Introduction to Tellurium: A modeling platform for Biochemical Systems

October 5, 2021

Contents

What's it for? Tellurium is a python package that can be used to model things like, glycolysis, protein signaling pathways, gene regulatory networks, predatory prey systems and even the spread of infectious diseases. This document will show you how to build models using Tellurium.

Basic Concepts I

Mass-action Kinetics

The simplest chemical kinetics is mass-action kinetics. This is where the rate of reaction is a simple function of the concentrations of the participating reactants. The mathematical rules that determines a reaction rate is called a **rate law**. There are three mass-action rate laws to be aware of, Figure 1:

Consider the reaction $A \rightarrow$. This describes the degradation of a species A.

Zero Order: $v = v_o$. This is a special case where the rate of reaction, v, is constant and is independent of the concentration of reactant A.

First Order: $v = kA$. k is the rate constant for the reaction and shows that the reaction rate v is proportional to the concentration of A.

Consider the reaction $A + A \rightarrow$. This describes a reaction involving two molecules of A combining in the reaction. Such reactions are described using second order rate laws:

Second Order: $v = kA^2$. k is the rate constant for the reaction and shows that the concentration of A is raised to the power of two.

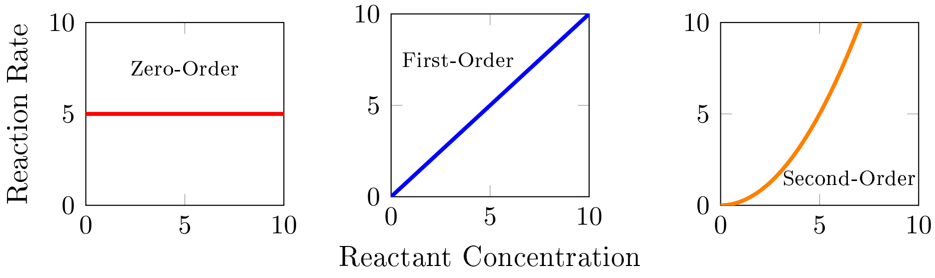


Figure 1: Curves illustrating zero-order, first-order and second-order kinetics.

Reversible Reactions: Reactions can also be described using reversible mass-action kinetics. For example:

$$A \rightarrow B$$

The kinetic rate laws is given by:

$$v = k_1 A - k_2 B$$

where k_1 is the forward rate constant and k_2 is the reverse rate constant. Some useful definitions:

Rate of change of a species $A = \dfrac{dA}{dt}$

Stoichiometric Coefficient, c_i:

$$c_i = \text{Molar amount of Product} - \text{Molar amount of reactant}$$

Rate of reaction v:

$v = \dfrac{1}{c_A}\dfrac{dA}{dt}$

Equilibrium Constant K_{eq} for the reaction:

$$\alpha A + \beta B + \ldots \rightleftharpoons \rho P + \sigma Q + \ldots$$

$$K_{eq} = \frac{P^\rho Q^\sigma \ldots}{A^\alpha B^\beta \ldots}$$

For a reversible unimolecular reaction:

$$A \rightleftharpoons B$$

with forward and reverse rate constants: k_f, k_r respectively it is easy to show that:

$$K_{eq} = \frac{k_f}{k_r}$$

Mass-action Ratio:

$$\Gamma = \frac{P^\rho Q^\sigma \dots}{A^\alpha B^\beta \dots}$$

where the concentrations of reactants and products are measured *in vivo*.

Disequilibrium Ratio:

$$\rho = \frac{\Gamma}{K_{eq}}$$

Enzyme Kinetic Rate Laws

An enzyme mechanism can be described by:

$$E + S \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} ES \xrightarrow{k_2} E + P \tag{1}$$

This is the irreversible version of the mechanism. By making the assumption that the enzyme substrate complex reaches steady-state very quickly, the following approximation can be written:

$$v = \frac{V_m\ S}{K_m + S} \tag{2}$$

This is the Briggs-Haldane equation even though it is often mislabelled the Michaelis-Menten equation.

DESMOS Demo.

For the reversible case, which is more relevant for cellular models we have:

$$E + S \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} ES \underset{k_{-2}}{\overset{k_2}{\rightleftharpoons}} E + P \tag{3}$$

and the derived equation is a little more complicated:

$$v = \frac{V_f/K_S(S - P/K_{eq})}{1 + S/K_S + P/K_P} \tag{4}$$

Cooperativity - Sigmoid Responses

Cooperativity is an extremely important effect in cellular biology. It provides cells the ability to implement Op Amp like characteristics in the form of sigmoid responses. It is how feedback circuits in biology implement high gains. In metabolism cooperativity is achieved using multi-protein complexes with specific ligand binding sites for the feedback signal.

In protein signaling networks phosphorylation cycles are used to generate very high gains. Feedback is achieved by signals molecules binding to control proteins by sequestration.

In gene regulatory network, high gains are achieve by having multiple bindings sites on DNA regulatory sequences which transctiptional factors can bind to and affect expression. Feedback signals in for the form of other proteins can bind to the transcriptional factors and changes their binding characteristics.

Hormonal control is achieved by receptor binding sites on cell surfaces that feedback into layers of protein phosphorylation cycles.

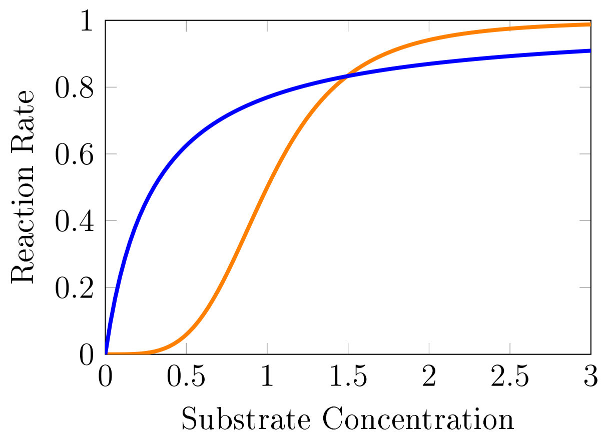


Figure 2: Plot comparing positive cooperativity (lighter line) to a hyperbolic response (darker line).

