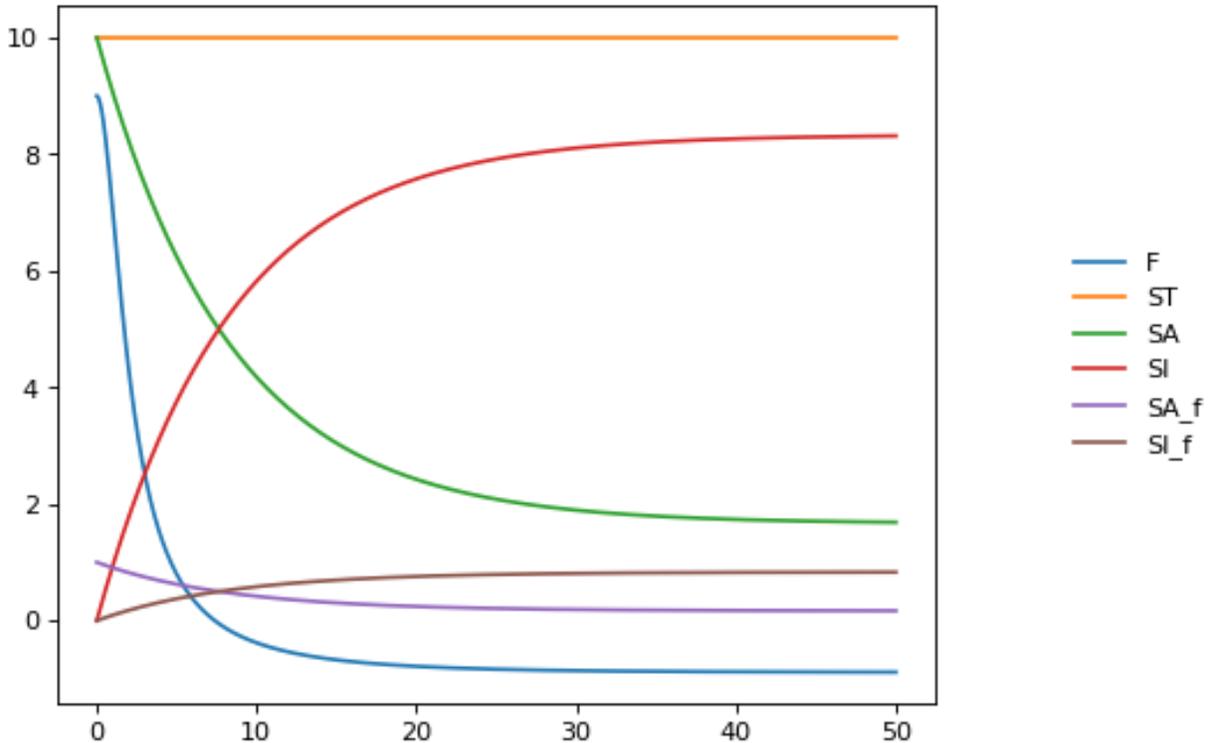
We perform a simple model simulation to demonstrate the main features using `roadrunner`: - normalized values `SA_f` and `SI_f` are normalized in  $[0, 1]$  - the normalized values have same dynamics like `SA` and `SF` - the normalized values can be used to calculates some dependent function, here `F`

```

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 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.).

### 4.3.1 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
  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 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 v1.0.7 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">
      <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>

```

### 4.3.2 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
te.executeSEDML(sedml_str, workingDir=workingDir)

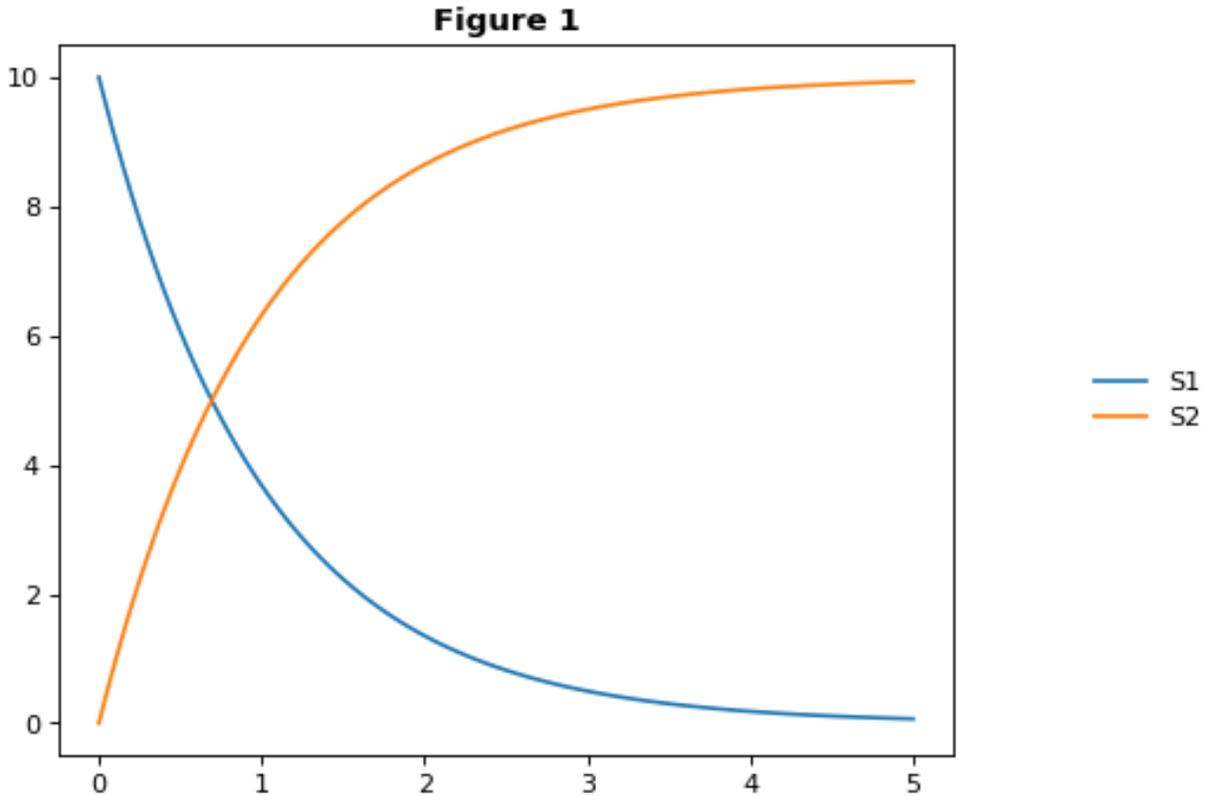
# clean up
#shutil.rmtree(workingDir)

```

```

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

```



### 4.3.3 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)
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.
↳PZ, task2.PY

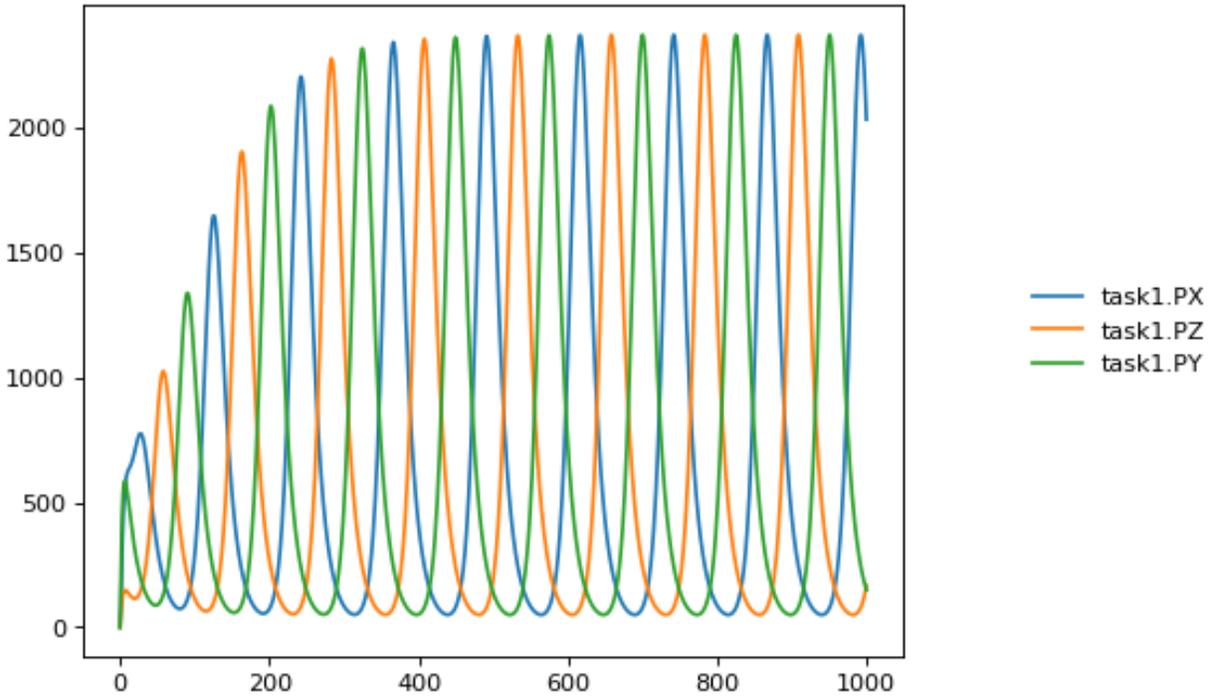
    # Applying postprocessing
    plot "Figure 1.3 Timecourse after post-processing" task1.PX/max(task1.PX) vs_
↳task1.PZ/max(task1.PZ), \
                                                                    task1.PY/max(task1.PY) vs_
↳task1.PX/max(task1.PX), \
                                                                    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())

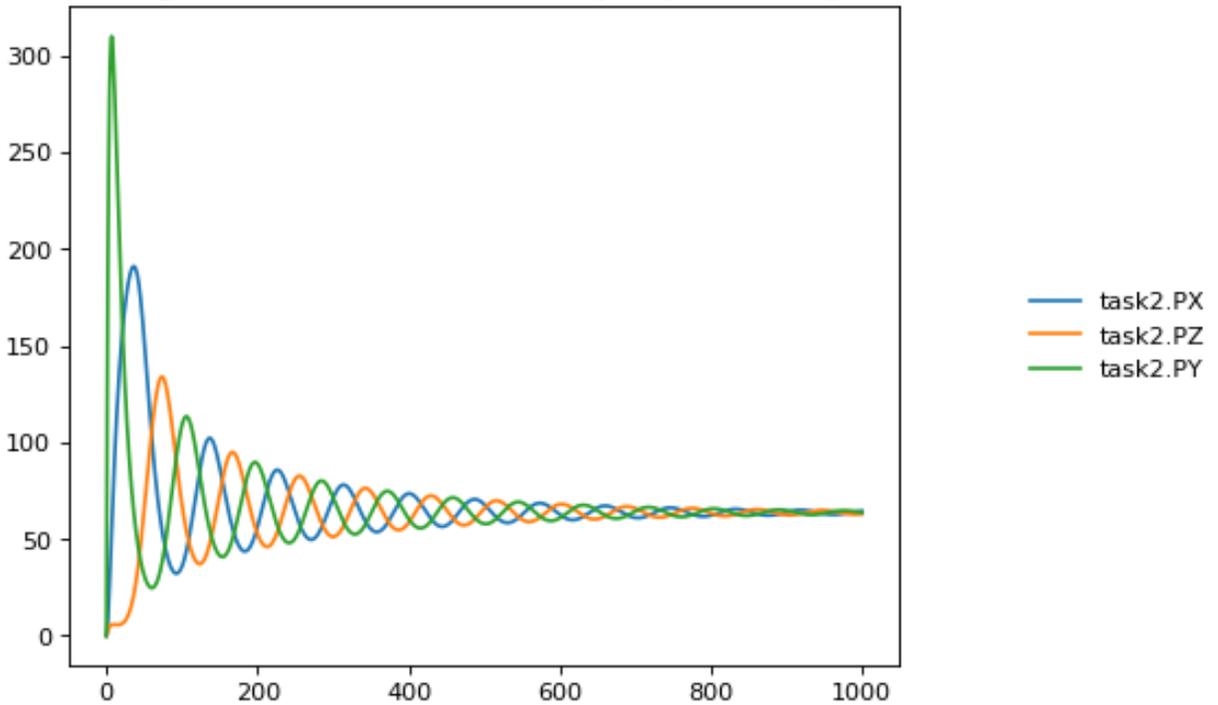
# Run the SED-ML file with results written in workingDir
import tempfile, shutil, os
workingDir = tempfile.mkdtemp(suffix="_sedml")
# write out SBML
with open(os.path.join(workingDir, 'BIOMD0000000012'), 'wb') as f:
    f.write(sedml_str.encode('utf-8'))
te.executeSEDML(sedml_str, workingDir=workingDir)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

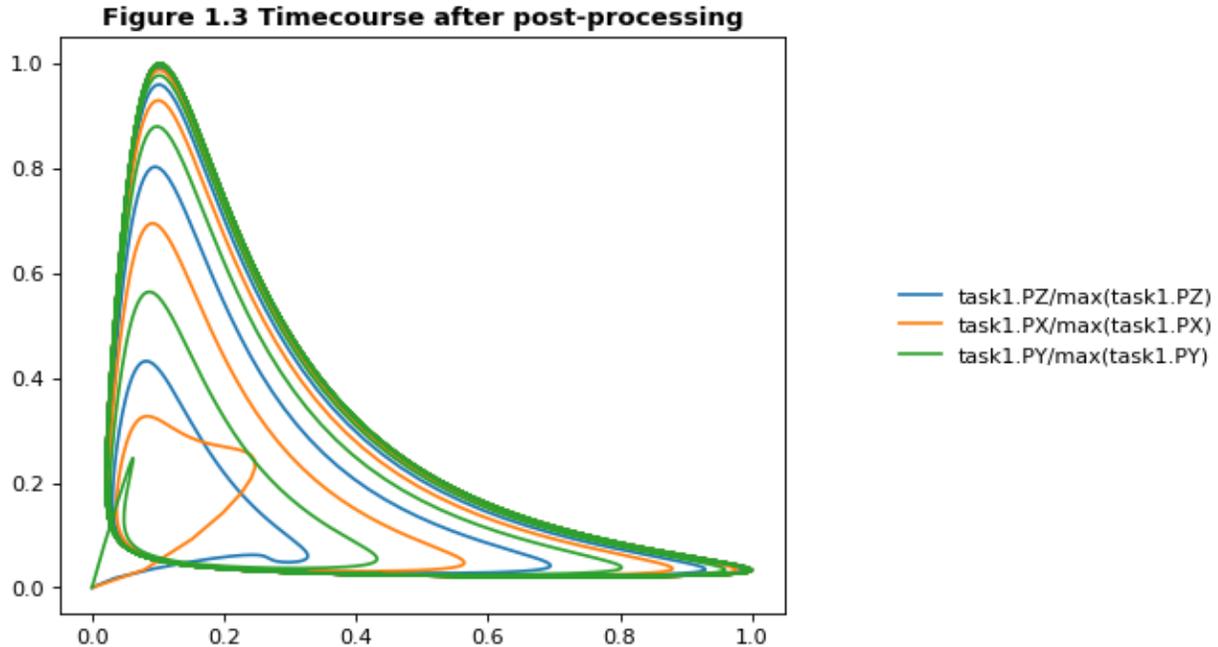
```

**Figure 1.1 Timecourse of repressilator**



**Figure 1.2 Timecourse after pre-processing**





#### 4.3.4 Execute SED-ML Archive

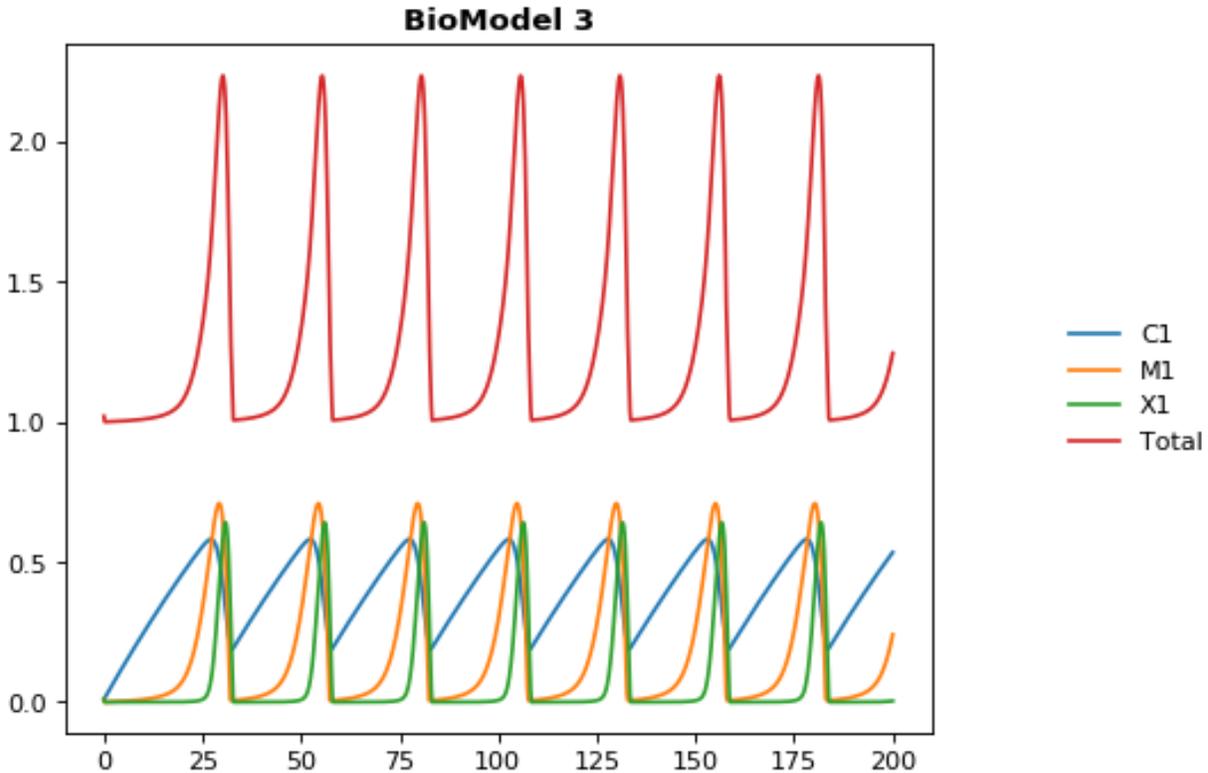
Tellurium can read and execute the SED-ML from a SED-ML archive. This is **not** the same as a COMBINE archive (see below for COMBINE archive examples).

```
import tellurium as te
from tellurium.tests.testdata import sedxDir
import os
omexPath = os.path.join(sedxDir, "BIOMD0000000003.sedx")
print('Loading SED-ML archive from path: {}'.format(omexPath))
print('Using {} as a working directory'.format(os.path.join(os.path.
↳split(omexPath)[0], '_te_BIOMD0000000003')))

# execute the SED-ML archive
te.executeSEDML(omexPath)
```

```
Loading SED-ML archive from path: /home/poltergeist/devel/src/tellurium/tellurium/
↳tests/testdata/sedml/sedx/BIOMD0000000003.sedx
Using /home/poltergeist/devel/src/tellurium/tellurium/tests/testdata/sedml/sedx/_te_
↳BIOMD0000000003 as a working directory
```

```
/home/poltergeist/devel/src/tellurium/tellurium/tecombine.py:330: UserWarning:
No 'manifest.xml' in archive, trying to resolve manually
```



## 4.4 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)
```

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.

### 4.4.1 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.

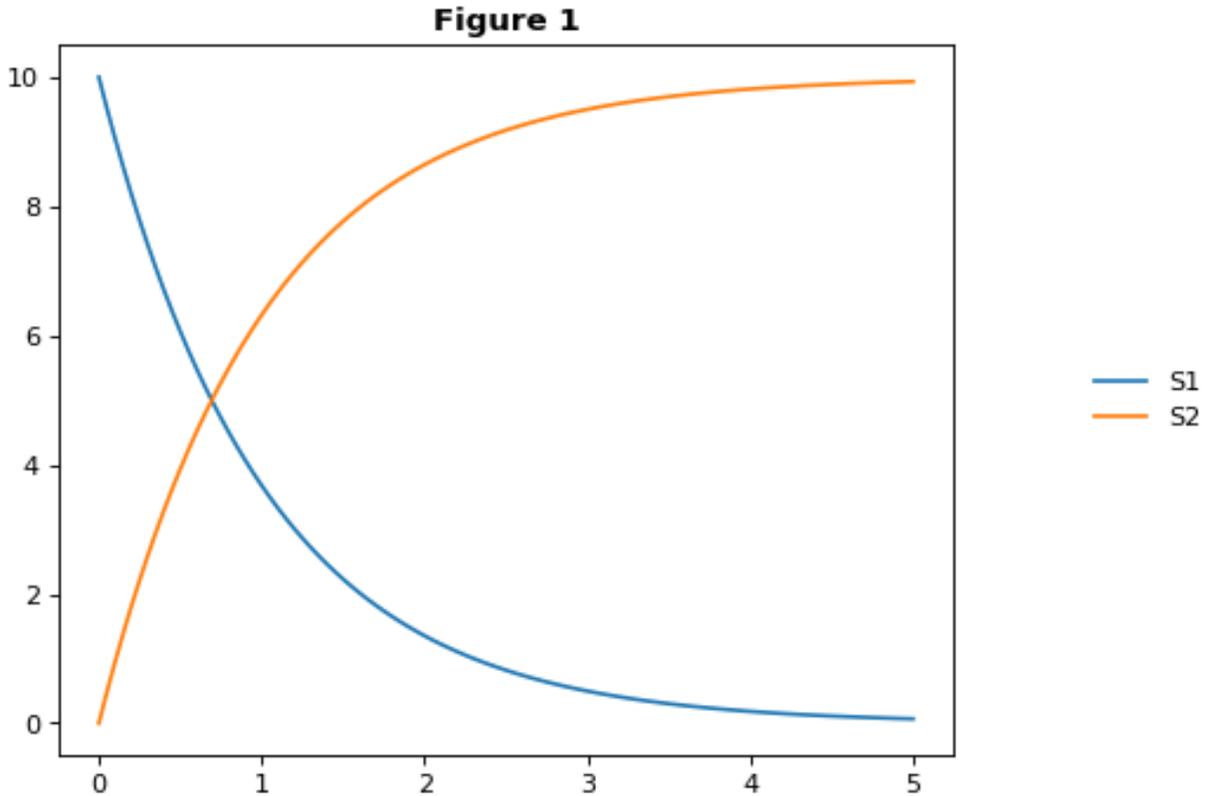
```
import tellurium as te, tempfile, os
te.setDefaultPlottingEngine('matplotlib')

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'))
```



#### 4.4.2 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;

// 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;

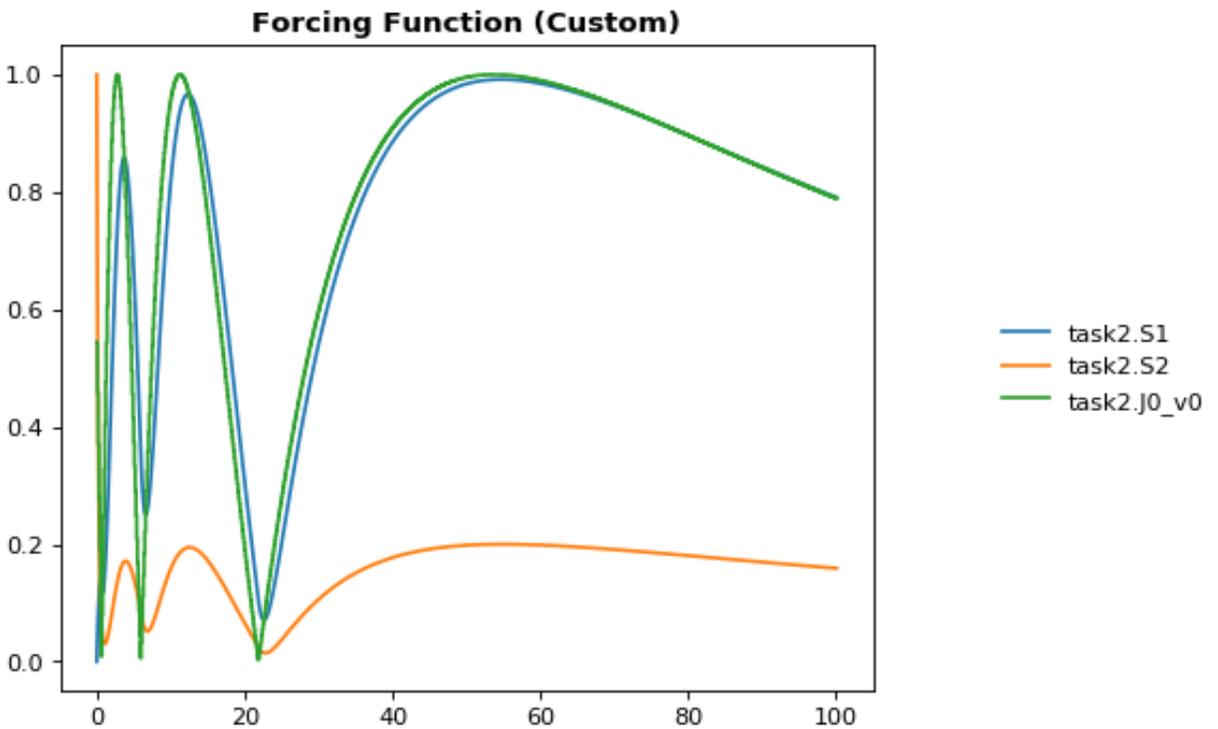
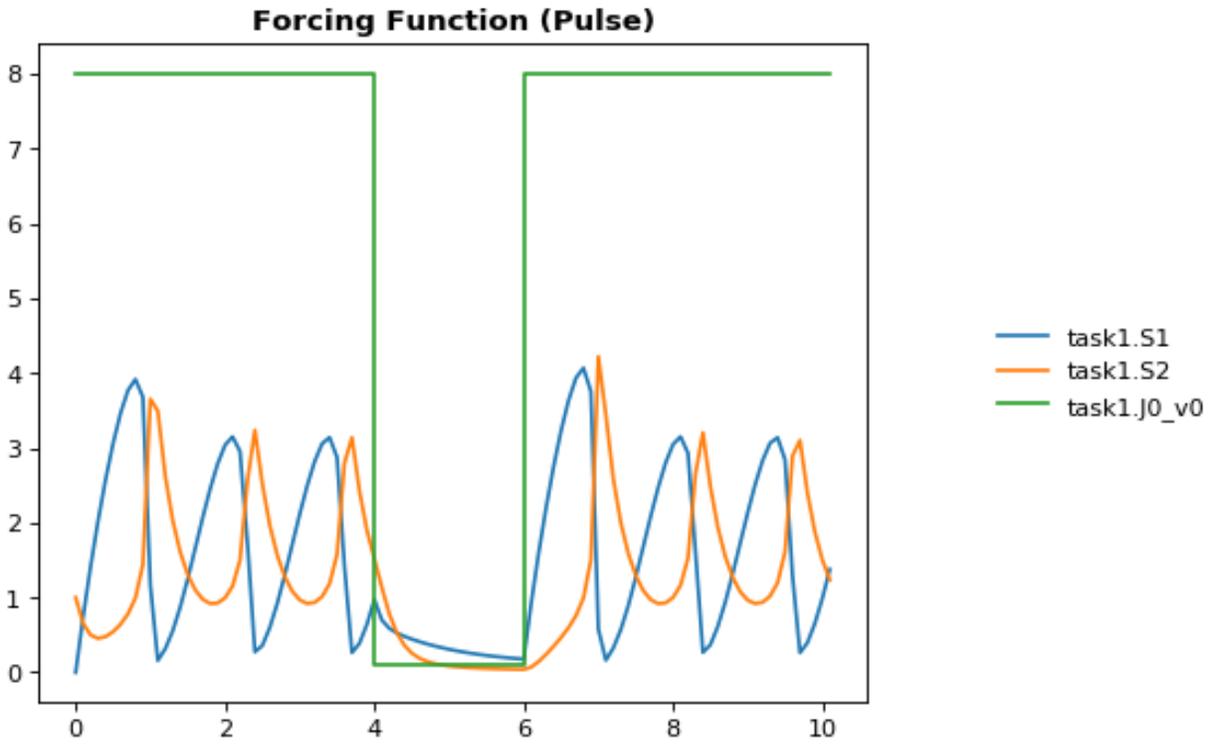
// Other declarations:
const compartment_, J0_v0, J1_k3, J2_k1, J2_k_1, J2_c, J2_q, J3_k2;
end
'''

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)

```



### 4.4.3 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_k_1*S2)*(1 + J2_c*S2^J2_q);
J3: S2 => $X2; J3_k2*S2;

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

// Compartment initializations:
compartment_ = 1;

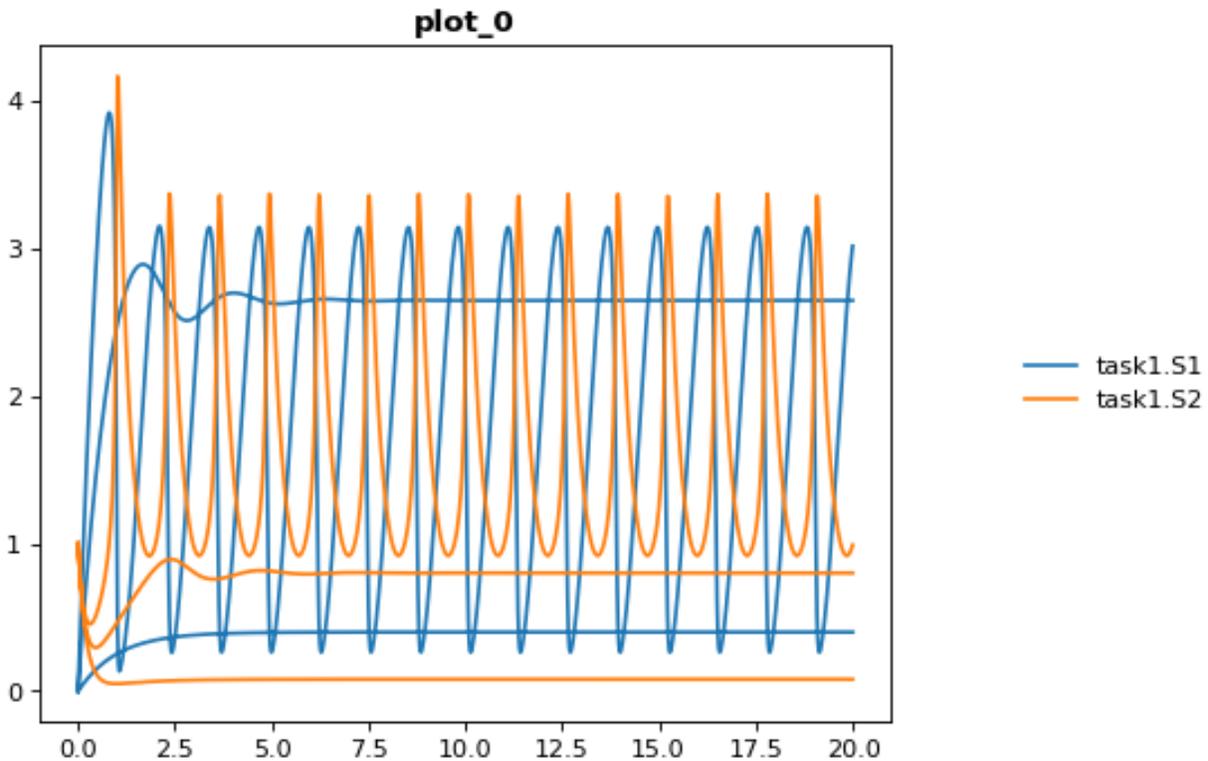
// Variable initializations:
J0_v0 = 8;
J1_k3 = 0;
J2_k1 = 1;
J2_k_1 = 0;
J2_c = 1;
J2_q = 3;
J3_k2 = 5;

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

phrasedml_str = '''
modell = model "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)
```



#### 4.4.4 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);
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

```

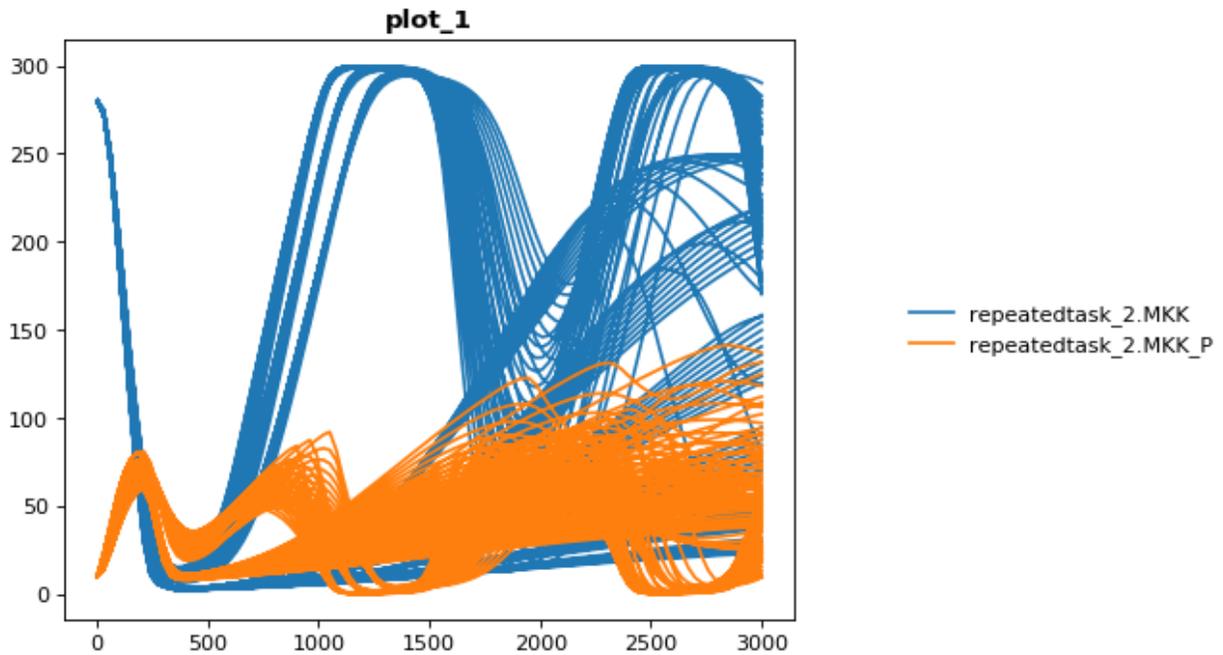
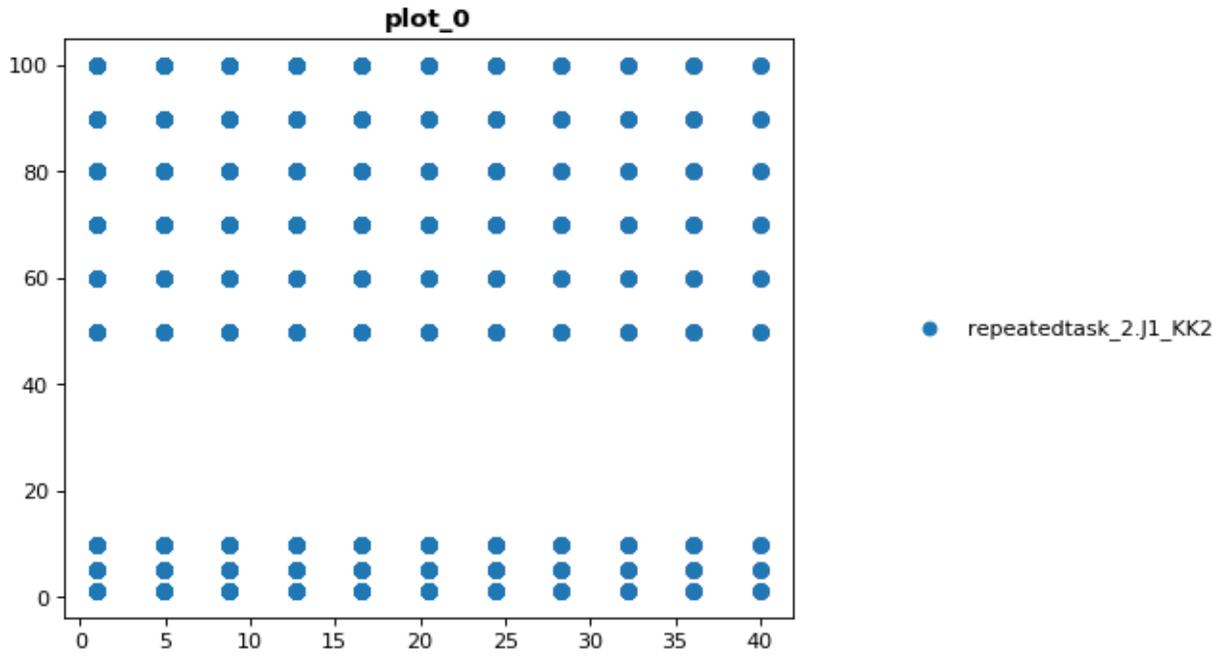
```

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)

```



## 4.4.5 Stochastic Simulation and RNG Seeding

It is possible to programatically 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);
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;
```

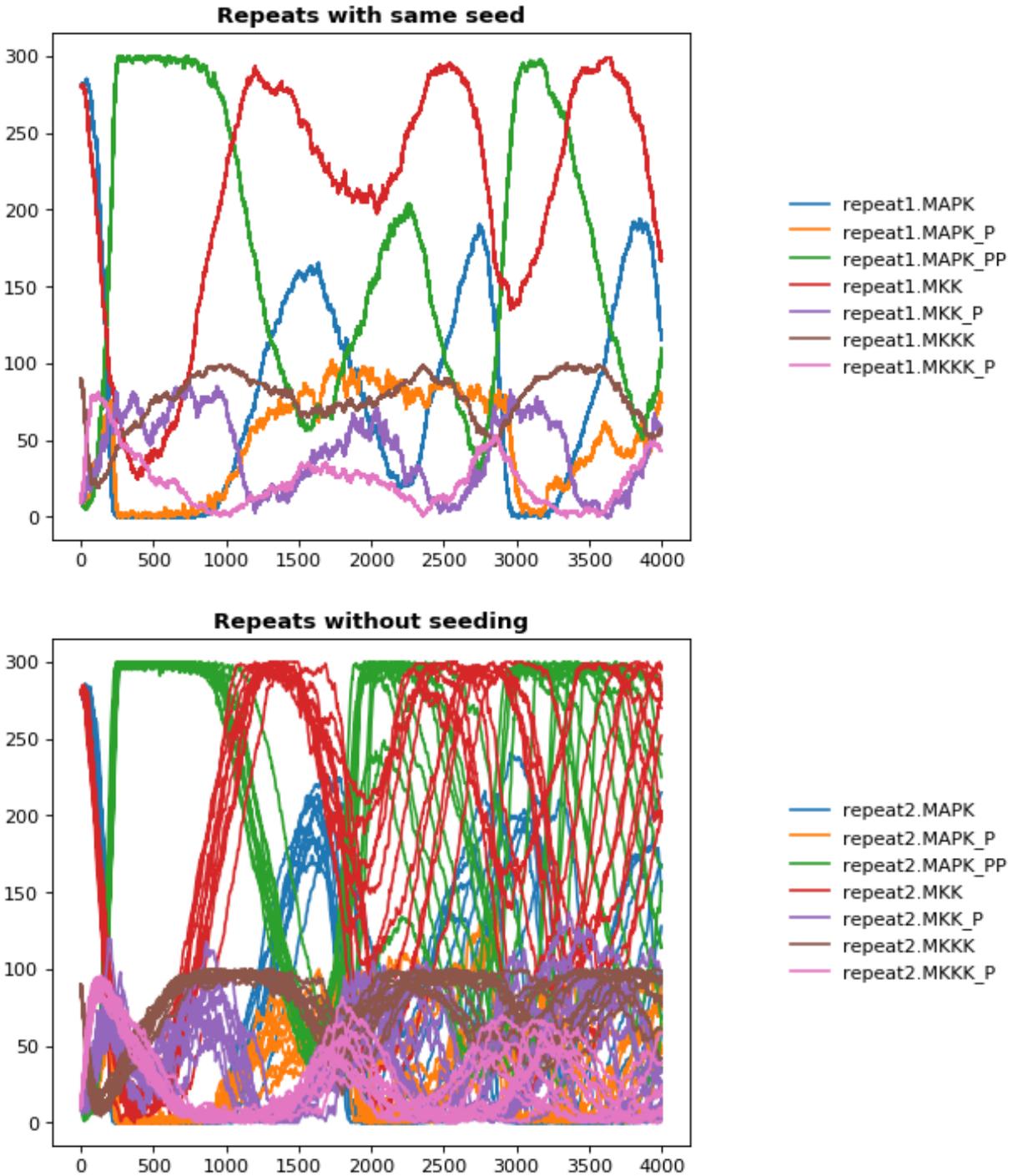
```
J4_V5 = 0.75;
J4_KK5 = 15;
J5_V6 = 0.75;
J5_KK6 = 15;
J6_k7 = 0.025;
J6_KK7 = 15;
J7_k8 = 0.025;
J7_KK8 = 15;
J8_V9 = 0.5;
J8_KK9 = 15;
J9_V10 = 0.5;
J9_KK10 = 15;

// 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 = '''
modell = model "repeatedStochastic"
timecourse1 = simulate uniform_stochastic(0, 4000, 1000)
timecourse1.algorithm.seed = 1003
timecourse2 = simulate uniform_stochastic(0, 4000, 1000)
task1 = run timecourse1 on modell
task2 = run timecourse2 on modell
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)
```



#### 4.4.6 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

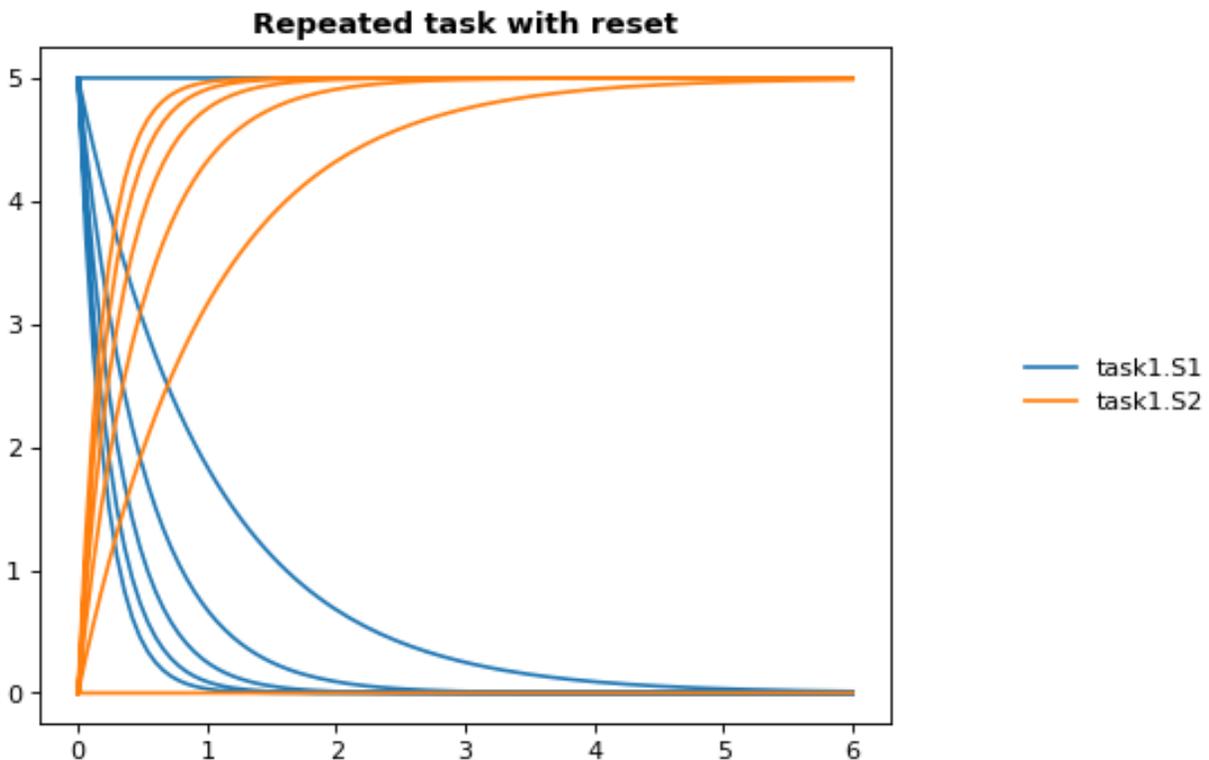
antimony_str = """
model case_02
  J0: S1 -> S2; k1*S1;
  S1 = 10.0; S2=0.0;
  k1 = 0.1;
end
"""

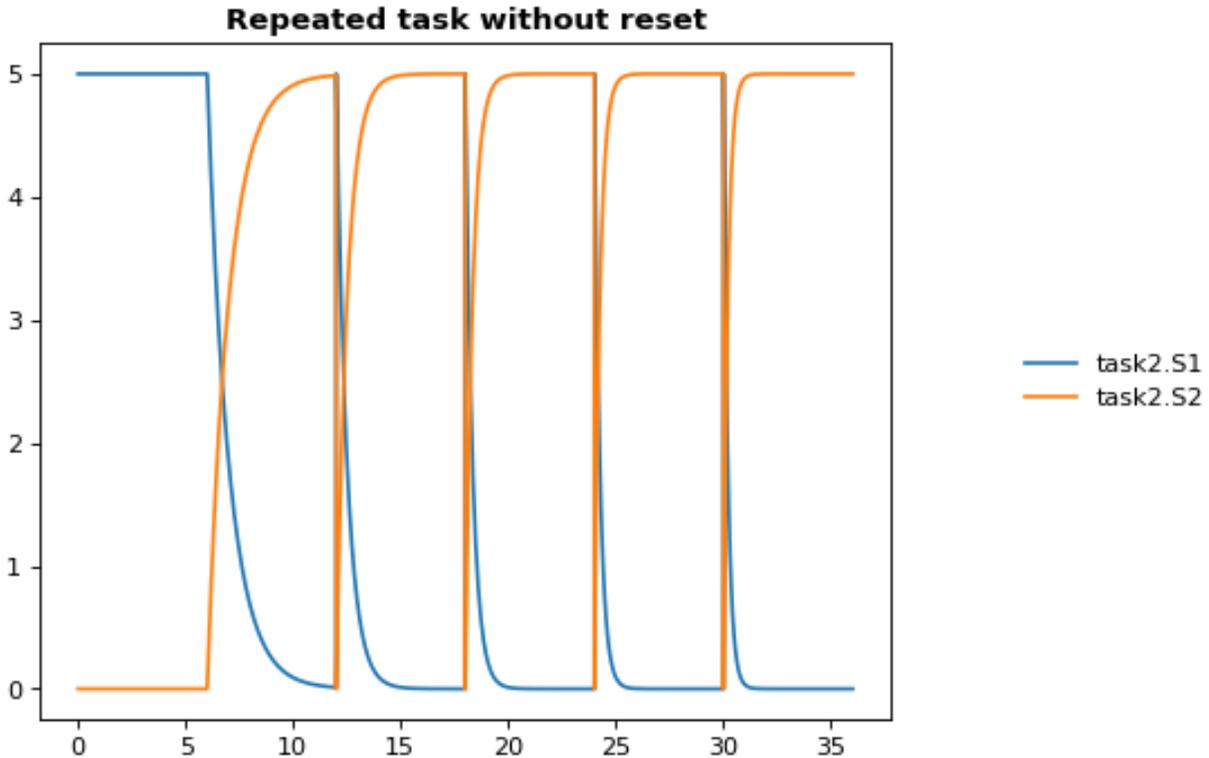
phrasedml_str = """
model0 = model "case_02"
model1 = model model0 with S1=5.0
sim0 = simulate uniform(0, 6, 100)
task0 = run sim0 on model1
# reset the model after each simulation
task1 = repeat task0 for k1 in uniform(0.0, 5.0, 5), reset = true
# show the effect of not resetting for comparison
task2 = 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)