Here is a simple expression that mimics sigmoid curves called the Hill equation:

$$v = \frac{V_m \; S^n}{K_d + S^n} \tag{5}$$

The higher the power value, n, the greater the gain (or sigmoidicity).

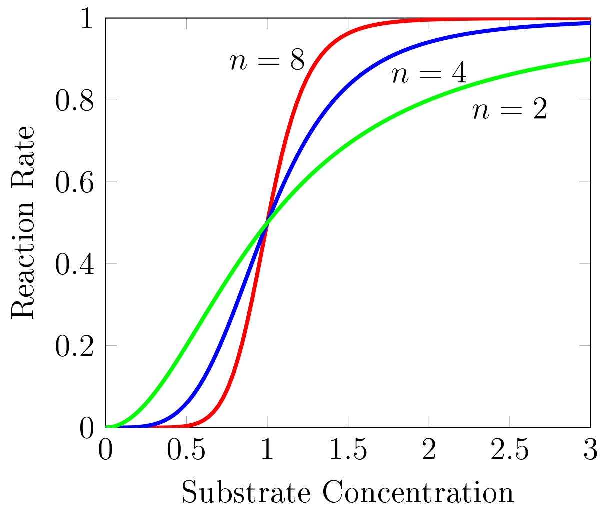


Figure 3: Plots showing the response of the Hill equation set to the indicated values for n, $K_H = 1$.

Reversible version exist for the Hill equation but aren't often used although they should be.

Adding Signals to the Hill Equation

There are a number of ways to add signal terms to a Hill equations, here is a common equation that does that:

$$v = V_f \left(\frac{A}{K_m}\right)^n \frac{\left(\frac{A}{K_m}\right)^{n-1}}{\left(\frac{A}{K_m}\right)^{n-1} + \frac{1 + (S/K_s)^n}{1 + \sigma(S/K_s)^n}}$$

V_f is the maximal forward reaction rate, K_m is the Michaelis constant for the substrate A. S is the concentration of the signal molecule, n determines the level of sigmoidicity, and σ determines the the signal activates or represses the reaction rate according to the rule:

$$\begin{array}{ll} \sigma < 1 & \text{inhibitor} \\ \sigma > 1 & \text{activator} \end{array}$$

Modeling with Tellurium

Defining Models

Reaction models are defined using the Antimony syntax. A full description of the Antimony syntax can be found at:

`https://tellurium.readthedocs.io/en/latest/antimony.html`

A reaction model can be stored in a file or created as a string at runtime. For example the following shows a single reaction with a simple mass-action rate law:

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

The first part of the string describes a reactant `S1` being converted to a product `S2`. The `->` symbol is the reaction separator between the reactants and products nd indicate the positive direction of the reaction rate. The second part defines the reasction rate law, in this case a simple mass-action rate law. Note that it is unnecessary to declare the species and parameter names before hand. The antimony parser will automatically realize that `k1` is a parameter and `S1` and `S2` are both chemical species. Finally, note the semicolon between the reaction and the rate law.

Any number of reactions can be defined to create larger systems. For example, a linear chain of connected reactions would be defined as follows:

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

Assigning Values

Values can be assigned to parameters and species, for example:

```
S1 -> S2; k1*S1
S2 -> S3; k2*S2
S3 -> S4; k3*S3
k1 = 0.1; k2 = 0.43; k3 = 3.4
S1 = 10
```

Values assigned to chemical species are concentrations. Any implied units in the rate constants and concentrations should be consistent.

Adding Comments to a Model

Comments can be added to an antimony model by either using the one line comment `//`, the multiple comment `/* comment */`, or even the python-style single line comment `# comment`. Examples of all three are shown below:

```
 /* this is multiline comment.
   As you can see it

   stretches over multiple lines
 */

 // Here is a single line comment
 # And finally a python style comment
```

Multiple Reactants and Products

Reactions can include multiple reactants and product to describe bimolecular etc. reactions, for example:

```
S1 + S2 -> S3; k1*S1*S2
```

This describes a reaction where `S1` combines with `S2` to form `S3`. Note that the rate laws has been updated to reflect the new reaction. There is no limit to the number of reactants and products that can be included this way. For example:

```
S1 + S2 -> S3; k1*S1*S2
S3 + S4 + S5- > S1 + S6; k2*S3*S4*S3
S4 + S4 -> S7; k3*S4*S4
```

No unity stoichiometries

No unity stoichiometries can also be specified. For example, the last reaction in the above example was `S4 + S4 -> S7`. this could be replaced with the equivalent form:

```
2 S4 -> S7; k3*S4*S4
```

Running Simulations

Once a model has been defined using the antimony syntax it is possible to load the model into the simulator and run a simulation. To do this the Tellurium package must be imported first:

```
import tellurium as te
```

We can now use the tellurium command `loada`, which stands for load antimony. For example:

```
r = te.loada ('S1 -> S2; k1*S1*k1')
```

or

```
 r = te.loada ('''
   S1 -> S2; k1*S1
   S2 -> S3; k2*S2
   S3 -> S4; k3*S3
   k1 = 0.1; k2 = 0.5; k3 = 0.001
   S1 = 5
 ''')
```

Note that in the second example we use triple quotes to allow a string to be specified on multiple lines. It is also possible to first assign the antimony string to a variable and then use that variable in `loada`. For example:

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

One thing that has not been mentioned is that `loada` returns a variable that represents the loaded model. Any further interaction with the model must be through this returned value. In this case the return values is stored in the variable `r` which stands for roadrunner. Any variable name can be used here.