```





#### 4.4.7 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);
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;

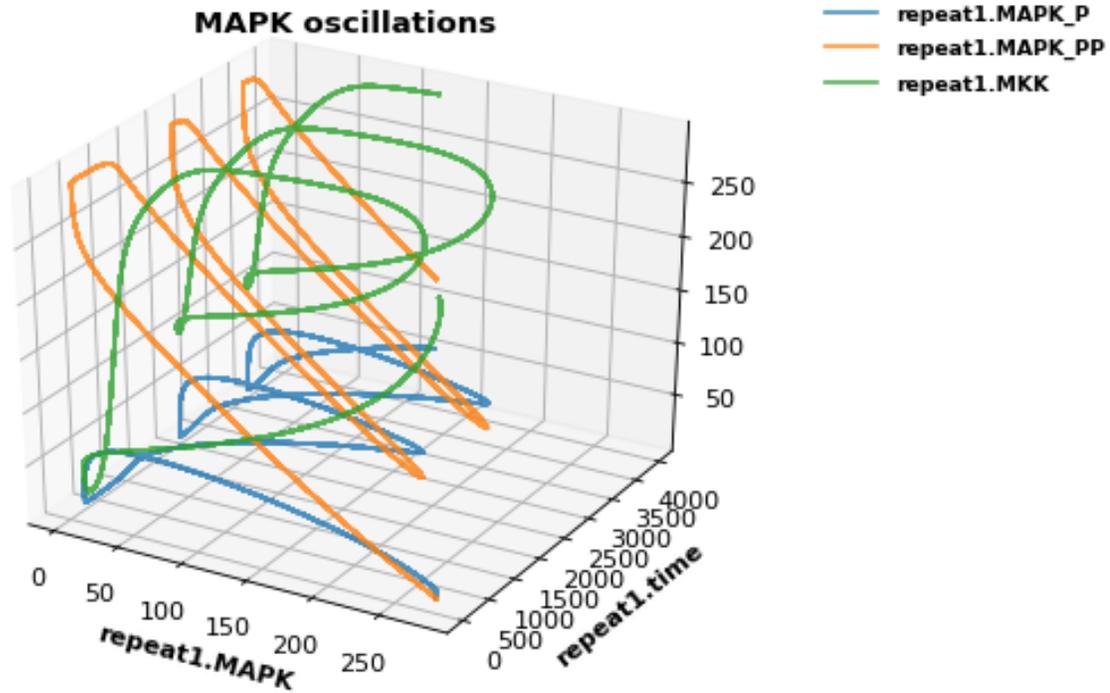
// Other declarations:
const compartment_, J0_V1, J0_Ki, J0_n, J0_K1, J1_V2, J1_KK2, J2_k3, J2_KK3;
const J3_k4, J3_KK4, J4_V5, J4_KK5, J5_V6, J5_KK6, J6_k7, J6_KK7, J7_k8;
const J7_KK8, J8_V9, J8_KK9, J9_V10, J9_KK10;
end
'''

phrasedml_str = '''
  mod1 = model "case_09"
  # sim1 = simulate uniform_stochastic(0, 4000, 1000)
  sim1 = simulate uniform(0, 4000, 1000)
  task1 = run sim1 on mod1
  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
'''

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

```

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



## 4.5 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.

### 4.5.1 Preliminaries

In order to draw the network graphs in these examples, 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)
import sys
print('Please run \n      {} -m pip install pygraphviz\nfrom a terminal or command_
↳propt (without the quotes) to install pygraphviz. Then restart your kernel in this_
↳notebook (Language->Restart Running Kernel)'.format(sys.executable))
```

```
Please run
/home/poltergeist/.config/Tellurium/telocal/python-3.6.1/bin/python3.6 -m pip_
↳install pygraphviz
```

from a terminal or command prompt (without the quotes) to install pygraphviz. Then, restart your kernel in this notebook (Language→Restart Running Kernel).

## 4.5.2 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/"
```

## 4.5.3 Activator system

```
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);

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

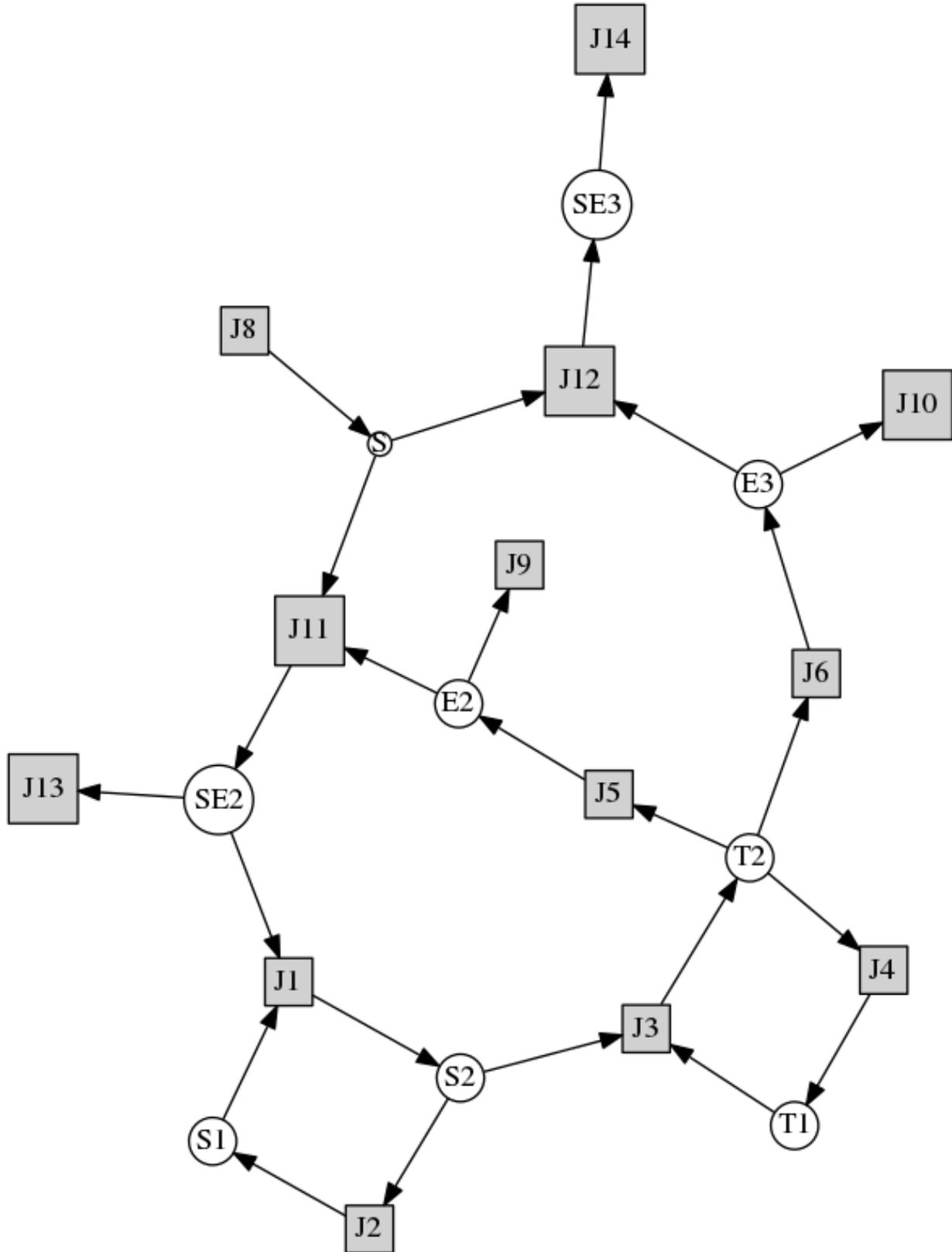
    Km1 = 0.01; Km2 = 0.01; Km3 = 0.01; Km4 = 0.01; Km11 = 1; Km12 = 0.1;
    S1 = 6; S2 =0.1; T1=6; T2 = 0.1;
    SE2 = 0; SE3=0;
    S=0;
    E2 = 0; E3 = 0;
    kcat1 = 0.1; kcat3 = 3; kcat8 =1; kcat11 = 1; kcat12 = 1; kcat13 = 0.1;
    kcat14=0.1;
    E1 = 1;
```

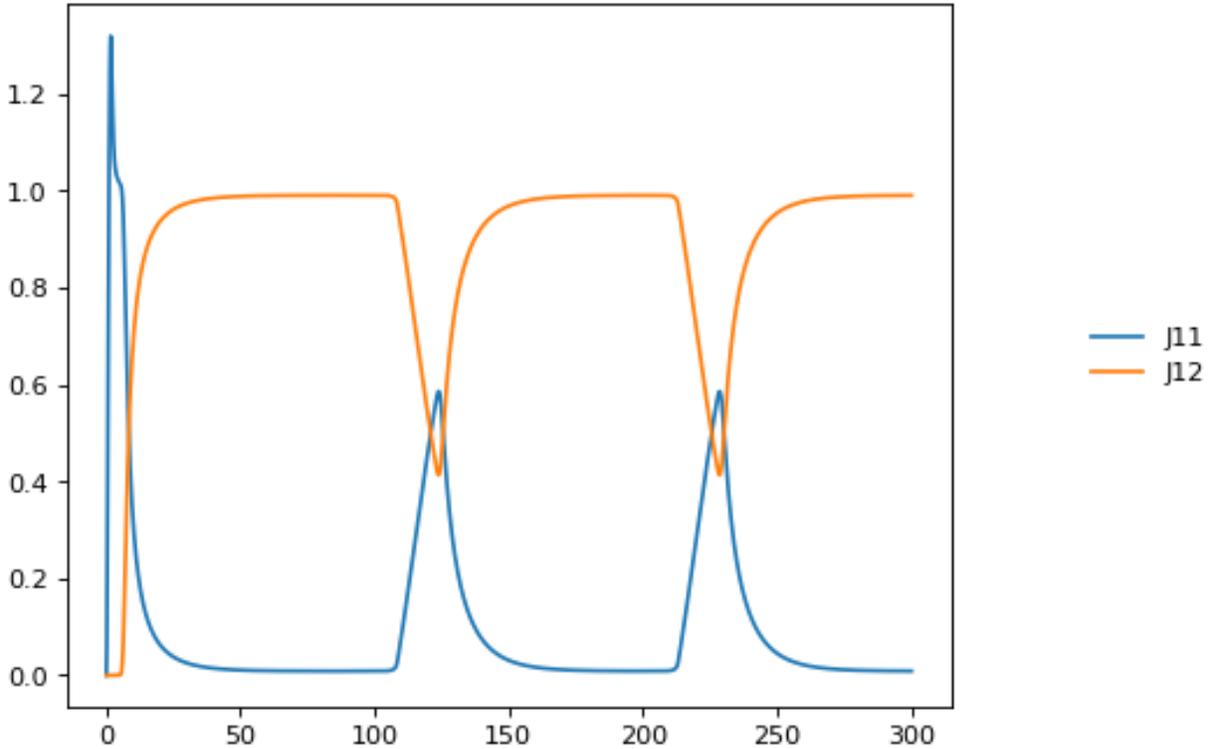
```
k9 = 0.1; k10=0.1;
Vf6 = 1;
Vf5 = 3;
Vm2 = 0.1;
Vm4 = 2;
h6 = 2; h5=2;
Ks6 = 1; Ks5 = 1;
Activator = 0;

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

```
/home/poltergeist/.config/Tellurium/telocal/python-3.6.1/lib/python3.6/site-packages/
↳pygraphviz/agraph.py:1338: RuntimeWarning:

Warning: node 'S', graph '%3' size too small for label
```





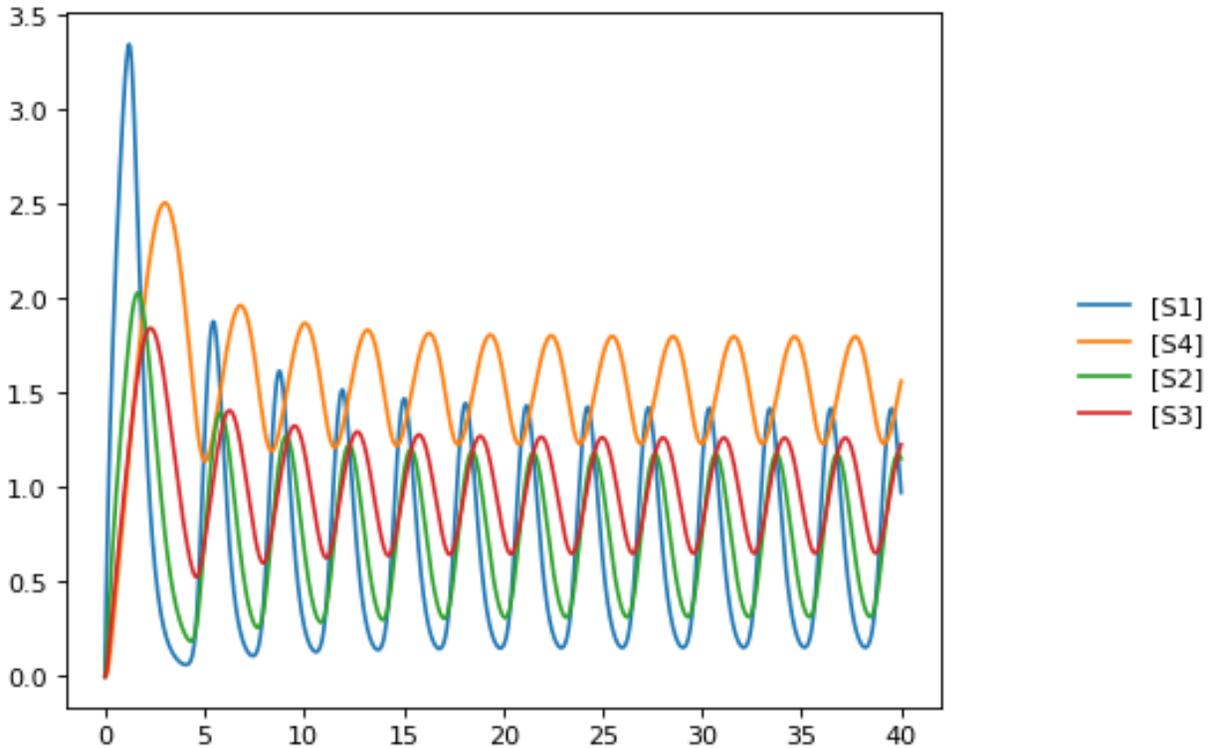
#### 4.5.4 Feedback oscillations

```
# 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()
```



### 4.5.5 Bistable System

Example showing how to do 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;

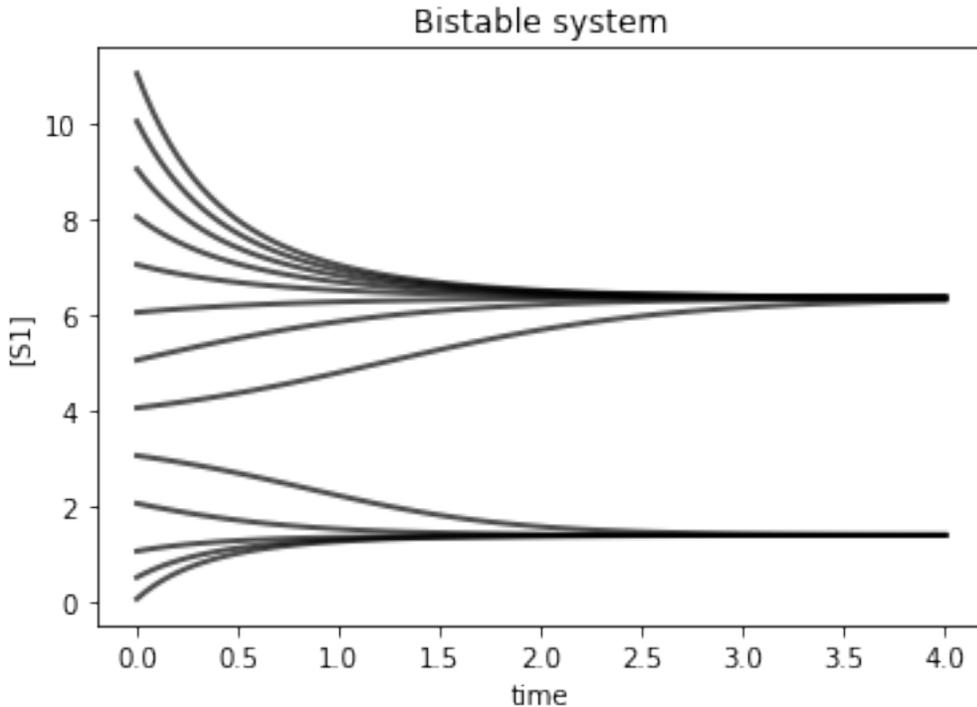
Xo = 0.09; X1 = 0.0;
S1 = 0.5; k1 = 3.2;
''')
print(r.selections)

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]']
```



#### 4.5.6 Add plot elements

```
import tellurium as te
import numpy
import roadrunner

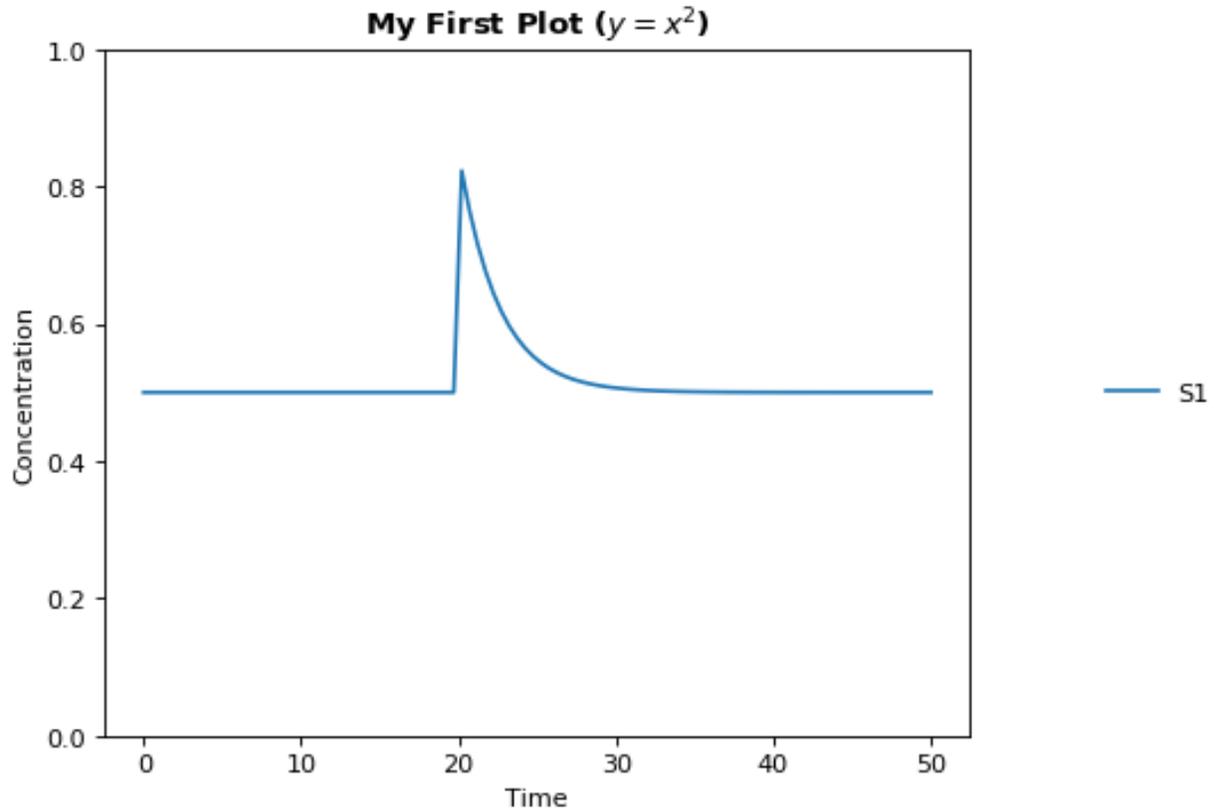
# Example showing how to embellish a graph, change title, axes labels.
# Example also uses an event to pulse S1

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

    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"])

r.plot(m, ylim=(0.,1.), xtitle='Time', ytitle='Concentration', title='My First Plot (
    ↪$y = x^2$)')
```



## 4.5.7 Events

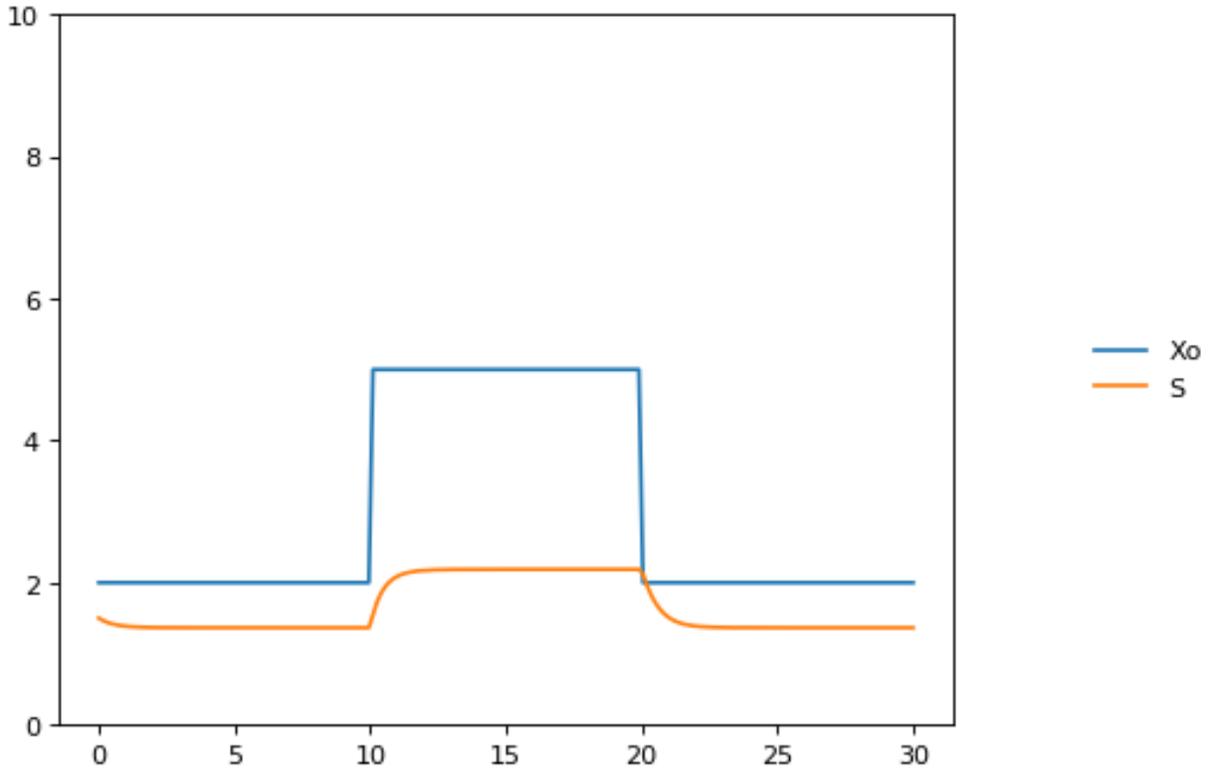
```
import tellurium as te
import matplotlib.pyplot as plt

# 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))
```



#### 4.5.8 Gene network

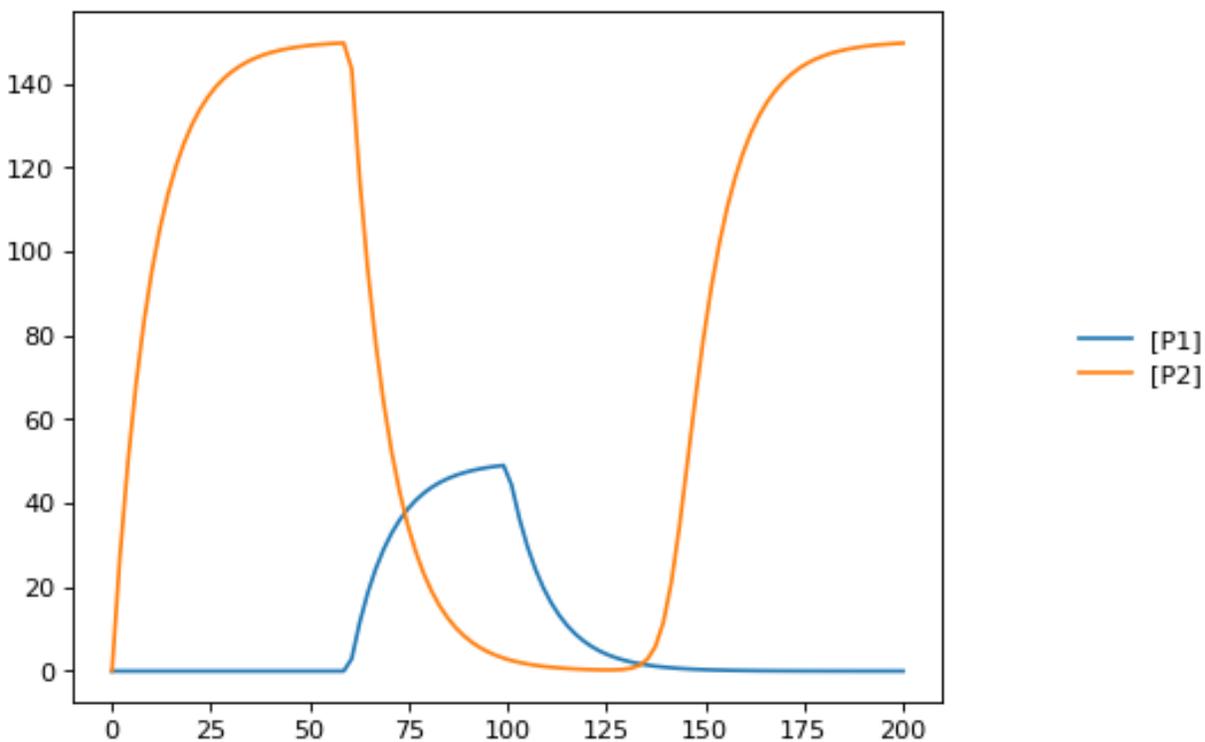
```
import tellurium as te
import numpy

# 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()
```



#### 4.5.9 Stoichiometric matrix

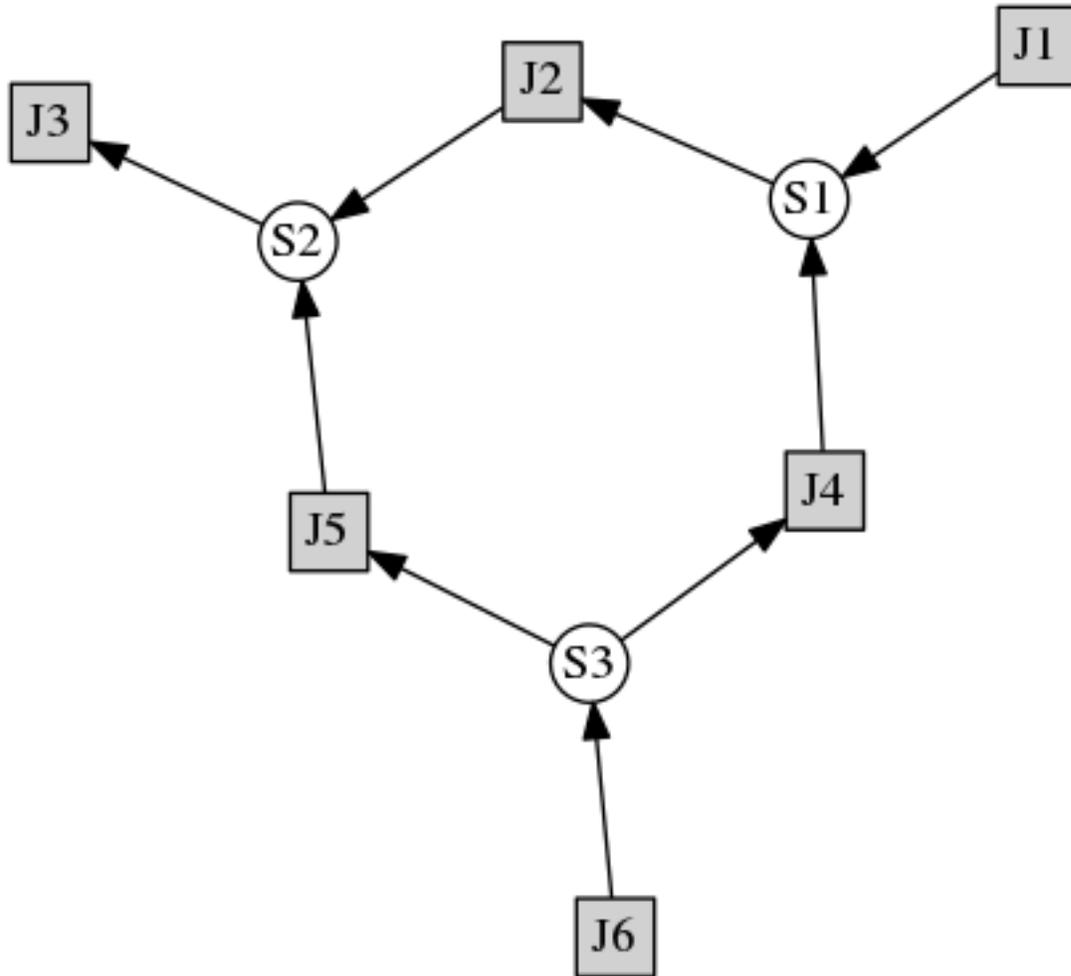
```
import tellurium as te

# 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]]
```



#### 4.5.10 Lorenz attractor

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

```

import tellurium as te

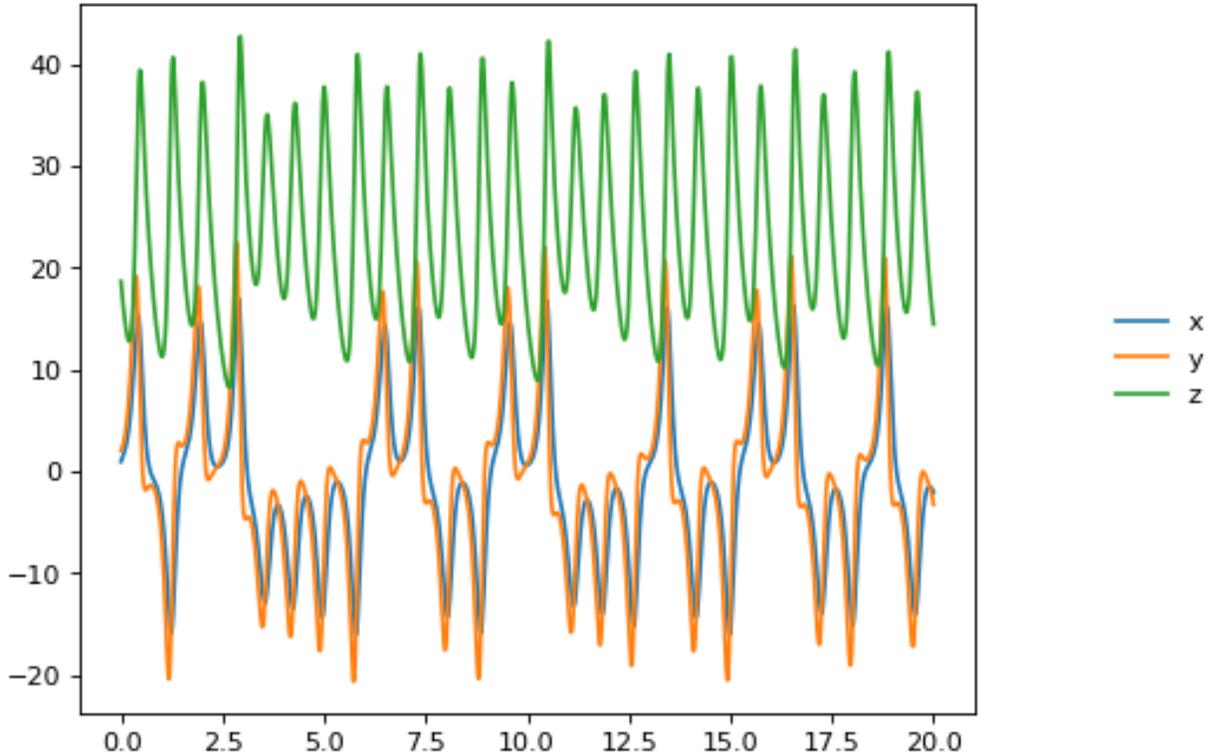
r = te.loada (''
  x' = sigma*(y - x);
  y' = x*(rho - z) - y;
  z' = x*y - beta*z;

  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()

```



#### 4.5.11 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.

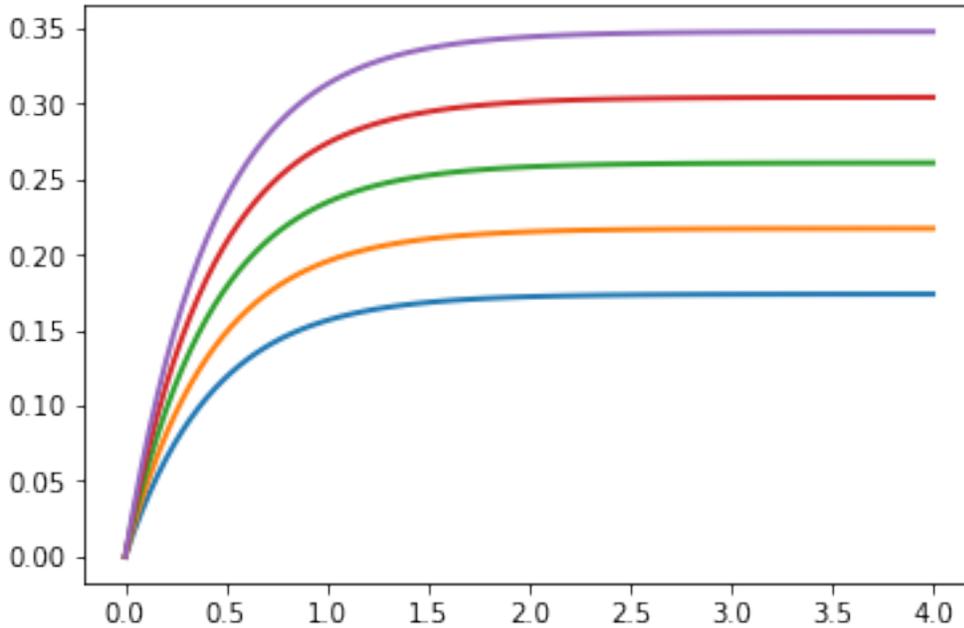
```
import tellurium as te
import numpy as np

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
```



#### 4.5.12 Merge multiple simulations

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

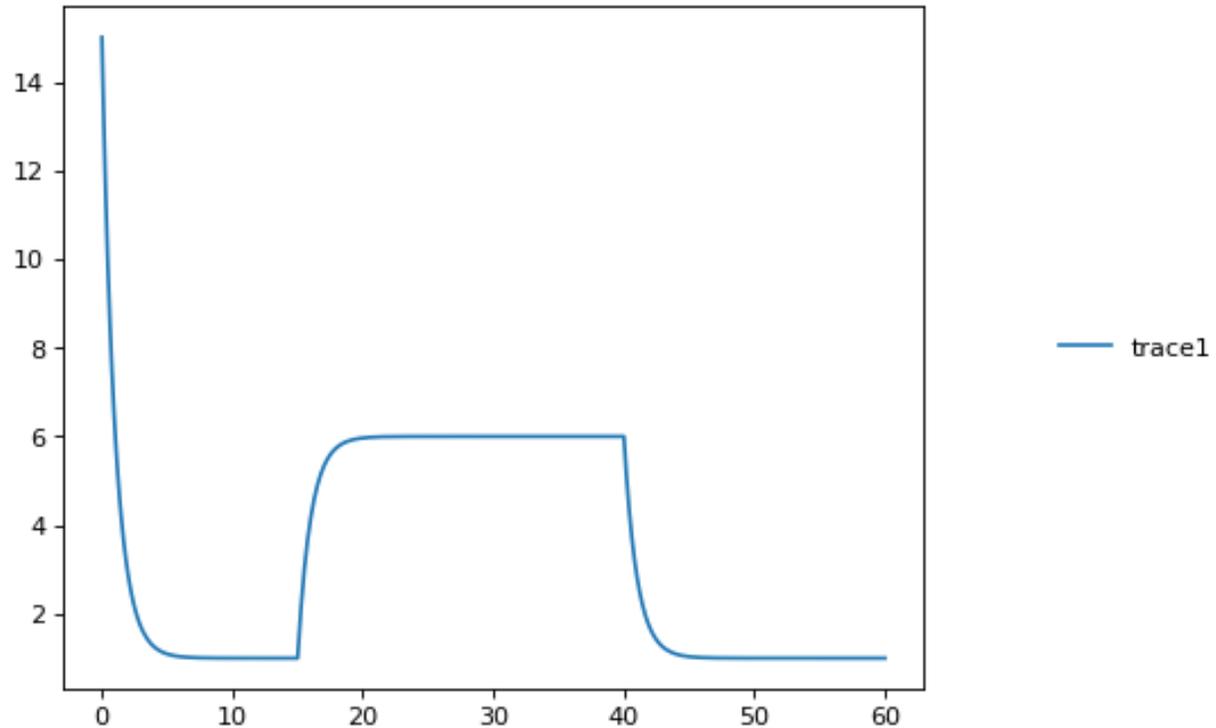
```
import tellurium as te
import numpy

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')
```



### 4.5.13 Relaxation oscillator

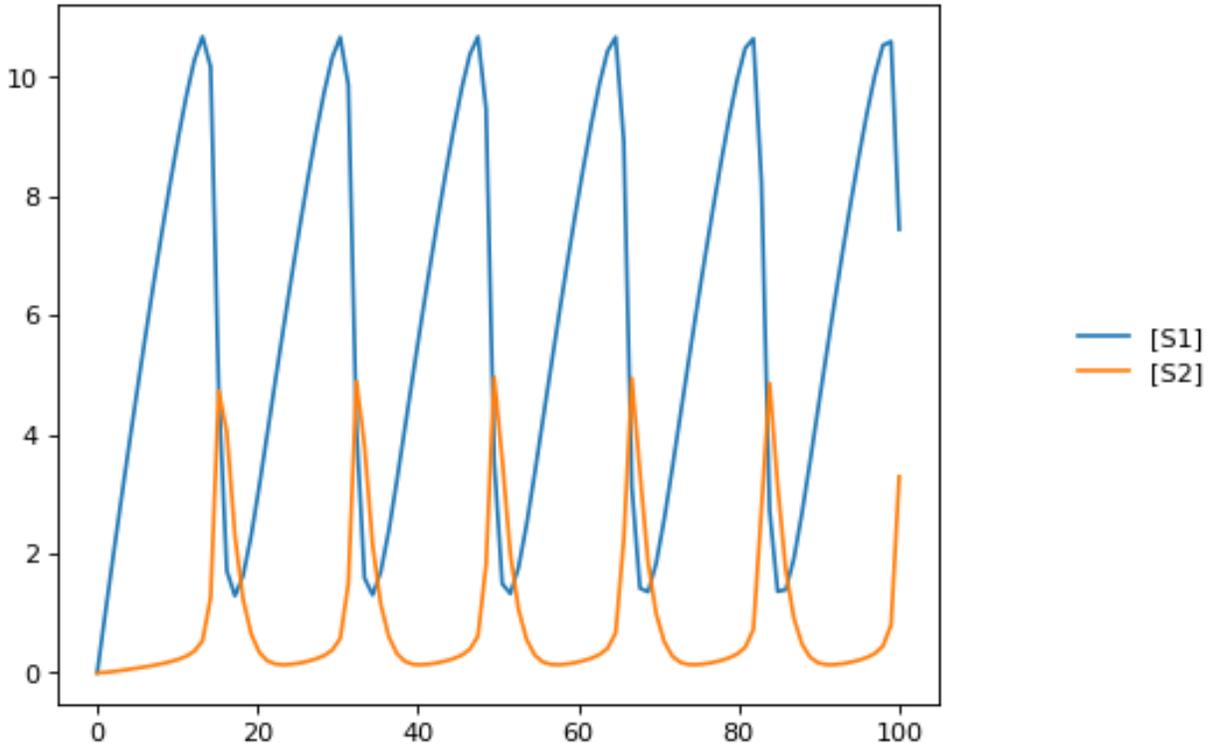
Oscillator that uses positive and negative feedback. An example of a relaxation oscillator.

```
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);
```



#### 4.5.14 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;
```

```

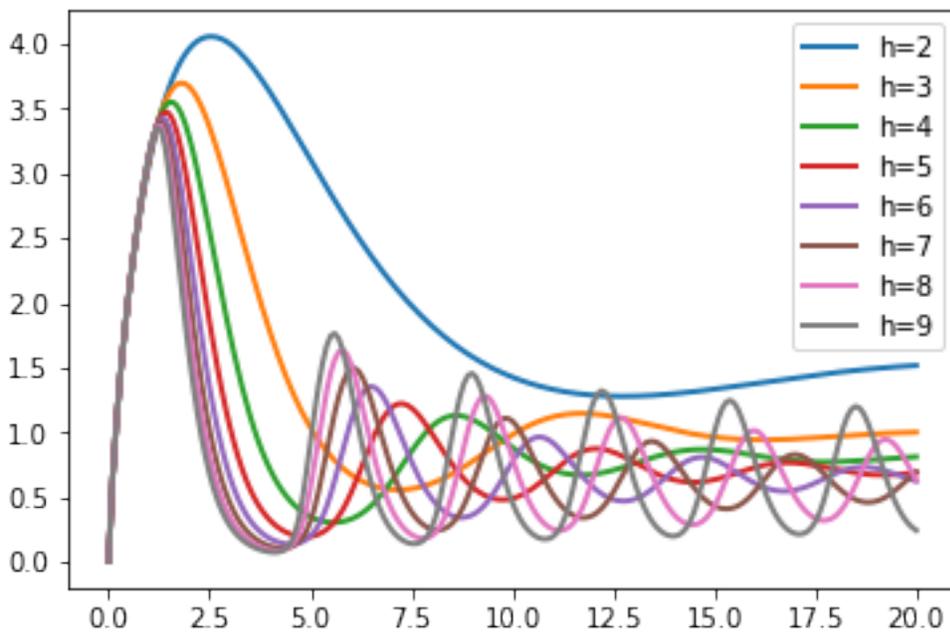
// Other declarations:
const J0_VM1, J0_Keq1, J0_h, J4_V4, J4_KS4;
'''

# 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

```



#### 4.5.15 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);

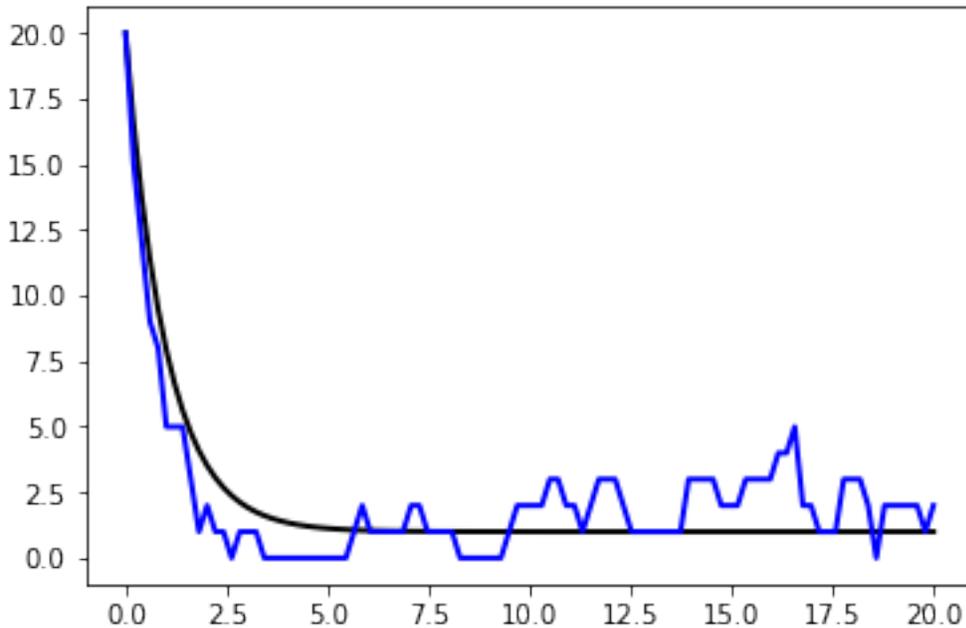
```

```

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

```



#### 4.5.16 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;

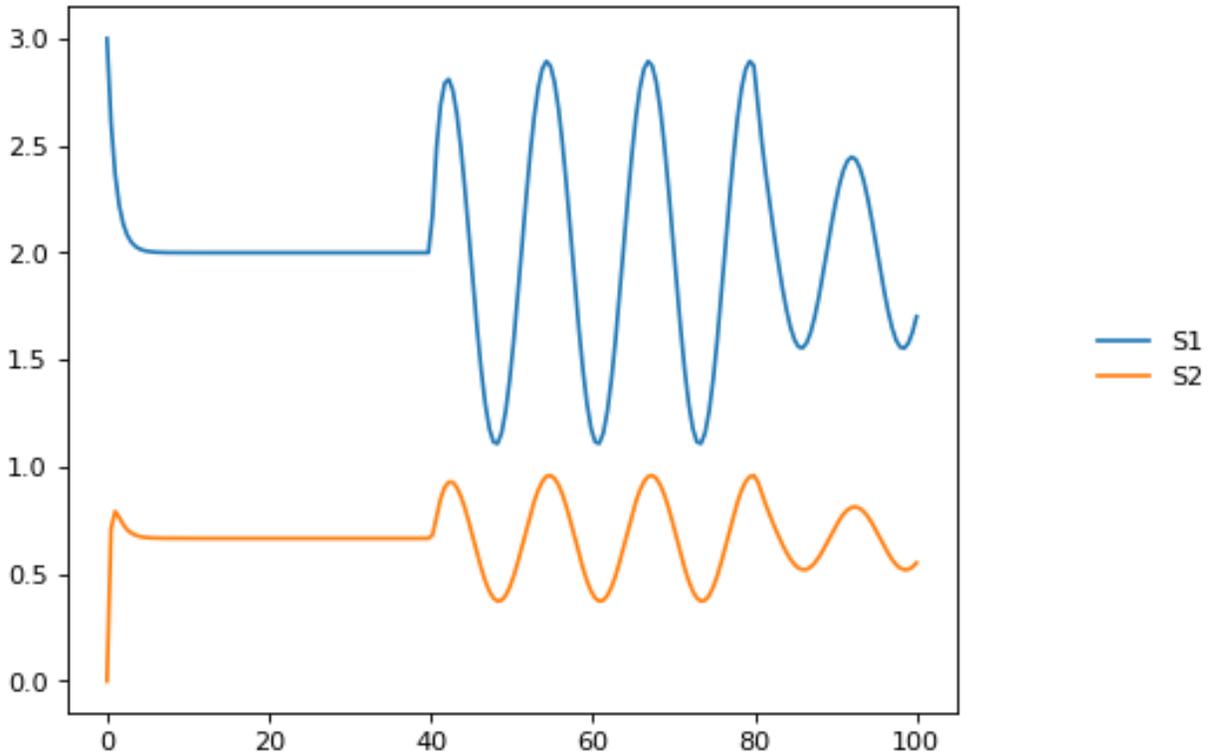
# Model Definition
v1: $Xo -> S1; k1*Xo;
v2: S1 -> S2; k2*S1;
v3: S2 -> $X1; k3*S2;

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;
''')