Once a model has been defined, a simulation can be carried using the simulate method. This is very simply done with:

```
m = r.simulate()
```

This method returns a numpy array of the simulated data. By default the simulator simulates from time zero to time 5 and generates 51 output points, meaning the time interval between each point is 0.1. These values can be changed by including arguments with the `simulate` call. For example, to simulate to an end time of 15 and only generate 10 points we would use:

```
m = r.simulate(0, 15, 10)
```

The first value, `0`, is the time start (which need not necessarily be zero), the second value is the end time and the third value the number of points to return to the caller. The output generated by this is shown below for the simple model `S1 -> S2`:

```
[[0, 10, 0],
[1.66667, 8.46482, 1.53518],
[3.33333, 7.1653, 2.8347],
[5, 6.06529, 3.93471],
[6.66667, 5.13416, 4.86584],
[8.33333, 4.34597, 5.65403],
[10, 3.67878, 6.32122],
[11.6667, 3.11402, 6.88598],
[13.3333, 2.63596, 7.36404],
[15, 2.2313, 7.7687]]
```

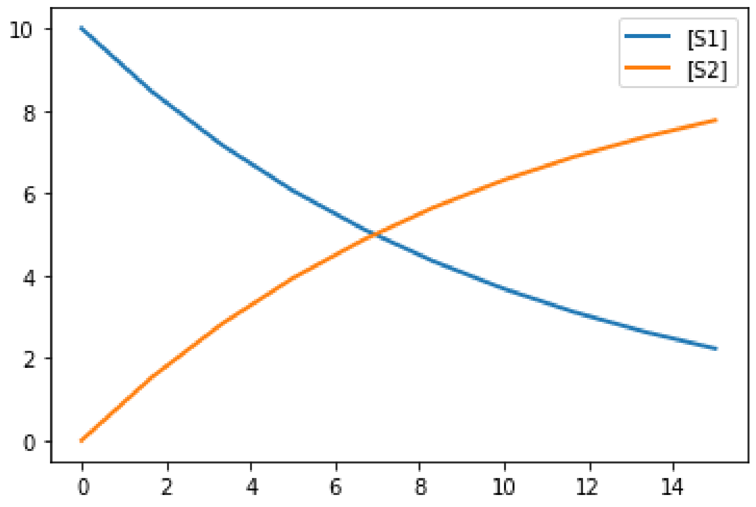
Note: By default, `simulate` will return the time variable in the first column and all chemical species in the remaining columns.

Plotting

To plot the results of the simulation we can use the `r.plot()` method.

```
r.plot()
```

This yields the following plot:



Floating and Boundary Species

Chemical species can be grouped into two categories. One group includes those species that evolve in time during the simulation, these are called the `floating species`. The second group includes those chemical species that are fixed during a simulation, these are called **boundary species**.

By default all species a floating. To make a species a boundary species, there are two ways. For large models it is recommended to use the antimony statement:

```
const S1
```

For example:

```
r = te.loada ('''
  const S1, S4

  S1 -> S2; k1*S1
  S2 -> S3; k2*S2
  S3 -> S4; k3*S3
  k1 = 0.1; k2 = 0.5; k3 = 0.001
  S1 = 5
''')
```

In the above model, `S1` and `S4` are fixed and will not change during the simulation. `S2` and `S3` will however change.

For small models one can use the dollar notation to fix species. This means placing a '$' symbol in front of each species you wish to be fixed. For example:

```
r = te.loada ('''
  $S1 -> S2; k1*S1
  S2 -> S3; k2*S2
  S3 -> $S4; k3*S3
  k1 = 0.1; k2 = 0.5; k3 = 0.001
  S1 = 5
''')
```

Exercises I

a) Build the following model:

```
  S1 -> S2; k1*S1
  S2 -> S1; k2*S2
  k1 = 0.1; k2 = 0.2
  S1 = 10
```

i. Is this system an open or closed system?

ii. Run a simulation from `t = 0` to `t = 50` and generate 100 simulation data points. Plot the results.

iii. What is the equilibrium constant for the reaction `S1 <-> S2`?

iv. Is the value of the equilibrium constant consistent with the simulation results? Don't just give a yes or no, say what it is about the simulation results that convinces you.

b) Build the following model:

```
S1 -> S2; k1*S1
S2 -> S3; k2*S2
k1 = 0.1; k2 = 0.2
S1 = 10
```

i. Is this system an open or closed system?

ii. Run a simulation from `t = 0` to `t = 50` and generate 100 simulation data points. Plot the results.

iii. Run the same simulation again from `t = 0` to `t = 200`. At the end of the simulation run the method: `r.getReactionRates()`, assuming `r` is the reference variable to your model. Explain the values that `getReactionRates` prints out. Do these results match your answer to b) i., if so why?

c) Fix the levels of `S1` and `S3` in the previous model, i.e make them boundary species. Read Basic Concepts II to learn how.

i. Is this an open or closed systems?

ii. Run a simulation from `t = 0` to `t = 50` and generate 100 simulation data points. Plot your results

iii. Explain the difference in results you get in the two cases b) ii. and c).

Basic Concepts II

Setting and Getting Values

Assume we have loaded the following model:

```
r = te.loada ('''
  S1 -> S2; k1*S1
  S2 -> S3; k2*S2
  S1 = 5
  k1 = 0.1; k2 = 0.2
''')
```

Let's say we wish to inspect the current values of the model. This is easily done by using the loaded model variable followed by the name of the value we wish to inspect. For example, to check the value of `k1` we would use:

```
print (r.k1)
```

or to check the value of `S1` we would use:

```
print (r.S1)
```

In an interactive environment would could also simply type `r.S1` and the value will be printed.