```

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



#### 4.5.17 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);

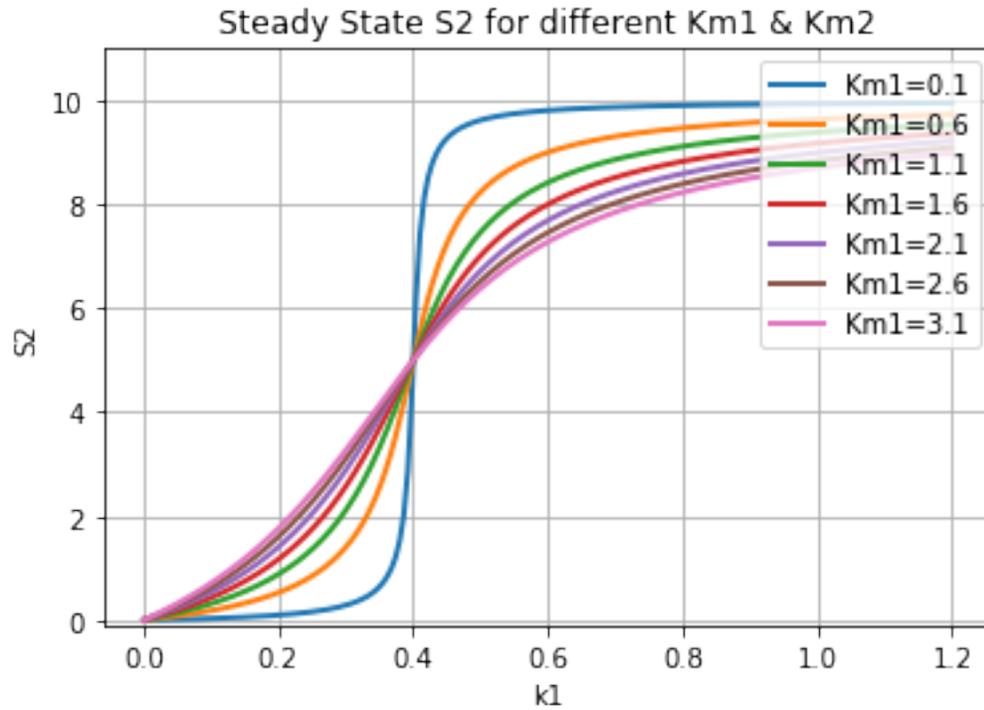
  k1 = 0.1; k2 = 0.4; S1 = 10; S2 = 0;
  Km1 = 0.1; Km2 = 0.1;
''')

r.conservedMoietyAnalysis = True

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",
             ylim=[-0.1, 11], grid=True)

r.k1 = 0.1
r.Km1 = r.Km1 + 0.5;
r.Km2 = r.Km2 + 0.5;
```





---

## 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*
    - \* *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*
- \* *Comments*
- \* *Units*
- \* *DNA Strands*
- \* *Interactions*
- \* *Function Definitions*
- \* *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 oscli
  #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
```

### 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
```

```
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
  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 with `X'`, 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 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 to 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](#).

## 5.5.9 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  $k1*s1^2 + k2*s1 + k3$ .

## 5.5.10 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.

### 5.5.11 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.

### 5.5.12 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'.
```

## 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
''')

# 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"))))

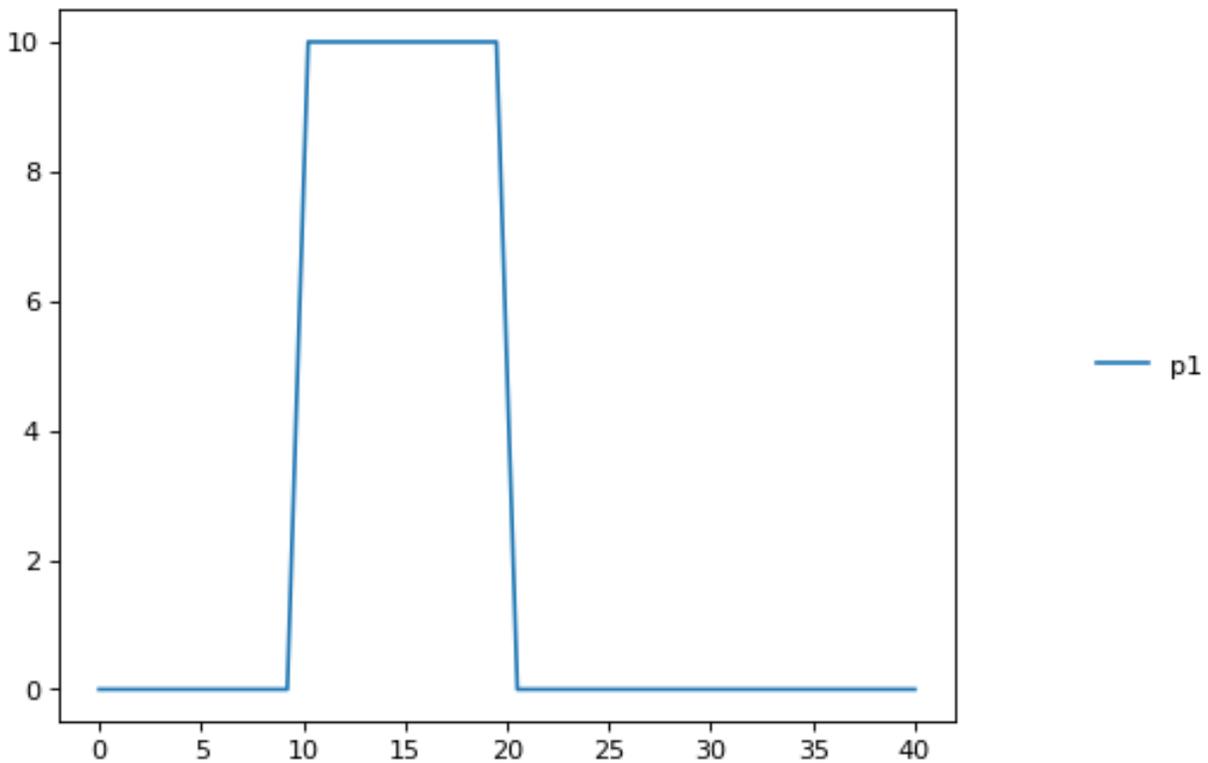
# now simulate up to time 15
r.resetAll()
```

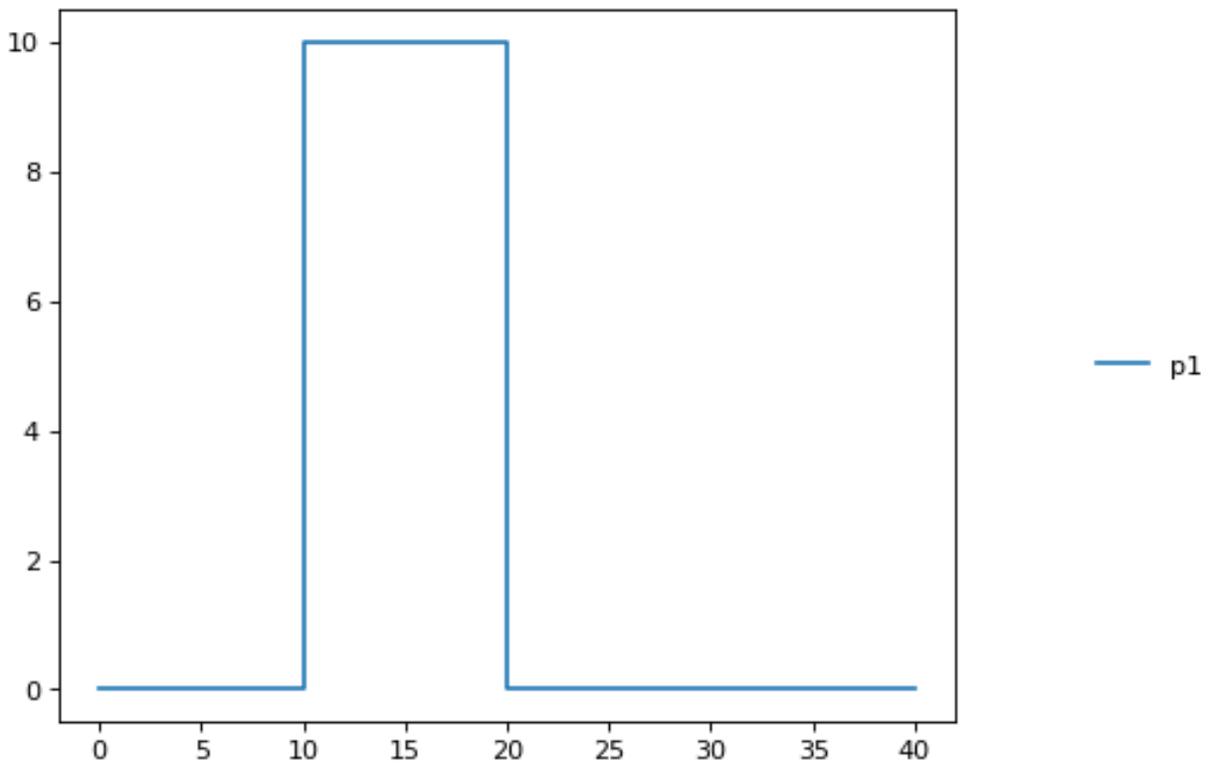
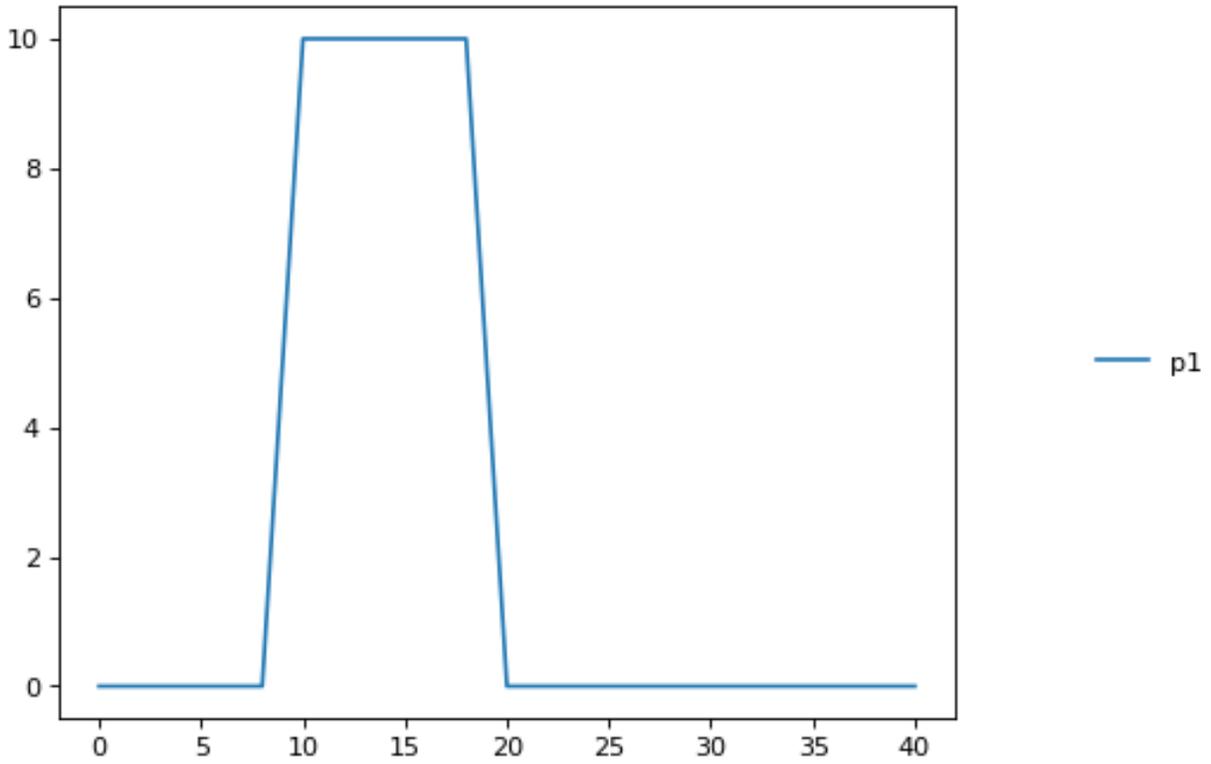
```
s4 = r.simulate(0, 15)
r.plot()

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

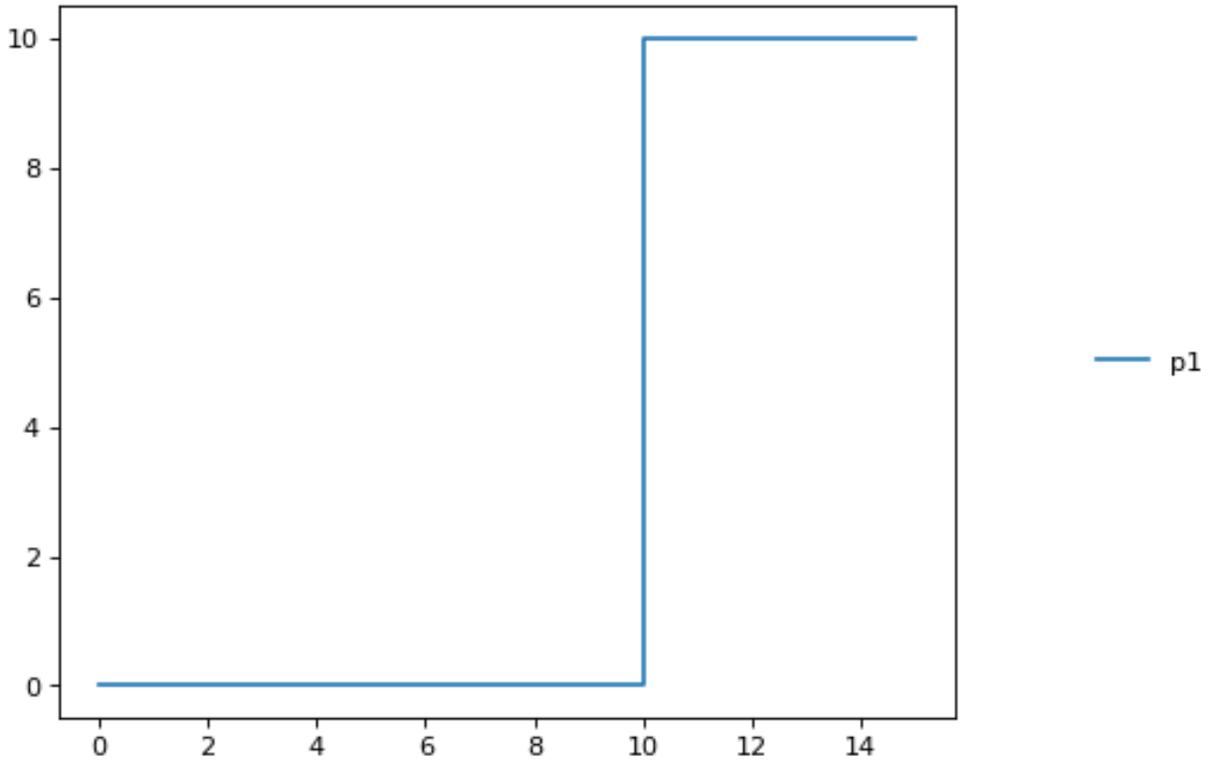
print("Comparing Antimony at time 0 & 15")
print('\n'.join(list(diffliib.unified_diff(ant_str_before.splitlines(), ant_str_after2.
↪splitlines(), fromfile="before.sb", tofile="after.sb"))))
```

```
-----
tellurium : 2.0.1
roadrunner : 1.4.21; Compiler: gcc 4.8.2, C++ version: 199711; JIT Compiler: LLVM-3.3;
↪ Date: Jul  5 2017, 18:38:02; LibSBML Version: 5.14.0
antimony : 2.9.3
libsbml : 5.15.0
libsedml : 402
phrasedml : 1.0.7
-----
```





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;
```

```
  // 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, `k1` and `k2` 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 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;
  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, t0, 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 alpha 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 subtract the ramp slope that results in a horizontal line.

```
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; 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;

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)
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 Comments

Comments in Antimony can be made on one line with `//[comments]`, or on multiple lines with `/* [comments] */`:

```
/* The following initializations were
   taken from the literature */
X=3; //Taken from Galdziki, et al.
Y=4; //Taken from Rutherford, et al.
```

Comments are not translated to SBML or CellML, and will be lost if round-tripped through those languages.

### 5.7.13 Units

As of version 2.4 of Antimony, units may now be created and translated to SBML (but not CellML, yet). Units may be created 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’.

### 5.7.14 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.

### 5.7.15 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 “->”: for activations, use “-o”; for inhibitions, use “-|”, and for unknown interactions or for interactions which sometimes activate and sometimes inhibit, use “-(“:

```
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.

### 5.7.16 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 thus:

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

And then use it in a later equation:

```
S3 = quadratic(s1, k1, k2, k3);
```

This would effectively define S3 to have the equation  $k1*s1^2 + k2*s1 + k3$ .

### 5.7.17 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.

### 5.7.18 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 model1.txt model2.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 model1.txt -o sbml-comp
```

will output `model1.xml` in the working directory, containing all models in the `model1.txt` file, using the SBML `comp` package.

### 5.7.19 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.

### 5.7.20 Further Reading

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

---

## Parallel Programming

---

### 6.1 Tellurium in Distributed Environment

Tellurium features a new functionality that can help users to perform simulation much faster. In the background it runs on a distributed cluster of machines and computations are done in-memory to achieve a very low latency. Currently, it allows users to run the computations/simulations with regards to parameter scans, parameter fitting, sensitivity analysis and parameter identifiability. [Apache Spark](#) has been integrated with Tellurium and this allow users to run different simulations parallely.

To run this, first it is required to setup Spark Cluster on one ore more machines which follows Master-Slave architecture. Spark can run on Spark's Standalone, [Apache Mesos](#) and [Hadoop YARN](#). If you are a python lover, you are in garden. Run the below pip command to install spark in a single machine.

```
pip install pyspark
```

Or follow the instruction available in [Spark Downloads Page](#).

### 6.2 Getting Started

First it is required to initialize spark context (sc) as this coordinates everything that needs to run parallely. This is available in default if you open pyspark from terminal or through [Zeppelin](#).

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName('<APPLICATION_NAME>').setMaster('<SPARK_MASTER>')
sc = SparkContext(conf=conf)
```

The two important things that needs to get replaced are <APPLICATION\_NAME> which can be anything that a program can be named while the other thing is <SPARK\_MASTER> which can be local[2] (Which means your application uses 2 threads and it can be any 'n' value) if you are running it on a single machine or it could be of the form spark://127.208.10.259:8000 assuming your spark master is on 8000 port and running on node with IP 127.208.10.259.

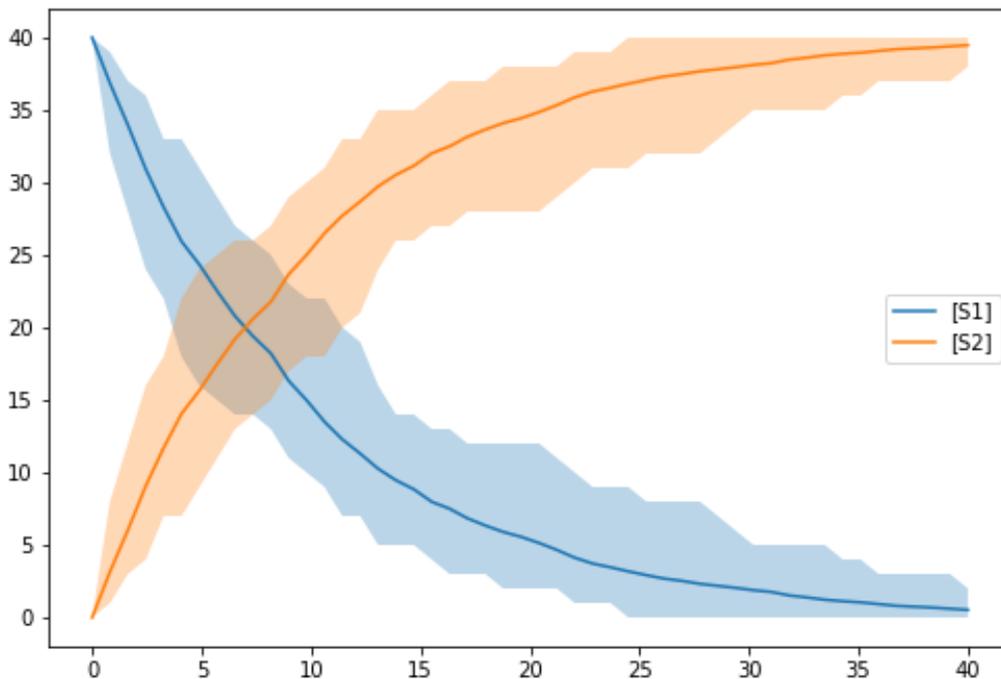
## 6.3 Your First Stochastic Simulation

After the Spark Context is perfectly initialised, lets run our first simple stochastic simulation.

```
import tellurium as te

model_def = 'S1 -> S2; k1*S1; k1 = 0.1; S1 = 40'
stochastic_simulation_model = te.StochasticSimulationModel(model=model_def,
    seed=1234, # not used
    variable_step_size = False,
    from_time=0,
    to_time=40,
    step_points=50)
stochastic_simulation_model.integrator = "gillespie"
results = te.distributed_stochastic_simulation(sc, stochastic_simulation_model, 50)
te.plot_stochastic_result(results)
```

In the above example, we have created a model as an antimony string and then created a `StochasticSimulationModel` object with the parameters mentioned above. From the above example your model provided in the form of antimony string will simulate from 0 to 40 with 50 step points. You might see the result that looks like below



## 6.4 Parameter Estimation

As a next example, you want to run Parameter Estimation for a specific set of attributes of your model(s). Additionally you may need to provide an optimization function so as to estimate values accurately. It begins with usual object creation of stochastic simulation model (Check the above code for reference)

Lets test Lotka Volterra Model and try estimating parameters in that model. We will use differential\_evolution as an optimization function available in scipy library.

```
import tellurium as te
from scipy.optimize import differential_evolution as diff_evolve
```

Now we will initialise our Lotka-Volterra antimony model.

```
# Load the Lotka-Volterra model
lotka_volterra_antimony_model_definition = '''
model *LVModel()
  // Compartments and Species:
  compartment compartment_;
  species A in compartment_, B in compartment_;
  // Reactions:
  R1: A => 2A; compartment_*R1_k1*A;
  R2: A => B; compartment_*A*B*R2_p;
  R4: B => ; compartment_*R4_k1*B;
  // Species initializations:
  A = 71;
  B = 79;
  // Compartment initializations:
  compartment_ = 1;
  // Variable initializations:
  R1_k1 = 0.5;
  R2_p = 0.0025;
  R4_k1 = 0.3;
  // Other declarations:
  const compartment_;
  // Unit definitions:
  unit volume = 1e-3 litre;
  unit substance = item;
end
'''
```

We use the above initialisation in order to create a Stochastic Simulation Model

```
stochastic_simulation_model = te.StochasticSimulationModel(model=lotka_volterra_
↳antimony_model_definition,
  seed=1234, # not used
  variable_step_size = False,
  from_time=0,
  to_time=1000,
  step_points=1000)
stochastic_simulation_model.integrator = "gillespie"
```

Defining the bounds of the parameters we wish to estimate

```
bounds = {"R1_k1": (0.0, 1.0), "R4_k1": (0.0, 0.5)}
```

If you wish to run it as a stochastic simulation with running number of simulation in a distributed enviroment.

```
from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName('RunningMonteCluster').setMaster('SPARK_MASTER')
sc = SparkContext(conf=conf)
```

Now, you just need to call run function to evaluate and estimate the parameters based on the bounds provided

```
parameter_est = te.ParameterEstimation(stochastic_simulation_model,bounds,
↳stochastic=False, sc=None)
path = "/home/shaik/year/stoch-param-fitting-benchmarks/zimmer/ID/"
parameter_est.setDataFromFile(path+FILENAME)
print parameter_est.run(diff_evolver,maxiter=1)
```

If we want to test it for Immigration-Death Model, all we need to do is to change the antimony\_model string and change the bounds.

```
immigration_death_antimony_model_definition = '''
model *IDModel()

// Compartments and Species:
compartment compartment_;
species A in compartment_;

// Reactions:
R1: => A; compartment_*p1;
R2: A => ; compartment_*p2*A;

// Species initializations:
A = 10;

// Compartment initializations:
compartment_ = 1;

// Variable initializations:
p1 = 1;
p2 = 0.1;

// Other declarations:
const compartment_, p1, p2;

// Unit definitions:
unit volume = 1e-3 litre;
unit substance = item;
end
'''
bounds = {"p1":(0.0,5.0),"p2":(0.0,0.5)}
```

Lets look into more Complex models using Stochastic Simulations

```
from scipy.optimize import differential_evolution as diff_evolver
stochastic_simulation_model = te.StochasticSimulationModel(model=antimony_model,
    seed=1234, # not used
    variable_step_size = False,
    from_time=0,
    to_time=1000,
    step_points=1000)
stochastic_simulation_model.integrator = "gillespie"
```

Here we are using differential evolution as optimization function and we are creating a stochastic simulation model providing antimony string as model along with necessary arguments. The next step would be providing parameters with their lower and upper bounds

```
bounds = {
    "kdegMdm2":(0.0000001, 0.9),
    "kdegp53":(0.0000001,0.9),
```

```

"kbinMdm2p53": (0.000001, 9),
"krelMdm2p53": (0.00000001, 0.09),
"kphosMdm2": (0.001, 10000.0),
"kdephosMdm2": (0.0001, 900),
"kdegATMMdm2": (0.00000001, 0.9)
}

```

Then to trigger Parameter Estimation we need to call ParameterEstimation from tellurium module.

```

parameter_est = te.ParameterEstimation(stochastic_simulation_model, bounds,
↳stochastic=True, sc=sc)

```

The key points that we need to change here are

- **stochastic**

Can be True/False. True will run stochastic simulation and get the mean of the results. It should be noted that `sc` which represents Spark Context should be provided if you need to set stochastic as True. If set to False, this will run in normal python without Spark and will perform only a single run.

- **sc**

This represents the Spark Context Object created. This is a mandatory argument when stochastic is True

```

from pyspark import SparkContext, SparkConf
conf = SparkConf().setAppName('<APPLICATION_NAME>').setMaster('<SPARK_MASTER>')
sc = SparkContext(conf=conf)

```

After creating ParameterEstimation object we need to set the data which is compared against and then need to run.

```

parameter_est.setDataFromFile(path+FILENAME)
print parameter_est.run(diff_evolution, maxiter=1)

```

In the above block we have configured our data using the `setDataFromFile` method which accepts FILEPATH as an argument. Then we performed `run` method which accepts the optimization function along with the other arguments required for the optimization function. Your result structure may look similar to this.

```

{
    'Estimated Result': array([8.09843570e-04, 2.05751914e-05, 2.06783537e-03, 4.
↳93490582e-04]),
    'Average SSE': 9.2357328599694437,
    'Parameters': ['kdegp53', 'krelMdm2p53', 'kbinMdm2p53', 'kdegMdm2']
}

```

The results are in the form of numpy array along with Average SSE (Sum of Squared Errors) for the simulations. Here is the explanation of the mapping

- **kdegp53** : 8.09843570e-04
- **krelMdm2p53** : 2.05751914e-05
- **kbinMdm2p53** : 2.06783537e-03
- **kdegMdm2** : 4.93490582e-04
- **Average SSE** : 9.2357328599694437

## 6.5 Parameter Scanning

With Distributed nature of tellurium, now you can run parameter scanning for multiple models. More work is currently in progress which enables user to store images in multiple formats and also in HDFS or any other file system.

```

model_one_road_runner = '''
    J1: $Xo -> x; 0.1 + k1*x^2/(k2+x^3);
    x -> $w; k3*x;

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

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

    k1 = 0.8;
    k2 = 0.7;
    k3 = 0.5;
    x = 0;
'''

model_one_parameters = {"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'] }

model_two_parameters = {"startTime" : 0,"endTime" : 20,"numberOfPoints" : 60,
↳ "polyNumber" : 10,"endValue" : 1.5,"alpha" : 0.6,"value" : "x","selection" : "x",
↳ "color" : ['#0F0F3D', '#141452', '#1A1A66', '#1F1F7A', '#24248F', '#2929A3', '#2E2EB8
↳ ', '#3333CC', '#4747D1', '#5C5CD6'] }

```

We have created two models and parameters separately and we are interested in running Parameter Scan for these models (it can be run for any number of models Parallely), we wrap it an array and call distributed\_parameter\_scanning method.

```

plots = te.distributed_parameter_scanning(sc, [(model_one_road_runner,model_one_
↳ parameters),model_two_road_runner,model_two_parameters]),"plotPolyArray")

```

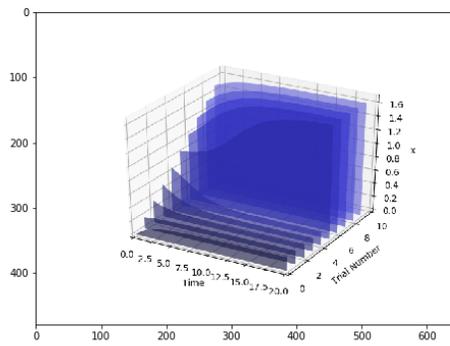
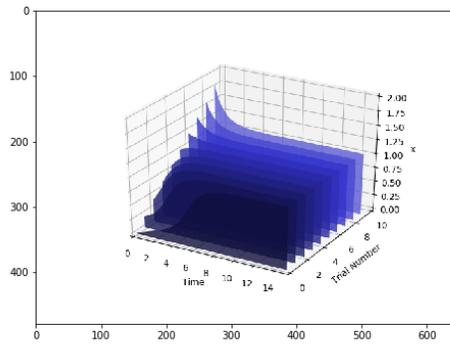
After the computation, plotting the results is easy

```

for fig in plots:
    te.plotImage(fig)

```

You should see results similar to the ones below



Check this space for more information . . .



## 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` – package name

**Returns**

`tellurium.upgradePackage` (*name*)

Upgrade pip package.

**Parameters** `name` – package name

**Returns**

`tellurium.searchPackage` (*name*)

Search pip package for package name.

**Parameters** `name` – package name

**Returns**

```
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')
```

```

Requirement already satisfied: cobra in /home/poltergeist/.config/Tellurium/telocal/
↳python-3.6.1/lib/python3.6/site-packages
Requirement already satisfied: numpy>=1.6 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already satisfied: tabulate in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already satisfied: future in /home/poltergeist/.config/Tellurium/telocal/
↳python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already satisfied: swiglpk in /home/poltergeist/.config/Tellurium/telocal/
↳python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already satisfied: pandas>=0.17.0 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already satisfied: optlang>=1.2.1 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already satisfied: ruamel.yaml<0.15 in /home/poltergeist/.config/
↳Tellurium/telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already satisfied: pytz>=2011k in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from pandas>=0.17.0->cobra)
Requirement already satisfied: python-dateutil>=2 in /home/poltergeist/.config/
↳Tellurium/telocal/python-3.6.1/lib/python3.6/site-packages (from pandas>=0.17.0->
↳cobra)
Requirement already satisfied: sympy>=1.0.0 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from optlang>=1.2.1->cobra)
Requirement already satisfied: six>=1.9.0 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from optlang>=1.2.1->cobra)
Requirement already satisfied: mpmath>=0.19 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from sympy>=1.0.0->optlang>=1.2.1->
↳>cobra)
Requirement already up-to-date: cobra in /home/poltergeist/.config/Tellurium/telocal/
↳python-3.6.1/lib/python3.6/site-packages
Requirement already up-to-date: numpy>=1.6 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already up-to-date: tabulate in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already up-to-date: future in /home/poltergeist/.config/Tellurium/telocal/
↳python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already up-to-date: swiglpk in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already up-to-date: pandas>=0.17.0 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already up-to-date: optlang>=1.2.1 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already up-to-date: ruamel.yaml<0.15 in /home/poltergeist/.config/
↳Tellurium/telocal/python-3.6.1/lib/python3.6/site-packages (from cobra)
Requirement already up-to-date: pytz>=2011k in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from pandas>=0.17.0->cobra)
Requirement already up-to-date: python-dateutil>=2 in /home/poltergeist/.config/
↳Tellurium/telocal/python-3.6.1/lib/python3.6/site-packages (from pandas>=0.17.0->
↳cobra)
Requirement already up-to-date: sympy>=1.0.0 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from optlang>=1.2.1->cobra)
Requirement already up-to-date: six>=1.9.0 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from optlang>=1.2.1->cobra)
Requirement already up-to-date: mpmath>=0.19 in /home/poltergeist/.config/Tellurium/
↳telocal/python-3.6.1/lib/python3.6/site-packages (from sympy>=1.0.0->optlang>=1.2.1->
↳>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

### 7.2.1 Version information

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)
```

```
/home/poltergeist/.config/Tellurium/telocal/python-3.6.1/lib/python3.6/site-
↳packages/matplotlib/___init___py:1405: UserWarning:
This call to matplotlib.use() has no effect because the backend has already
been chosen; matplotlib.use() must be called before pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.
```

```
warnings.warn(_use_error_msg)
```

```
te.__version__
2.0.1
te.getTelluriumVersion()
2.0.1
-----
tellurium : 2.0.1
roadrunner : 1.4.21; Compiler: gcc 4.8.2, C++ version: 199711; JIT Compiler: LLVM-3.3;
↳ Date: Jul 5 2017, 18:38:02; LibSBML Version: 5.14.0
antimony : 2.9.3
libsbml : 5.15.0
libsedml : 402
phrasedml : 1.0.7
-----
```

## 7.2.2 Repeat simulation without notification

```
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 of 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()
```

## 7.2.3 File helpers for reading and writing

```
# create tmp file
import tempfile
ftmp = tempfile.NamedTemporaryFile(suffix=".xml")
# load model
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

    % 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));

x_dot = [
    - 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;

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
end

```

```
function z = leq(a,b)
    if a <= b
        z = 1;
    else
        z = 0;
    end
end

function z = neq(a,b)
    if a ~= b
        z = 1;
    else
        z = 0;
    end
end

function z = and(varargin)
    result = 1;
    for k=1:nargin
        if varargin{k} ~= 1
            result = 0;
            break;
        end
    end
    z = result;
end

function z = or(varargin)
    result = 0;
    for k=1:nargin
        if varargin{k} ~= 0
            result = 1;
            break;
        end
    end
    z = result;
end

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

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`

### 7.3.1 Load Antimony

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
te.setDefaultPlottingEngine('matplotlib')

# Load an antimony model
ant_model = '''
    S1 -> S2; k1*S1;
    S2 -> S3; k2*S2;

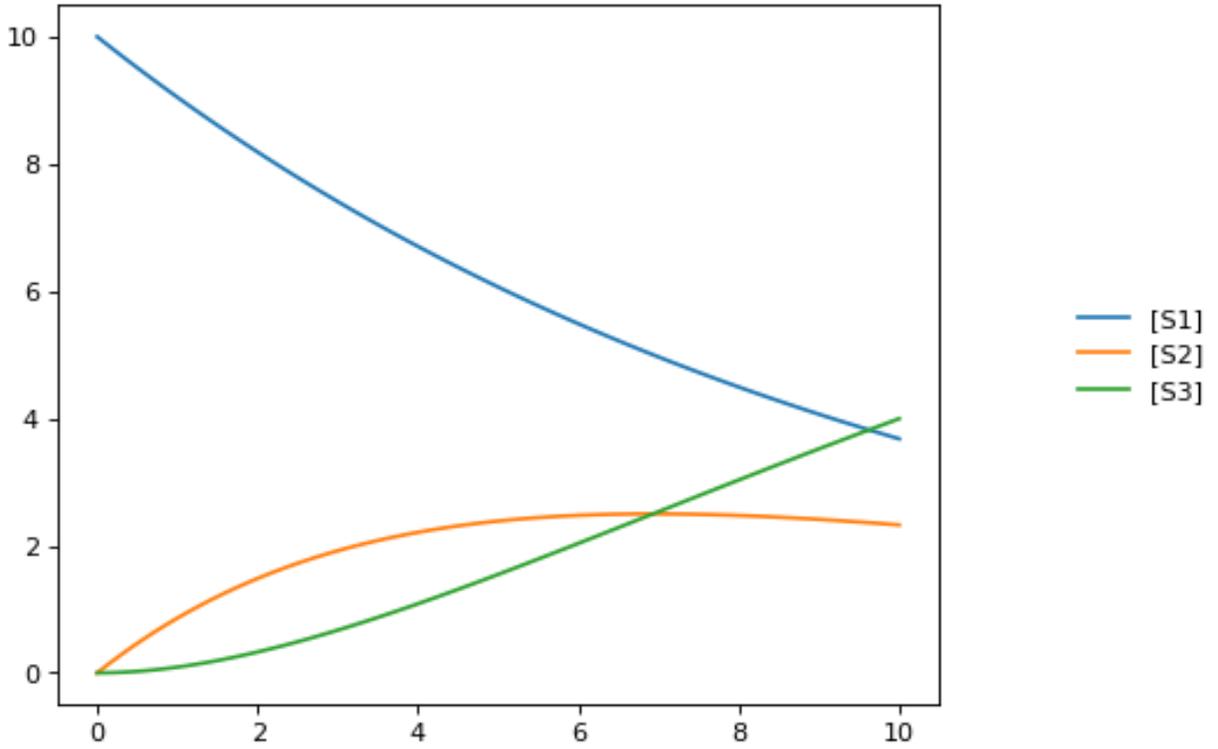
    k1= 0.1; k2 = 0.2;
    S1 = 10; S2 = 0; S3 = 0;
'''

# 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)

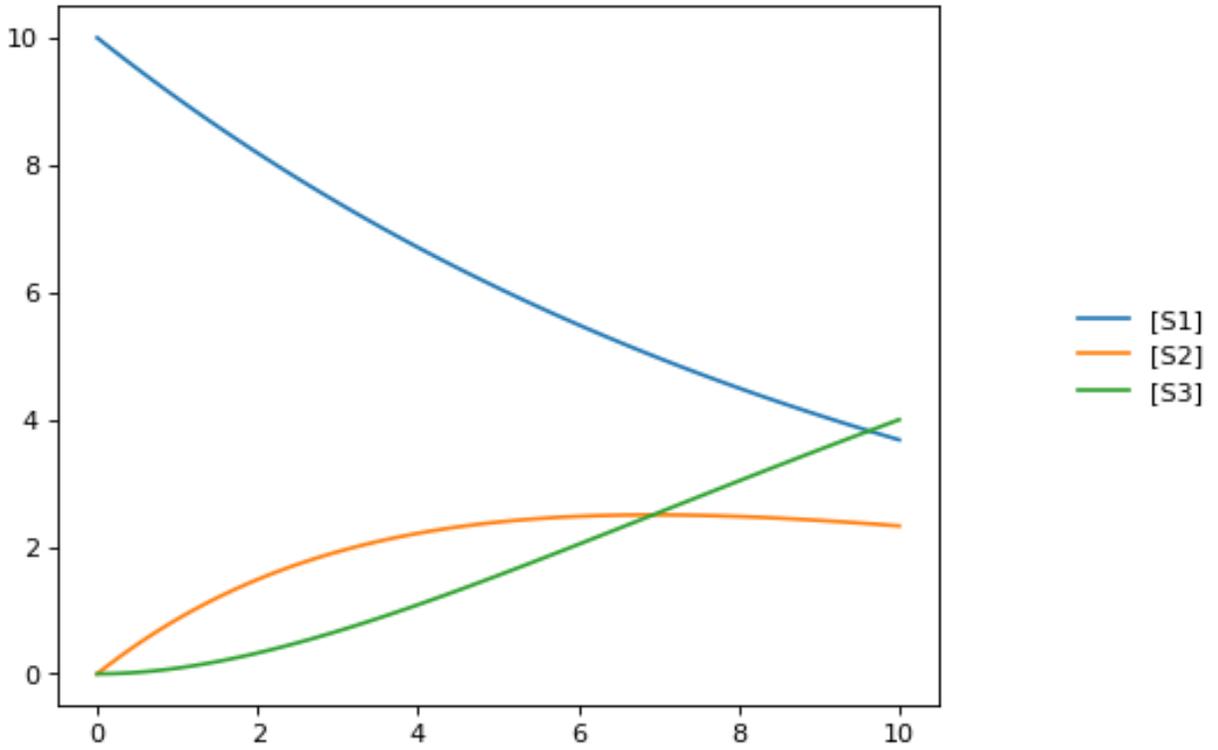
# The method loada is simply a shortcut to loadAntimonyModel
print('--- load using tellurium ---')
r = te.loada(ant_model)
result = r.simulate (0, 10, 100)
r.plot(result)

# 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`

### 7.4.1 Antimony, SBML, CellML

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 v2.9.3 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()* :return: Antimony :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

## 7.5.1 SBML

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
te.setDefaultPlottingEngine('matplotlib')
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 v2.9.3 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>
        <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>

```

## 7.5.2 Antimony

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.loada('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 v2.9.3
// 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;

```

## 7.5.3 CellML

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>S2</ci>
</apply>
<ci>_J0</ci>
</apply>
</math>
<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>
</component>
<group>
<relationship_ref relationship="encapsulation"/>
<component_ref component="__main"/>
</group>
</model>

```

## 7.5.4 Matlab

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).
% 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
    ];

    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;

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
            result = 1;
            break;
        end
    end
    z = result;
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;
    end

function z = root(a,b)
    z = a^(1/b);

```

## 7.5.5 Using Antimony Directly

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/>
<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
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 the 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 from time zero to 40 time units, on a grid with 20 points
# using the give 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

## 7.6.1 Stochastic simulation

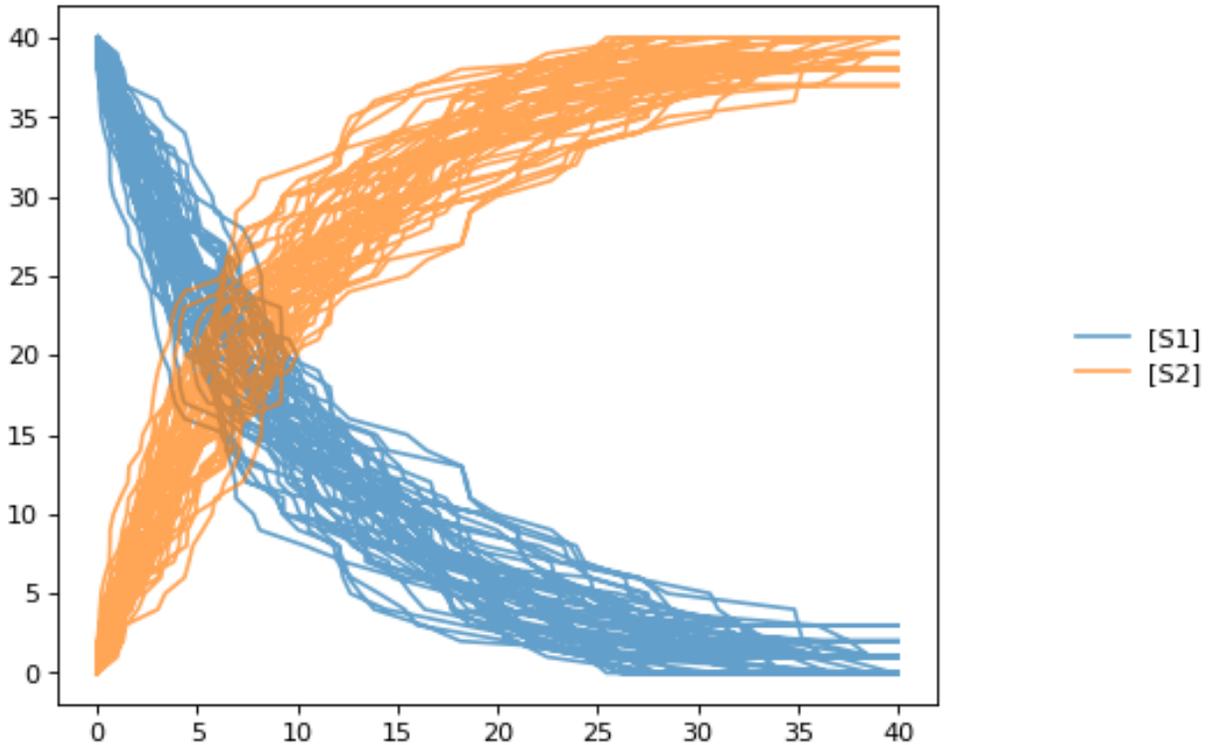
Stochastic simulations can be run by changing the current integrator type to 'gillespie' or by using the `r.gillespie` function.

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

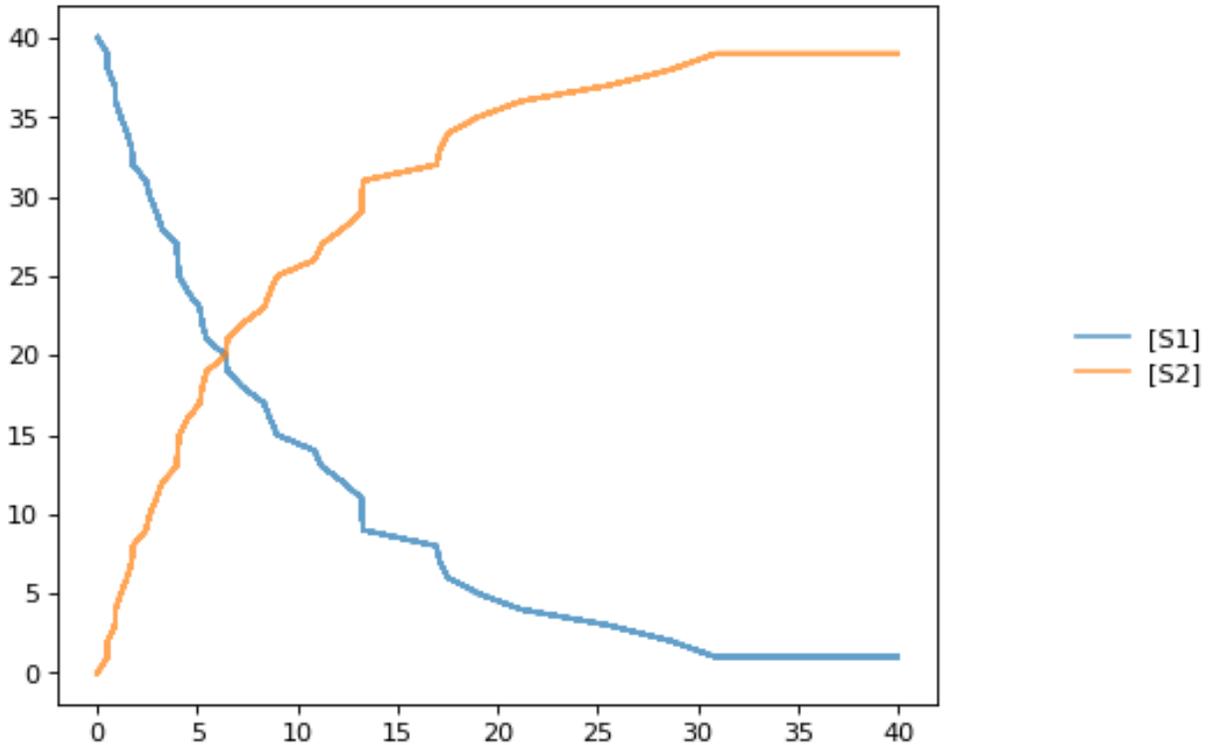
```
r.reset()
s = r.simulate(0, 40)
results.append(s)
r.plot(s, show=False, alpha=0.7)
te.show()
```



## 7.6.2 Seed

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()
```