Setting values is just as easy, simple assign the new value to the variable of choice. For example to set `k1` to 5.6, we would use:

```
r.k1 = 5.6
```

This applies to any value in the model. For example to double the current value of `S1` we would use:

```
r.S1 = r.S1 * 2
```

Another way of getting and setting value is to use the `getValue` method. The advantage of this method is that it accepts a string representation of the value. For example to set `k1` to a new value using `setValue` we would use:

```
r.setValue ('k1', 4.5)
```

or to get the value we would use:

```
value = r.getValue ('k1')
```

Returning selected variables from a simulation

The method P`simulate` returns all the evolution of all the chemical species in a model with time as the first column. It is often the case that for large models especially, not every chemical species needs to be returned by simulate. As a result is it possible to specific the actual columns that will appear in the returns numpy array. For example is we have a model with four species, `S1`, `S2`, `S3` and `S4` only we wish only to return `S2` and `S3`, including time we would write:

```
m = r.simulate (0, 10, 20, ['time', 'S2', 'S3'])
```

The list in the square brackets is called a selection list. One trick you can do is not specify the time column so that a plot now represents a phase plot of the simulation. For example:

```
m = r.simulate (0, 10, 20, ['S2', 'S3'])
```

Named Reactions and Reaction Rates

Reaction can be named, for example, the two reactions in the following model are named `J1` and `AnotherReaction`.

```
  J1: S1 -> S2; k1*S1

  AnotherReaction: S2 -> S3; k2*S2
```

A name for a reaction must proceed the reaction followed by a colon. Named reactions are useful for allowing access to the reaction rate during a simulation. For example, to plot the two reactions rates for the following model we would use:

```
 r = te.loada('''
  J1: S1 -> S2; k1*S1
  J2: S2 -> S3; k2*S2
  k1 = 0.1; k2 = 0.2
  S1 = 10
  ''')
   m = r.simulate (0, 10, 100, [time, 'J1', 'J2'])
  r.plot()
```

Basic Model Metrics

There are many commands one can use to interrogate a model. Here are some common metrics that you might want to know. This is especially the case if your model was an SBML model where you need to find out more about the model.

1. Get the number of reactions in a model: `r.getNumReactions()`
2. Get the number of floating species in the model: `r.getNumFloatingSpecies()`
3. Get the number of boundary species in the model: `r.getNumBoundarySpecies()`
4. Get the number of parameters in the model: `r.getNumGlobalParameters()`

Exercises II

WARNING: Apparently a bug was introduced in the last release of tellurium that means you'll get a strange error if you run the exercise below, even though the results you get back are correct. If you want to avoid the strange error, update your version of roadrunner by running

```
!pip install libroadrunner --upgrade
```

from your console (note the two dashes, it's not a typo). This will upgrade the version that fixes this issue. If you have problems email Lucian at `lucianoelsmitho@gmail.com`

a) Given the following model:

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

    const S1, S4

    S1 -> S2; k1*S1/(1 + S3/0.01)
    S2 -> S3; k2*S2
    S3 -> S4; k3*S3
    k1 = 2; k2 = 0.24; k3 = 0.52
    S1 = 1; S2 = 0; S3 = 0; S4 = 0
''')
```

Run a simulation from `t = 0` to `t = 30` and generate 1000 data points of the simulation. Ensure that the first column of the data returned is `S2` and the second column is `S3`. This will ensure you get a plot of `S3` versus `S2`. Show the plot.

b) Load the following model:

```
    # feedback model

    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 initialization:
   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
```

i. Is this an open or closed system?

ii. The model includes a feedback loop. Can you identify the feedback signal species and which reaction is the signal controlling?

iii. Write a python script to generate a report in the form of a table summarising the number of reactions, floating species, boundary species and global parameters in the model. Use the python package **`tableprint`** to output a nice looking table.

iv. Run a simulation of the model from `t = 0` to `t = 50` and generate 1000 data points. Plot the simulation results.

v. What kind of dynamics do you observe in the simulation? In other words describe how the various concentrations are changing.

c) Using the following model:

```
   const S1, S4
```

```
S1 -> S2; k1*S1
S2 -> S3; k2*S2
S3 -> S4; k3*S3
k1 = 0.1; k2 = 0.2; k3 = 0.34
S1 = 10
```

i. Is this an open or closed system?

ii. Run a simulation from `t = 0` to `t = 40` generating 100 points. Next make a change to the parameter `k1` by reducing it by 75%. (Read Basic Concepts II to learn how to do that) Rerun the simulation from `t = 40` to `= 80` with one hundred points. Merge the two data sets together (Hint: `numpy.vstack`) to form a single simulation data set, called `combinedData` and use the following method to plot the combined data:

```
te.plotArray (combinedData)
```

iii. Jump to the section 'Advanced I' and read the section `Events`. Using what you learn in the events section, change the model to use an event. That is, at `t = 40`, decrease `k1` by 75%. Confirm you get the same results as you did in ii by showing a plot of the results.

Basic Concepts III

Using SBML with Tellurium

The standard exchange format for biochemical models is SBML. Model repositories such as BIOMODELS use SBML as the storage format for models. It is therefore important to be able to load SBML models and in turn be able to convert your own model into SBML for publication purposes or other reasons.

Let's begin by showing how to obtain an SBML representation of a model. Given the model:

```
r = te.loada ('''
S1 -> S2; k1*S1
S2 -> S3; k2*S2
S1 = 5
k1 = 0.1; k2 = 0.2
''')
```

The SBML for this model can be obtained by using the `getSBML` method:

`sbmlstr = r.getSBML()`

Since SBML is just a string, we can assign the SBML to a string variable. To save the SBML to a file we can use the utility method `saveToFile` that saves strings to files, for example:

`te.saveToFile ('mymodel.xml', r.getSBML())`

Note that the file extension is `xml`. This is because SBML is represented using the XML format.

The more traditional way in python to save a string to a file is to use the script:

```
with open("mymodel.xml", "w") as text_file:
    text_file.write(r.getSBML())
```

If you've made changes to the model parameters etc or you've just run a simulation and you want to save the current state of the model as SBML, then use the method:

```
sbmlStr = r.getCurrentSBML()
```

To load a SBML model from a file we can use the method `loadSBMLModel` which will accept a SBML string **or** a file containing the SBML. To load the previously saved SBML model we can use:

```
r = te.loadSBMLModel ('model.xml')
```

Retrieve the Antimony String of a Model

If you've loaded a model from an SBML file and you'd like to see the model in Antimony format you can use the following method:

```
r.getAntimony()
```

This returns a string of the model in antimony format. To make the string render neatly to the screen, use the print method:

```
print (r.getAntimony())
```

Saving Results To a File

A common operation may be the need to save the results of a simulation to a file which could be loaded into other tools for further analysis, for example Excel. The quickest way to save simulation results is to use the `savetxt` from the numpy library. For example, if `m` is the variable that holds the results of a simulation then we can save the data as a csv file using:

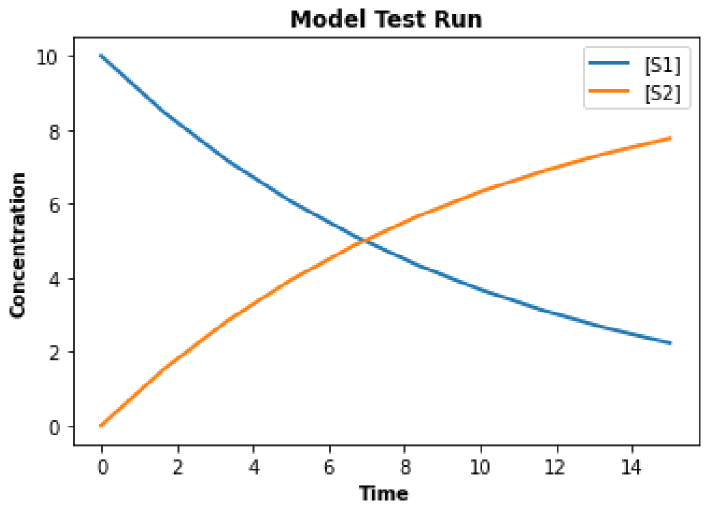
```
import numpy np
np.savetext ('data.csv', m, delimiter=',')
```

Extra Plot Options

The `r.plot()` method has a small number of options that can be used to customize a plot. For more flexibility one should use matplotlib (see later). For small things, such as handling axes titles etc, using `r.plot()` can be convenient. For example, the following call to plot will change various titles on the plot:

```
r.plot (xtitle='Time', ytitle='Concentration', title='Model Test Run')
```

This will result in:



A shortened list of additional options include:

Name	Function
xtitle	Set the x axis title
ytitle	Set the y axis title
linewidth	Set the line width of the plot lines, `linewidth=2`
title	Set the main title of the plot
xlim	Set the x axis limits eg `xlim=(0, 10)`
ylim	Set the x axis limits eg `ylim=(-5, 8)`
logx	Set log x axis, `logx=True`
logy	Set Set log y axis, `logy=True`
grid	Include a grid on the plot, `grid=True`
savefig	Save to plot to a file (pdf or png), `savefig='plot.pdf'`
figsize	Set the size of the plot (default inches) `figsize=(10,12)`

Exercises III

a) Load this simple model:

```
r = te.loada("""

   const S1, S4
```

```
    S1 -> S2; k1*S1/(1 + S3/0.01)
    S2 -> S3; k2*S2
    S3 -> S4; k3*S3
    k1 = 2; k2 = 0.24; k3 = 0.52
    S1 = 1; S2 = 0; S3 = 0; S4 = 0
 """)
```

b) Save it as an SBML model

c) Load the SBML model and run a simulation

d) Get the Antimony string for this SBML model and print it out to the screen.

Basic Concepts IV

Resetting a Model

An important operation that you are likely to use often is resetting a model or parts of a model. the most common operation is resetting the current floating species levels back to their initial state. This is achieved by calling the `reset` method:

```
r.reset()
```

If you have changed parameters and you want to reset the parameters back to their original values as well as the floating species concentrations, you can use the `resetAll` method:

```
r.resetAll()
```

If you wish to reset the entire model back to when the model was first loaded you can use the `resetToOrigin` method:

```
r.resetToOrigin()
```

Getting the Names of Variables and Parameters

To get a list of the names of the floating species we can use the method:

```
r.getFloatingSpeciesIds()
```

To get the list of names of the boundary species we can use the method:

```
r.getBoundarySpeciesIds()
```

To get the names of the global parameters in the model we can use the method:

```
r.getGlobalParameterIds()
```

Get the Values of Variables and Parameters

Another set of lists that are useful are the list of species and parameter values. To get the list of current values for the floating species we can use:

```
r.getFloatingSpeciesConcentrations()
```

Note that values are returned as numpy arrays.

To get the list of boundary species values use:

`r.getBoundarySpeciesConcentrations()`

Finally to get the list of parameter values we can use:

`r.getGlobalParameterValues()`

Note: In each case the order of the values returned by these methods will correspond to the order of names in the name lists.

Debugging Assistance

Debugging models can sometimes be difficult especially for very large models. To assist in debugging, three shortcut methods are provided as shown in the table below:

Short-cut	Function	Long version
r.dv()	Array of rates of change	r.getRatesOfChange()
r.rv()	Array of reaction rates	r.getReactionRates()
r.sv()	Array of floating species values	r.getFloatingSpeciesConcentrations()

Computing the Steady-State

Up to now we've only considered running time course simulation using `simulate`. However it is often the case that we will also want to know the steady-state of the model (if one exists). There are two approaches to this:

1) Run a time course simulation until all species no longer evolve in time;

2) Call the specific steady-state solver, `steadyState()`

Running a time course simulation will work but it has two issues, it could take a long time to reach steady-state and secondly, you need to be sure that the steady-state has been reached and it isn't just a slow evolution. If in doubt, run the steady-state method. When called, `steadyState()` returns a floating point value that represents how close the solution is to steady-state. Any value less than 1E-5 can be considered a successful steady-state determination.

Sometimes the model has no steady-state that can be computed. When this happens the software will issue a scary message such as:

```
Error:  Error :Jacobian matrix singular in NLEQ. Failed to converge to
steady-state.  Check if Jacobian matrix is non-invertible or steady-state
solution does not exist.
```

This usually happens when there is a problem with the model. For example one or model concentrations by be going negative or others are tending to infinity. When you get this error message, inspect for model for issues that might be the cause.

Advanced I

Additional Antimony Syntax

There are three other features of the Antimony modeling language that are useful.

Assignment Rules

Sometimes you might need to do additional calculations during a simulation. For example you might need to compute the pH as the model runs or some other derived quantity, you may wish to break up a big rate law into subexpressions to make them easier to mange, or you might wish to provide a specific input dynamic to a boundary species, for example a ramp. All these situations can be handled by **assignment rules**. Three examples ar given below:

Note: Unlike a normal assignment that uses a single equals symbol to set a value to a parameter or variable, such as `k1 = 1.2`; an assignment rule uses the symbol `:=`. This is an extremely important distinction and can be the source of errors in your model.

1. Derived quantities:

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

   ph := -log (H)

   H -> S2; k*H;
   k =  0.1; H = 0.3
''')

m = r.simulate (0, 40, 50, ['time', 'H', 'ph'])
```