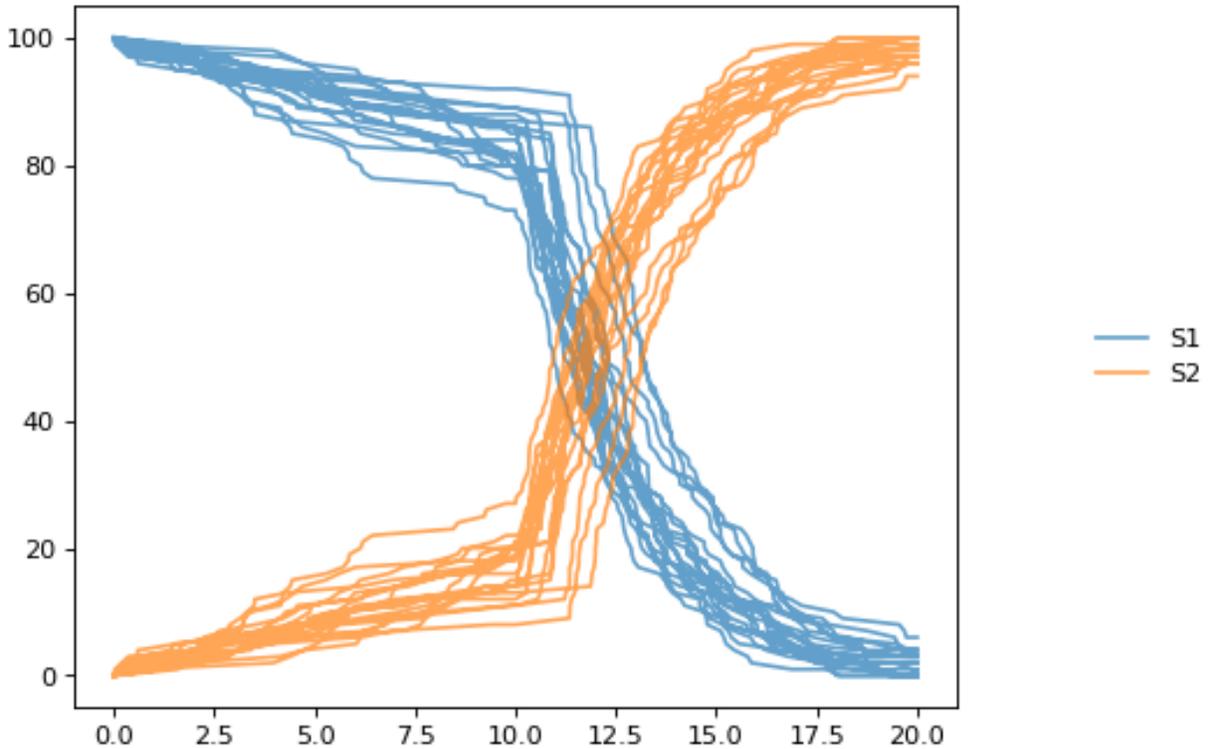
### 7.6.3 Combining Simulations

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 `te.plot` function, you can pass the parameter names, which controls the names that will be used in the figure legend, and `tags`, which ensures that traces with the same tag will be drawn with the same color.

```
import tellurium as te
import numpy as np

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)
    # change in parameter after the first half of the simulation
    r.k1 = r.k1*20
    res2 = r.gillespie(10, 20)
    sim = np.vstack([res1, res2])
    te.plot(sim[:,0], sim[:,1:], alpha=0.7, names=['S1', 'S2'], tags=['S1', 'S2'],
    ↪ show=False)
te.show()
```



## 7.7 Distributed Stochastic Simulation

Use these in order to run simulations in distributed environment.

```
class tellurium.StochasticSimulationModel (model=None, integrator='gillespie',
seed=1234, variable_step_size=False,
from_time=0, to_time=40, step_points=50)
```

## 7.8 Distributed Stochastic Simulation Utility

Use these in order to run simulations in distributed environment. It uses the object defined with StochasticSimulationModel.

```
tellurium.distributed_stochastic_simulation (sc, stochastic_model_object,
num_simulations, model_type='antimony')
```

## 7.9 Plot Distributed Stochastic Simulation Results

To plot the results retrieved from distributed\_stochastic\_simulation.

```
tellurium.plot_stochastic_result (result)
```

## 7.10 Distributed Parameter Scanning

Parameter Scanning one/more models in a distributed environment

```
tellurium.distributed_parameter_scanning(sc, list_of_models, function_name, anti-
                                         mony='antimony')
```

## 7.11 Plotting Image of Parameter Scan

Helps in plotting the results parameter Scanning of one/more models run in a distributed environment

```
tellurium.plotImage(img)
```

## 7.12 Distributed Sensitivity Analysis

Use these in order to run Sensitivity Analysis in distributed environment.

```
class tellurium.SensitivityAnalysis(model=None, sbml=False, conservedMoietyAnaly-
                                   sis=True)
```

## 7.13 Distributed Sensitivity Analysis Utility

Running the Sensitivity analysis using the model created using tellurium.SensitivityAnalysis

```
tellurium.distributed_sensitivity_analysis(sc, sensitivity_analysis_model, calcula-
                                         tion=None)
```

## 7.14 Math

Only one routine is currently available in this group which is a routine to compute the eigenvalues of given a matrix.

```
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

## 7.15 Plotting

Two useful plotting routines. They assume that the first column in the array is the x-axis and the second and subsequent columns represent curves on the y-axis.

```
tellurium.plotArray(result, loc='upper right', show=True, resetColorCycle=True, xlabel=None, yla-
                    bel=None, title=None, xlim=None, ylim=None, xscale='linear', yscale='linear',
                    grid=False, labels=None, **kwargs)
```

Plot an array.

The first column of the array must be the x-axis and remaining columns the y-axis. Returns a handle to the plotting object. Note that you can add plotting options as named key values after the array. To add a legend, include the label legend values:

```
te.plotArray(m, labels=['Label 1', 'Label 2', etc])
```

Make sure you include as many labels as there are curves to plot!

Use `show=False` to add multiple curves. Use `color='red'` to use the same color for every curve.

```
import numpy as np
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=(0, 5))
```

```
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, xlim=None, ylim=None,
      logx=False, logy=False, xscale='linear', yscale='linear', grid=False, ordinates=None,
      tag=None, **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, than 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** – show the plot, use `show=False` to plot multiple simulations in one plot
- **xtitle** – x-axis label (str)
- **ytitle** – y-axis label (str)
- **title** – plot title (str)
- **xlim** – limits on x-axis (tuple [start, end])
- **ylim** – limits on y-axis

- **logx** –
- **logy** –
- **xscale** – ‘linear’ or ‘log’ scale for x-axis
- **yscale** – ‘linear’ or ‘log’ scale for y-axis
- **grid** – 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
- **kwargs** – additional matplotlib keywords like marker, lineStyle, color, alpha, ...

### Returns

## 7.15.1 Draw diagram

This example shows how to draw a network diagram, requires `graphviz`.

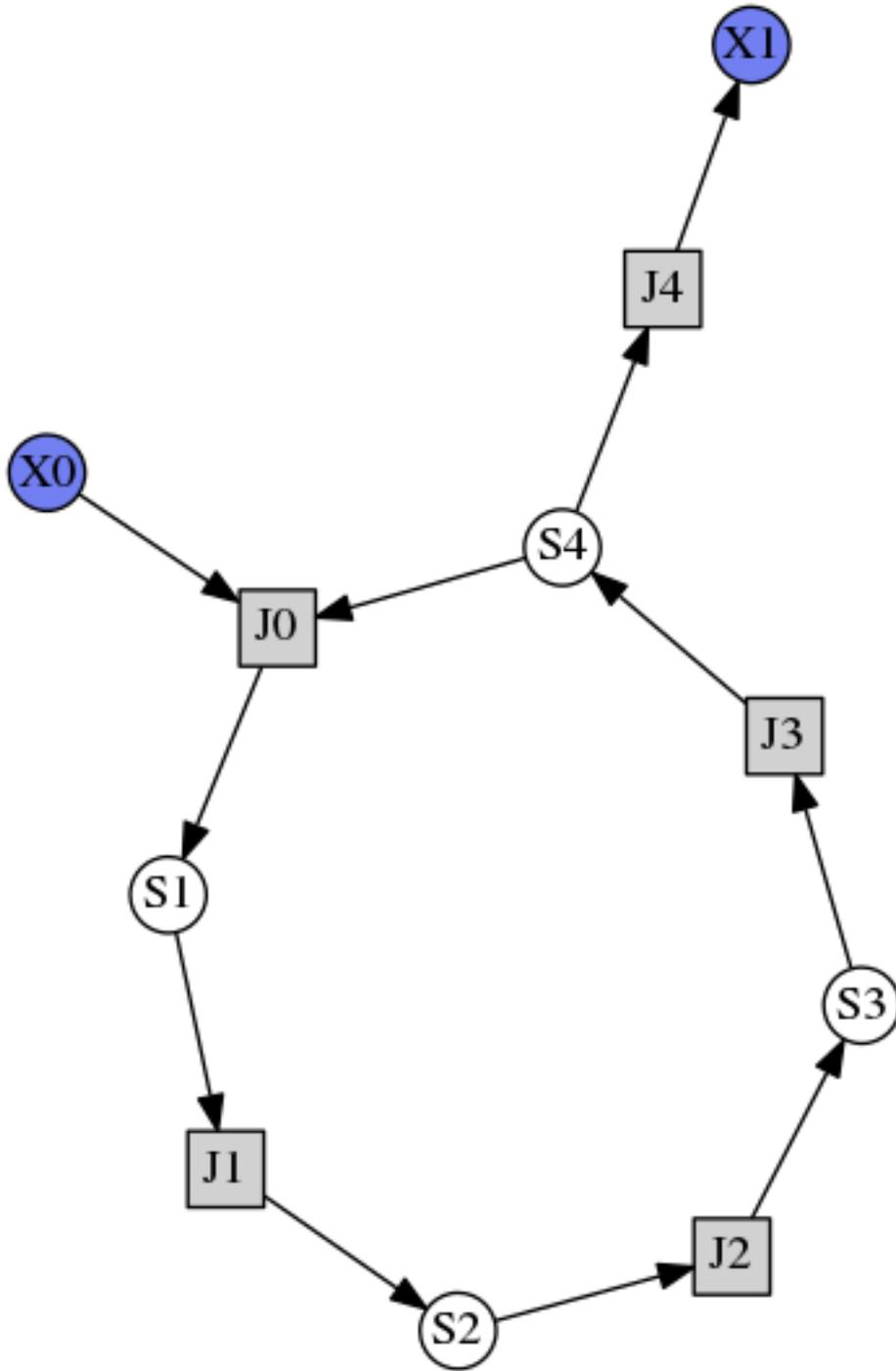
```
import tellurium as te
te.setDefaultPlottingEngine('matplotlib')

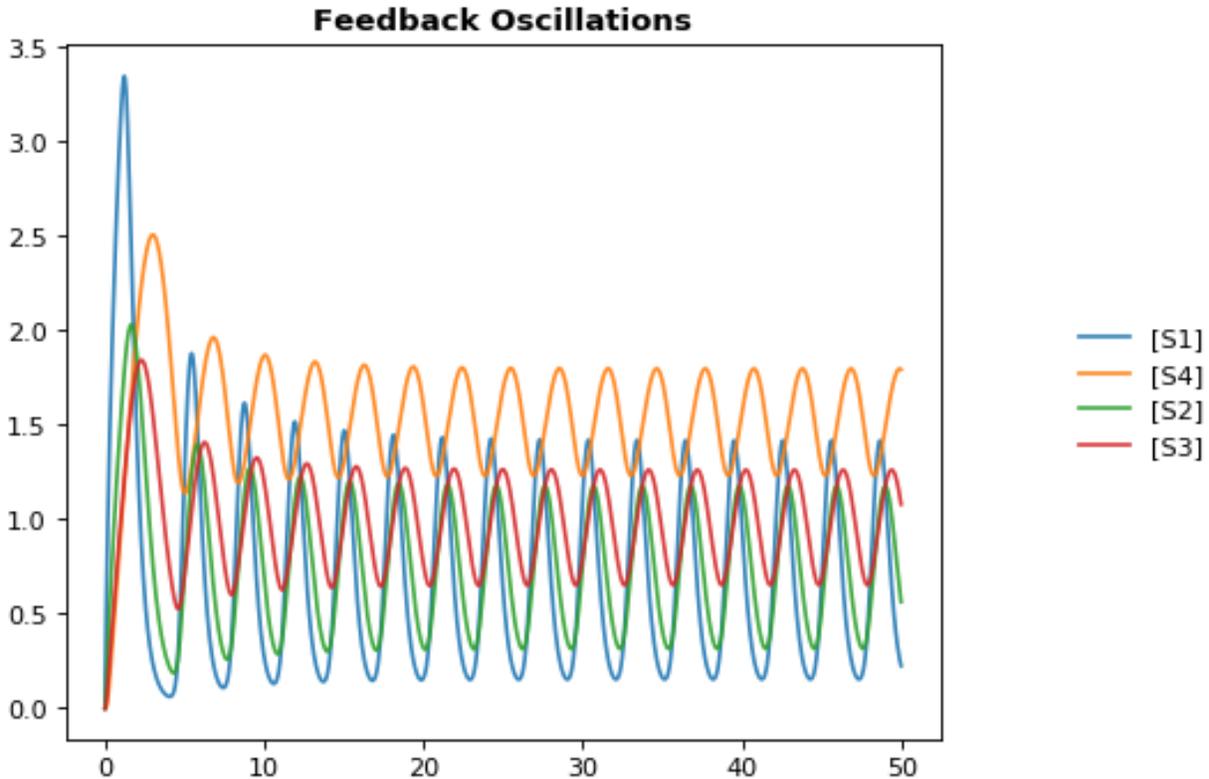
r = te.loada('''
model feedback()
  // Reactions:http://localhost:8888/notebooks/core/tellurium_export.ipynb#
  J0: $X0 -> S1; (VM1 * (X0 - S1/Keq1))/(1 + X0 + S1 + S4^h);
  J1: S1 -> S2; (10 * S1 - 2 * S2) / (1 + S1 + S2);
  J2: S2 -> S3; (10 * S2 - 2 * S3) / (1 + S2 + S3);
  J3: S3 -> S4; (10 * S3 - 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:
  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);
```





### 7.15.2 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
te.setDefaultPlottingEngine('matplotlib')
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)
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",
```

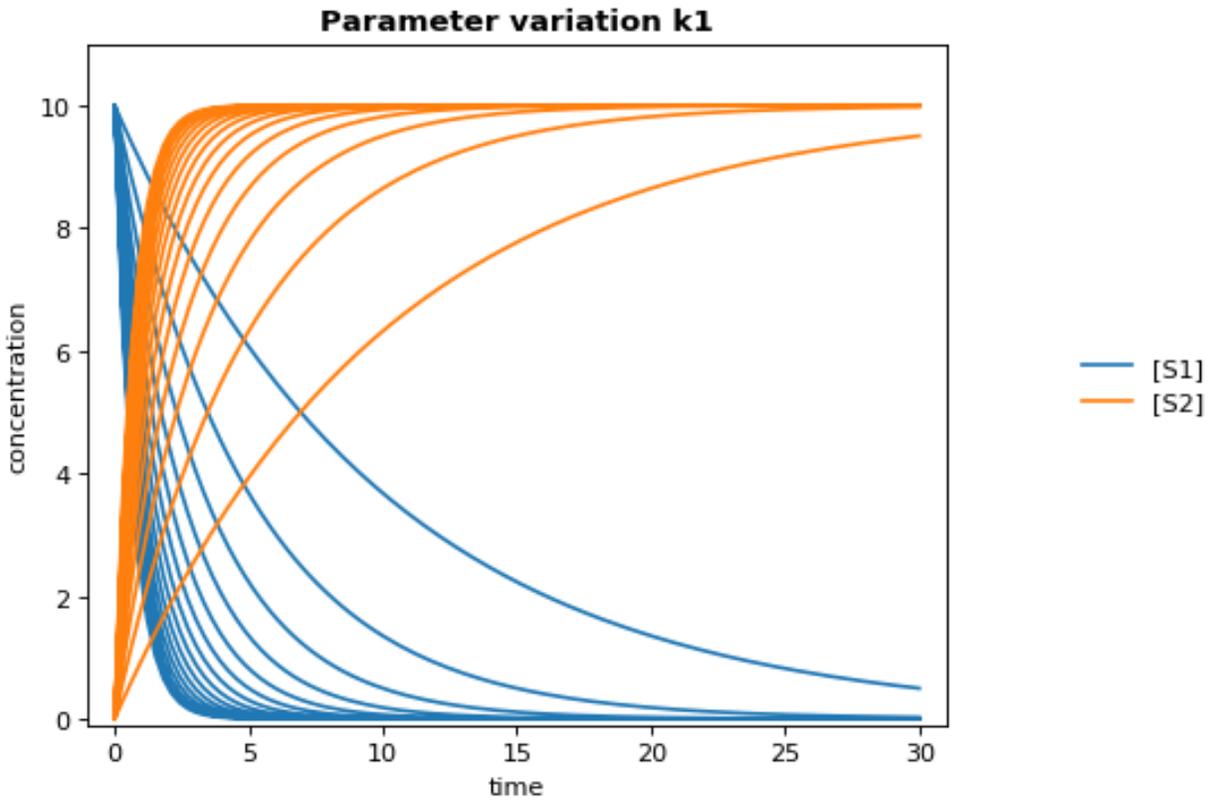
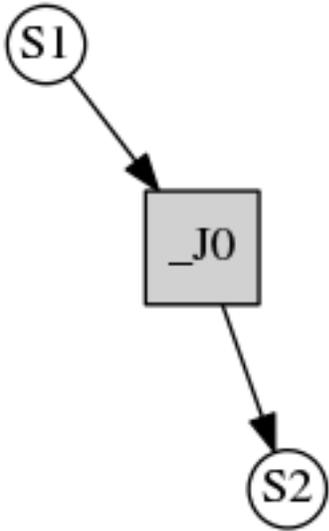
```

        xlim=[-1, 31], ylim=[-0.1, 11])

te.show()

print('Reference Simulation: k1 = {}'.format(r.k1))
print('Parameter variation: k1 = {}'.format(k1_values))

```



```

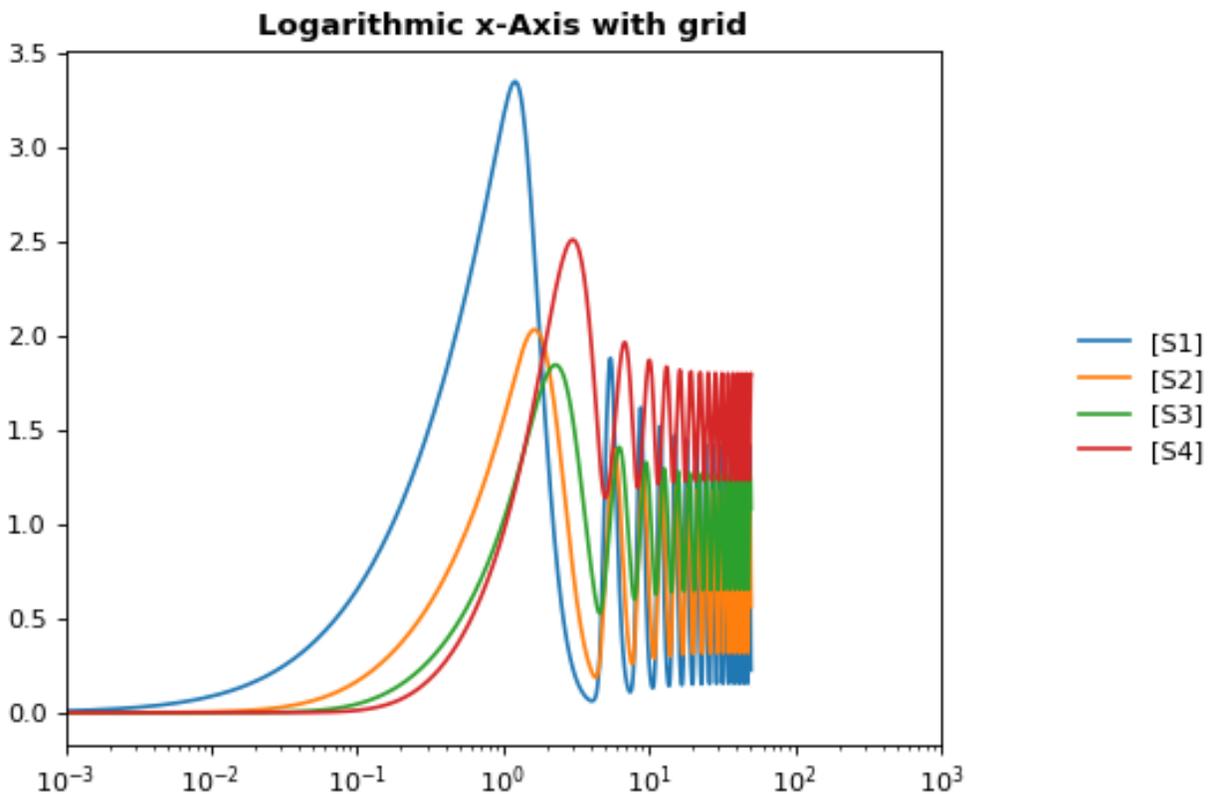
Reference Simulation: k1 = 1.5
Parameter variation: k1 = [ 0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.  1.1  1.
→ 2  1.3  1.4  1.5]

```

### 7.15.3 Logarithmic axis

The axis scale can be adapted with the `xscale` and `yscale` settings.

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



### 7.16 Model Reset

Use these routines reset your model back to particular states

```
class tellurium.tellurium.ExtendedRoadRunner (*args, **kwargs)
```

```
    resetAll ()
```

Reset all model variables to CURRENT init(X) values.

This resets all variables, S1, S2 etc to the CURRENT init(X) values. It also resets all parameters back to the values they had when the model was first loaded.

**resetToOrigin()**

Reset model to state when first loaded.

This resets the model back to the state when it was FIRST loaded, this includes all `init()` and parameters such as `k1` etc.

**identical to:** `r.reset(roadrunner.SelectionRecord.ALL)`

## 7.16.1 Reset model values

The `reset` function of a `RoadRunner` instance reset the system variables (usu. species concentrations) to their respective initial values. `resetAll` also resets parameters. `resetToOrigin` completely resets the model.

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

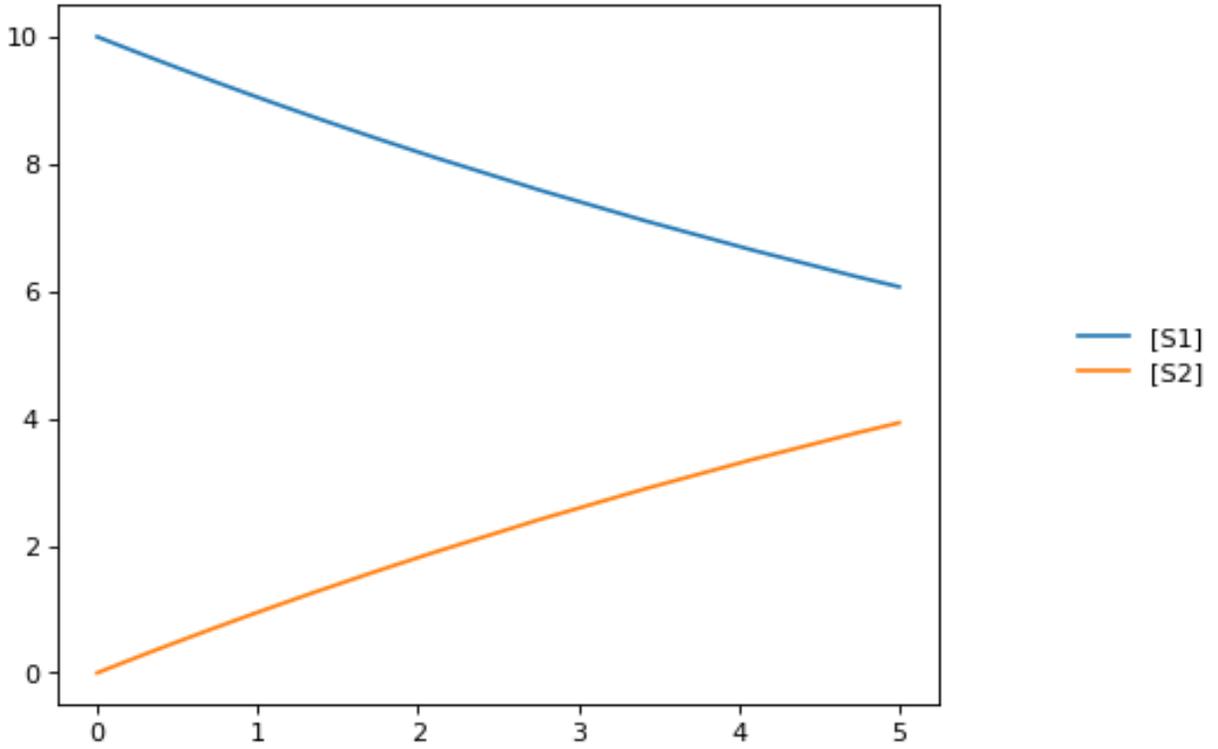
r = te.loada ('S1 -> S2; k1*S1; k1 = 0.1; S1 = 10')
r.integrator.setValue('variable_step_size', True)
# simulate model
sim1 = r.simulate(0, 5)
print('*** sim1 ***')
r.plot(sim1)

# 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)

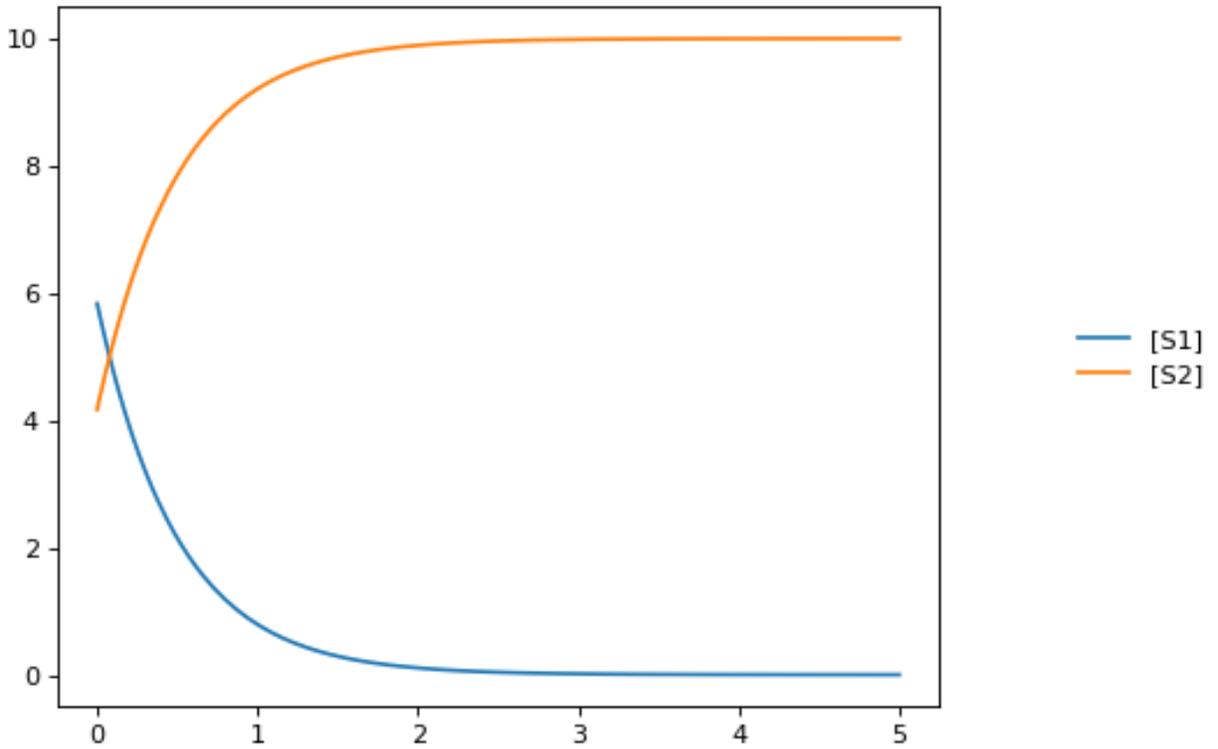
# Reset initial concentrations, does not affect the changed parameter
r.reset()
sim3 = r.simulate(0, 5)
print('-- sim3 --')
print('reset initial concentrations but keep changed parameter')
r.plot(sim3)

# Reset model to the state it was loaded
r.resetToOrigin()
sim4 = r.simulate(0, 5)
print('-- sim4 --')
print('reset all to origin')
r.plot(sim4);
```

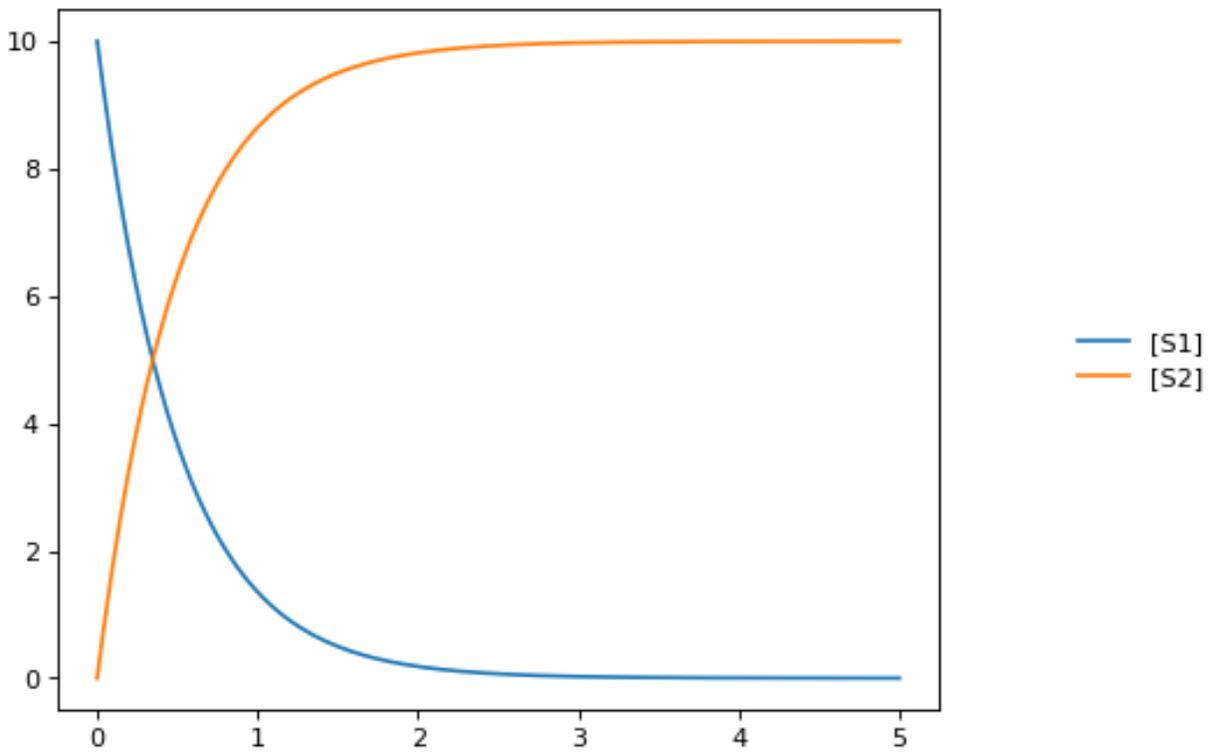
**\* sim1 \***



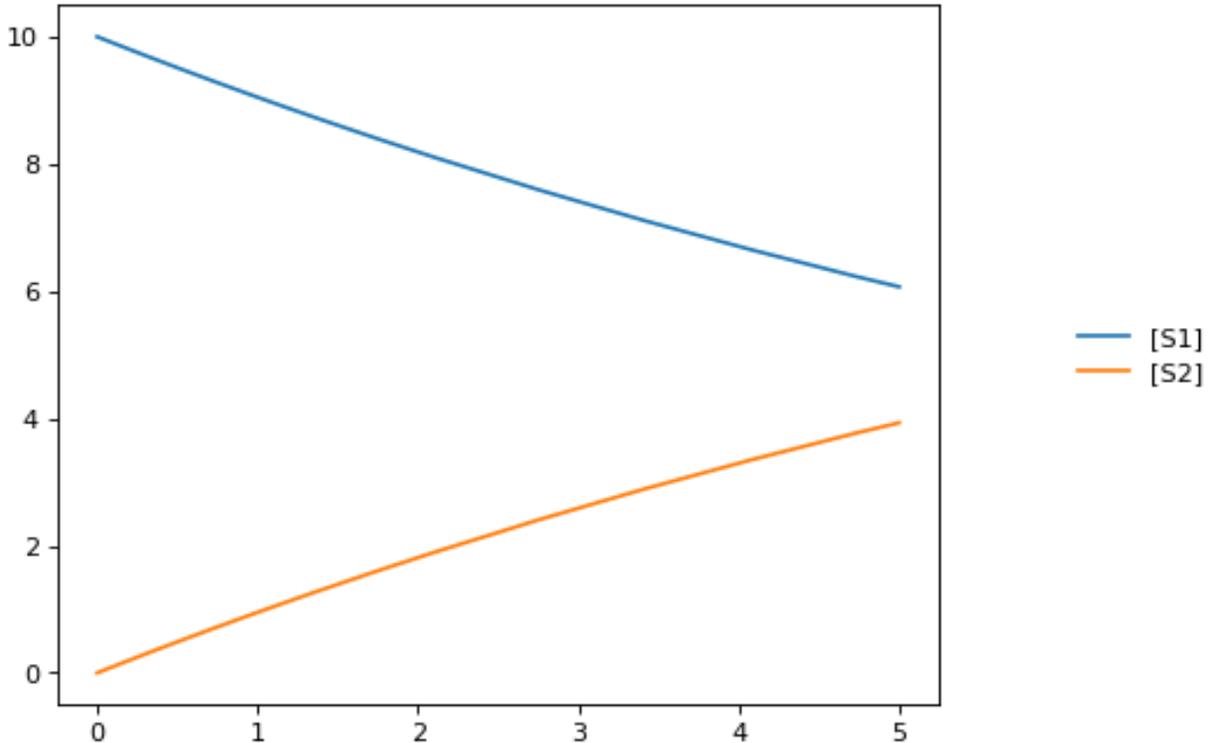
```
-- sim2 --
continue simulation from final concentrations with changed parameter
```



```
-- sim3 --  
reset initial concentrations but keep changed parameter
```



```
-- sim4 --  
reset all to origin
```



## 7.17 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** ()

ExecutableModel.getStateVector([stateVector])

Returns a vector of all the variables that represent the state of the system. The state is considered all values which change with the dynamics of the model. This would include all species which are produced or consumed by reactions, and all variables which are defined by rate rules.

Variables such as global parameters, compartments, or boundary species which do not change with the model dynamics are considered parameters and are thus not part of the state.

In performance critical applications, the optional stateVector array should be provided where the output variables will be written to.

**Parameters** `stateVector` (*numpy.ndarray*) – an optional numpy array where the state vector variables will be written. If no state vector array is given, a new one will be constructed and returned.

This should be the same length as the model state vector.

**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.

**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`

RoadRunner stores all initial conditions separately from the model state variables. This means that you can update the initial conditions at any time, and it does not affect the current state of the model. To reset the model, that is, reset it to its original state, or a new original state where what has changed the initial conditions use the `reset ()` method.

The following methods allow access to the floating species initial condition values:

**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.

## 7.18 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

```
import tellurium as te

# To get the builtin models use listTestModels
print(te.listTestModels())
```

```
['EcoliCore.xml', 'test_1.xml', 'feedback.xml', 'linearPathwayClosed.xml',
↪ 'linearPathwayOpen.xml']
```

### 7.18.1 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 v2.9.0
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
```

## 7.19 Model Methods

Routines flattened from model, aves 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`

RoadRunner stores all initial conditions separately from the model state variables. This means that you can update the initial conditions at any time, and it does not affect the current state of the model. To reset the model, that is, reset it to its original state, or a new original state where what has changed the initial conditions use the `reset ()` method.

The following methods allow access to the floating species initial condition values:

**getFloatingSpeciesIds ()**

Return a list of floating species sbml 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 ()**

Rate of change of all state variables in the model.

**Returns** rate of change of all state variables (eg species) in the model.

**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.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.executeInlineOmex` (*inline\_omex*)

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.

The first column of the array must be the x-axis and remaining columns the y-axis. Returns a handle to the plotting object. Note that you can add plotting options as named key values after the array. To add a legend, include the label legend values:

```
te.plotArray(m, labels=['Label 1', 'Label 2', etc])
```

Make sure you include as many labels as there are curves to plot!

Use `show=False` to add multiple curves. Use `color='red'` to use the same color for every curve.

```
import numpy as np
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=((0, 5)))
```

`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 ‘pyplot’.

`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, see [Tellurium on UserVoice](#)..

## 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**

antimonyToCellML() (in module tellurium), 122  
 antimonyToCellML() (in module tellurium.tellurium), 157  
 antimonyToSBML() (in module tellurium), 122  
 antimonyToSBML() (in module tellurium.tellurium), 157

**B**

bs() (tellurium.tellurium.ExtendedRoadRunner method), 149

**C**

cellmlToAntimony() (in module tellurium), 122  
 cellmlToAntimony() (in module tellurium.tellurium), 157  
 cellmlToSBML() (in module tellurium), 122  
 cellmlToSBML() (in module tellurium.tellurium), 157  
 colorCycle() (in module tellurium.tellurium), 158  
 convertAndExecuteCombineArchive() (in module tellurium.tellurium), 158  
 convertCombineArchive() (in module tellurium), 36  
 convertCombineArchive() (in module tellurium.tellurium), 158

**D**

distributed\_parameter\_scanning() (in module tellurium), 139  
 distributed\_sensitivity\_analysis() (in module tellurium), 139  
 distributed\_stochastic\_simulation() (in module tellurium), 138  
 draw() (tellurium.tellurium.ExtendedRoadRunner method), 140  
 DumpJSONInfo() (in module tellurium.tellurium), 157  
 dv() (tellurium.tellurium.ExtendedRoadRunner method), 149

**E**

executeInlineOmex() (in module tellurium), 36

executeInlineOmex() (in module tellurium.tellurium), 158  
 executeInlineOmexFromFile() (in module tellurium.tellurium), 158  
 exportInlineOmex() (in module tellurium), 36  
 exportInlineOmex() (in module tellurium.tellurium), 158  
 exportToAntimony() (tellurium.tellurium.ExtendedRoadRunner method), 124  
 exportToCellML() (tellurium.tellurium.ExtendedRoadRunner method), 124  
 exportToMatlab() (tellurium.tellurium.ExtendedRoadRunner method), 124  
 exportToSBML() (tellurium.tellurium.ExtendedRoadRunner method), 125  
 ExtendedRoadRunner (class in tellurium.tellurium), 124, 134, 140, 145, 149, 152  
 extractFileFromCombineArchive() (in module tellurium), 36  
 extractFileFromCombineArchive() (in module tellurium.tellurium), 158

**F**

fjac() (tellurium.tellurium.ExtendedRoadRunner method), 150  
 fs() (tellurium.tellurium.ExtendedRoadRunner method), 150

**G**

getAntimony() (tellurium.tellurium.ExtendedRoadRunner method), 125  
 getBoundarySpeciesConcentrations() (tellurium.tellurium.ExtendedRoadRunner method), 153  
 getBoundarySpeciesIds() (tellurium.tellurium.ExtendedRoadRunner method), 153  
 getCellML() (tellurium.tellurium.ExtendedRoadRunner method), 125

- getCompartmentIds()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 153
- getCompartmentVolumes()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 153
- getConservedMoietyValues()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 153
- getCurrentAntimony()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 125
- getCurrentCellML()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 125
- getCurrentMatlab()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 125
- getDefaultPlottingEngine() (in module  
     lurium.tellurium), 158
- getEigenvalues() (in module tellurium), 139
- getEigenvalues() (in module tellurium.tellurium), 158
- getFloatingSpeciesConcentrations()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 153
- getFloatingSpeciesIds()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getGlobalParameterValues()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getLastReport() (in module tellurium.tellurium), 158
- getMatlab() (tellurium.tellurium.ExtendedRoadRunner  
     method), 125
- getNumBoundarySpecies()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getNumCompartments()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getNumConservedMoieties()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getNumFloatingSpecies()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getNumGlobalParameters()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getNumReactions()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getRatesOfChange()  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getReactionIds() (tellurium.tellurium.ExtendedRoadRunner  
     method), 154
- getReactionRates() (tel- (tel-  
     lurium.tellurium.ExtendedRoadRunner  
     method), 154
- getSeed() (tel- (tellurium.tellurium.ExtendedRoadRunner  
     method), 134
- getTelluriumVersion() (in module tellurium), 113
- getTelluriumVersion() (in module tellurium.tellurium),  
     158
- getTestModel() (in module tellurium), 151
- getTestModel() (in module tellurium.tellurium), 158
- getVersionInfo() (in module tellurium), 113
- getVersionInfo() (in module tellurium.tellurium), 159
- gillespie() (tel- (tellurium.tellurium.ExtendedRoadRunner  
     method), 135
- I**
- inIPython() (in module tellurium.tellurium), 159
- installPackage() (in module tellurium), 111
- L**
- listTestModels() (in module tellurium), 151
- listTestModels() (in module tellurium.tellurium), 159
- loada() (in module tellurium), 119
- loada() (in module tellurium.tellurium), 160
- loadAntimonyModel() (in module tellurium), 119
- loadAntimonyModel() (in module tellurium.tellurium),  
     159
- loadCellMLModel() (in module tellurium), 119
- loadCellMLModel() (in module tellurium.tellurium), 159
- loads() (in module tellurium.tellurium), 160
- loadSBMLModel() (in module tellurium), 119
- loadSBMLModel() (in module tellurium.tellurium), 159
- loadTestModel() (in module tellurium), 151
- loadTestModel() (in module tellurium.tellurium), 159
- M**
- model() (in module tellurium.tellurium), 160
- N**
- noticesOff() (in module tellurium), 113
- noticesOff() (in module tellurium.tellurium), 160
- noticesOn() (in module tellurium), 113
- noticesOn() (in module tellurium.tellurium), 160
- P**
- plot() (tel- (tel- (tellurium.tellurium.ExtendedRoadRunner  
     method), 140
- plot\_stochastic\_result() (in module tellurium), 138
- plotArray() (in module tellurium), 139
- plotArray() (in module tellurium.tellurium), 160

plotImage() (in module tellurium), 139  
printVersionInfo() (in module tellurium), 113  
printVersionInfo() (in module tellurium.tellurium), 160

## R

readFromFile() (in module tellurium), 113  
resetAll() (tellurium.tellurium.ExtendedRoadRunner  
method), 145  
resetToOrigin() (tellurium.tellurium.ExtendedRoadRunner  
method), 145  
rs() (tellurium.tellurium.ExtendedRoadRunner method),  
150  
rv() (tellurium.tellurium.ExtendedRoadRunner method),  
150

## S

saveToFile() (in module tellurium), 113  
sbmlToAntimony() (in module tellurium), 122  
sbmlToAntimony() (in module tellurium.tellurium), 161  
sbmlToCellML() (in module tellurium), 122  
sbmlToCellML() (in module tellurium.tellurium), 161  
searchPackage() (in module tellurium), 111  
SensitivityAnalysis (class in tellurium), 139  
setDefaultPlottingEngine() (in module tel-  
lurium.tellurium), 161  
setLastReport() (in module tellurium.tellurium), 161  
setSavePlotsToPDF() (in module tellurium.tellurium),  
161  
setSeed() (tellurium.tellurium.ExtendedRoadRunner  
method), 135  
sm() (tellurium.tellurium.ExtendedRoadRunner method),  
150  
StochasticSimulationModel (class in tellurium), 138  
sv() (tellurium.tellurium.ExtendedRoadRunner method),  
150

## T

tellurium.tellurium (module), 157

## U

upgradePackage() (in module tellurium), 111

## V

VersionDict() (in module tellurium.tellurium), 157  
vs() (tellurium.tellurium.ExtendedRoadRunner method),  
151