2. Breaking up a large expression:

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

 t1 := Vm*S1
 t2 := Km + S1

 S1 -> S2; t1/t2
 k =  0.1; S1 = 0.3
''')
```

3. Applying time dependent signals to a model:

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

 S1 := time*k1 // ramp
 S3 := 3*sin (time*1) + 1 // sinusoid

 $S1 -> S2; k2*S1;
 $S3 -> S4; k3*S3
 k1 =  0.1; k2 = 0.05; k3 = 0.05
''')
```

Events

Events are step changes in a variable or parameter that occur during a model simulation. For example after a given time, a parameter might be doubled, or if a species concentration exceeds a given values a series of rate constants should be set to zero. Such cases are handled by **events**.

Two cases are given below

1. Change a parameter after a given time

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

   S1 -> S2; k1*S1;
   S2 -> S3; k2*S2;
   k1 = 0.1; k2 = 0.24
   S1 = 10

   at time > 8: k1 = 0
''')

m = r.simulate (0, 40, 100)
```

Events are specified using the `at` keyword. In general the syntax is given by:

`at <condition>:  <assignment, etc>`.

For example:

`at time > 1.2:  k4 = k4 * 1.75`

The condition is expressed as a Boolean formula, and the desired changes are expressed as assignments. This `at time > 8:  k1 = 0`, means when time exceeds 8, set `k1` to 0. You can have as many assignments as you wish for a given event.

1. Change two parameters if a species exceeds a certain value

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

   S1 -> S2; k1*S1;
   S2 -> S3; k2*S2;
   k1 = 0.1; k2 = 0.24
   S1 = 10

   at S3 > 3: k1 = 0, k2 = 0
''')
```

Note: One thing to be aware of when using events. Events are only executed once when a condition is satisfied. Thus the event `at time > 4:  k1 = 1` is only executed `once` when time is greater than 4. It is not executed again even though time continues to be greater than 4.

Exercises V

Run these two examples to observe the effect of the events.

Stability Analysis

The stability of biochemical models is an important consideration for systems that exhibit bistability or oscillations. The key metric for assessing stability is to look at the eigenvalues of the Jacobian matrix. There are two methods of interest: `r.getFullJacobian()` and `r.getFullEigenValues()`. Normally one would only need to call the `r.getFullEigenValues()` method since it computes the eigenvalues from the Jacobian.

Note: the Jacobian and eigenvalues are computed at the current state of the model. The model could be undergoing a transition or be at steady-state. Clearly if one is interested in steady-state stability, the model should be put into steady-state first.

In the following example, the steady-state of the model is determined then the eigenvalues are computed and printed out.

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

   S1 -> S2; k1*S1;
   S2 -> S3; k2*S2;
   k1 = 0.1; k2 = 0.24
   S1 = 10
''')

r.steadyState()
print (r.getFullEigenValues())
```

Advanced II

One Step

Up to now we've been using simulate to do time course simulations. What if we wish to run a simulation in steps and at each step we wanted to carry out additional computations? We could do this by running simulate with short time lengths. This is possible but potentially inefficient. Instead there is a method called `oneStep()` which carries out literally one step of the integration process. `oneStep()` takes two arguments: the current time and the requested time step. As a convenience the method returns the new time. For example:

```
t = r.oneStep (t, 0.1)
```

This will advance the simulation by 0.1, returns to t the value `t + 0.01`. oneStep is mostly used inside loops, for example:

```
t = 0
for i in range (20):
   t = r.oneStep (t, 0.1)
   # Do something
```

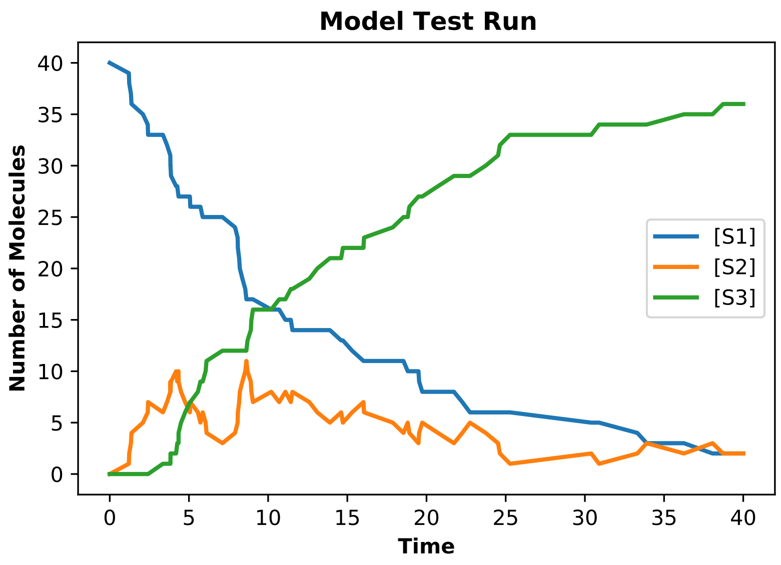
For example, instead of specifying events in the antimony model, it would be possible to test for conditions inside the loop. Such an example is shown below:

```
t = 0
for i in range (20):
   t = r.oneStep (t, 0.1)
   if t > 20:
      r.k1 = 0
      r.k2 = 0
```

Stochastic Simulations

Up to now we've only considered deterministic simulations but it is very easy to also carry out stochastic simulations. The only change is to call `r.gillespie ()` instead of `r.simulate()`. For example:

```
r = te.loada('''
   S1 -> S2; k1*S1;
   S2 -> S3; k2*S2;
   k1 = 0.1; k2 = 0.24
   S1 = 40
''')

m = r.gillespie(0, 40)
r.plot (xtitle='Time', ytitle='Number of Molecules',
      title='Model Test Run', savefig="stochastic.pdf")
```



Advanced III

More Advanced Plotting using Matplotlib

By combining simulation with matplotlib one can do a large variety of visualization including animation. We can start by using matplotlib directly to implement the equivalent of `r.plot()`.

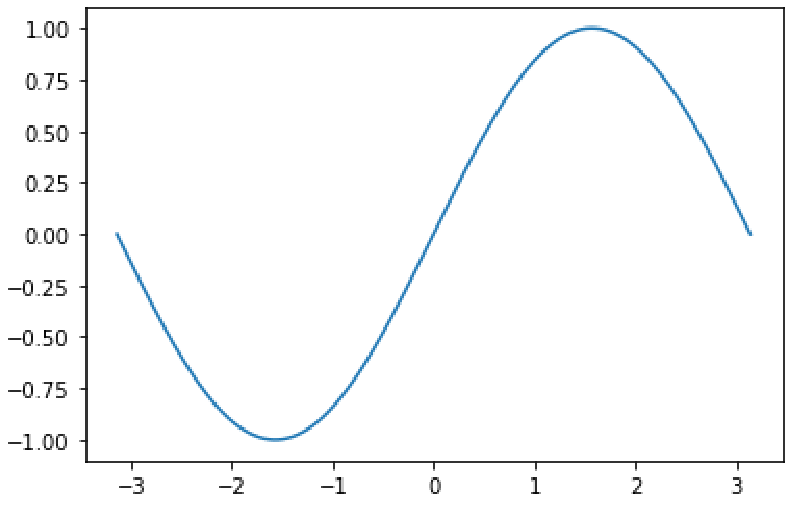
Importing matplotlib requires the unwieldy statement:

`import matplotlib.pyplot at plt`

This allows us to use `plt` in subsequent calls to matplotlib and is fairly standard practice. The method `plot` in matplotlib, takes two arguments, `x` data and `y` data. For example consider this:

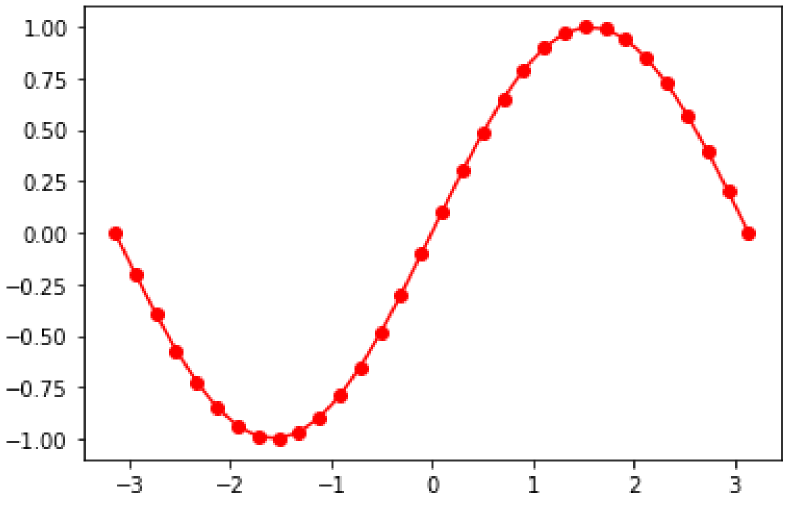
```
import numpy as np

x = np.linspace(-np.pi, np.pi, 256)
y = np.sin(x)
https://www.overleaf.com/project/5fff8651ec31f83c80ddea2e
plt.plot (x, y)
plt.show()
```

This yields:



By default, `plot` draws a line graph in blue. These settings can be easily changed with additional arguments. To just plot the points, say in red, we would include the argument 'ro', where the r represent red and the o represents a round filled marker. To draw a line and a marker, simply add a dash, as in 'ro-'. This yields:

```
import numpy as np

x = np.linspace(-np.pi, np.pi, 32)
y = np.sin(x)

plt.plot (x, y, 'ro')
```



Notice however that the line and marker are both in red. If you wish to have different colors for the marker and line you can call plot twice, one for the line and another for the markers:

```
import numpy as np

x = np.linspace(-np.pi, np.pi, 32)
y = np.sin(x)

plt.plot (x, y, 'g')
plt.plot (x, y, 'ro')
```

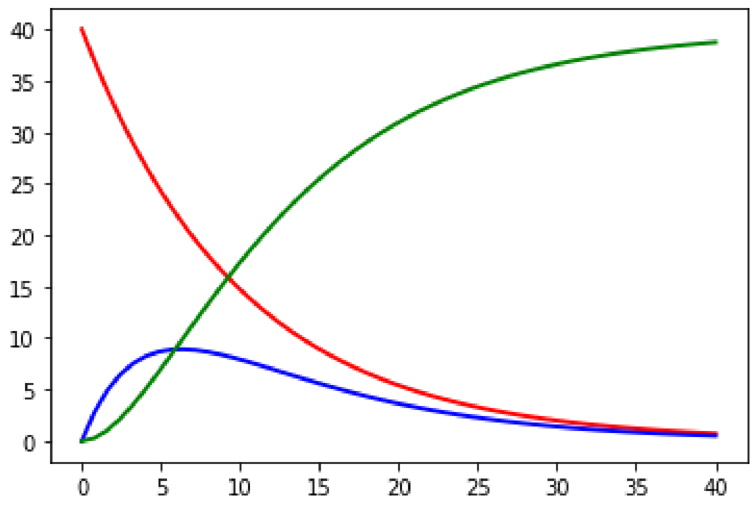
In this case the first plot draw the line in green and the second plot draws red markers. Plots in succession like this will be overlayed until a `plt.show()` is called.

The array of results returned from a simulation have the columns labelled with the specific variable. Thus the first column will be time and is labelled `time`. We can exploit this when using plot by using the named columns to pull out specific columns. For example if `m` has three columns, time, S1 and S2, we can plot time versus S2 by using:

```
plt.plot (m['time'], m['[S1]'])
```

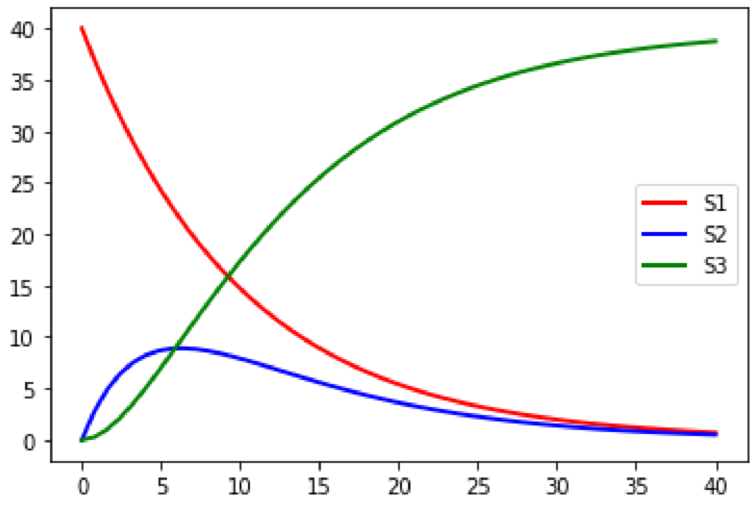
Note the square brackets round the name of the species, this is to indicate concentration. The following example shows how to plot multiple time courses with specified colors for each line

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

r = te.loada("""

   S1 -> S2; k1*S1;
   S2 -> S3; k2*S2;
   k1 = 0.1; k2 = 0.24
   S1 = 40
""")

m = r.simulate(0, 40)
plt.plot (m['time'], m['[S1]'], 'r', linewidth=2)
plt.plot (m['time'], m['[S2]'], 'b', linewidth=2)
plt.plot (m['time'], m['[S3]'], 'g', linewidth=2)
plt.show()
```



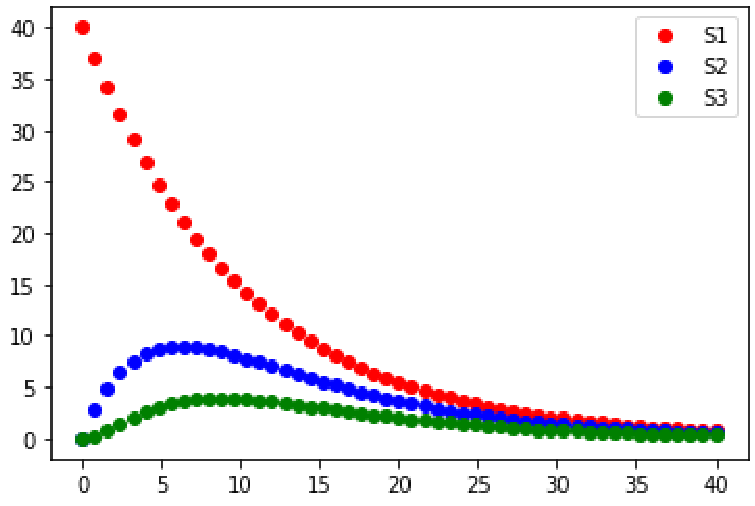
If we want to add a legend we need to specific the legend labels as in the following and call `plt.legend()` to display the legend:

```
m = r.simulate(0, 40)
plt.plot (m['time'], m['[S1]'], 'r', linewidth=2, label="S1")
plt.plot (m['time'], m['[S2]'], 'b', linewidth=2, label="S2")
plt.plot (m['time'], m['[S3]'], 'g', linewidth=2, label="S3")
plt.legend()
plt.show()
```



Finally if we want to see the simulated points we can use:

```
m = r.simulate(0, 40)
plt.plot (m['time'], m['[S1]'], 'ro', linewidth=2, label="S1")
plt.plot (m['time'], m['[S2]'], 'bo', linewidth=2, label="S2")
plt.plot (m['time'], m['[S3]'], 'go', linewidth=2, label="S3")
plt.legend()
plt.show()
```

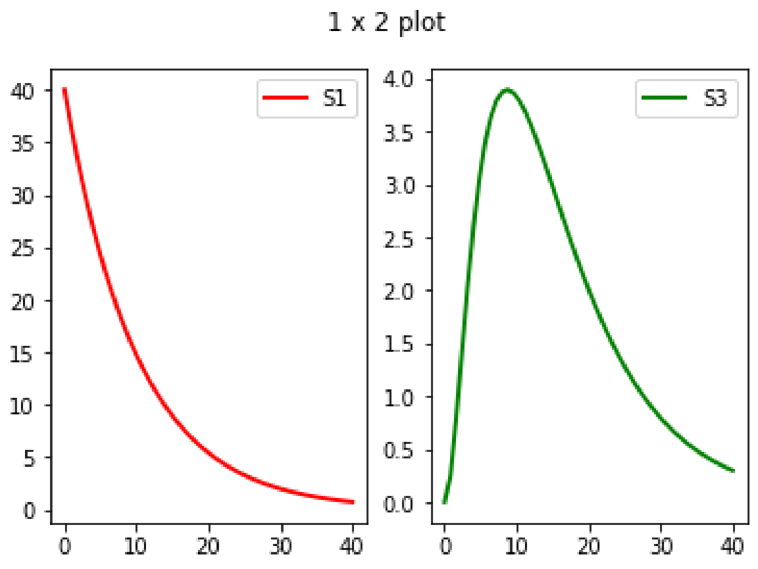


Plots Side by Side

What about plots that are side-by-side. This is possible using the subplot methods. The first example shows a side by side plot:

```
fig, axs = plt.subplots(1, 2,figsize=(5,4))
fig.suptitle('Vertically stacked subplots')
axs[0].plot(m['time'], m['[S1]'], 'r', linewidth=2, label="S1")
axs[1].plot(m['time'], m['[S3]'], 'g', linewidth=2, label="S3")
axs[0].legend(); axs[1].legend()
```

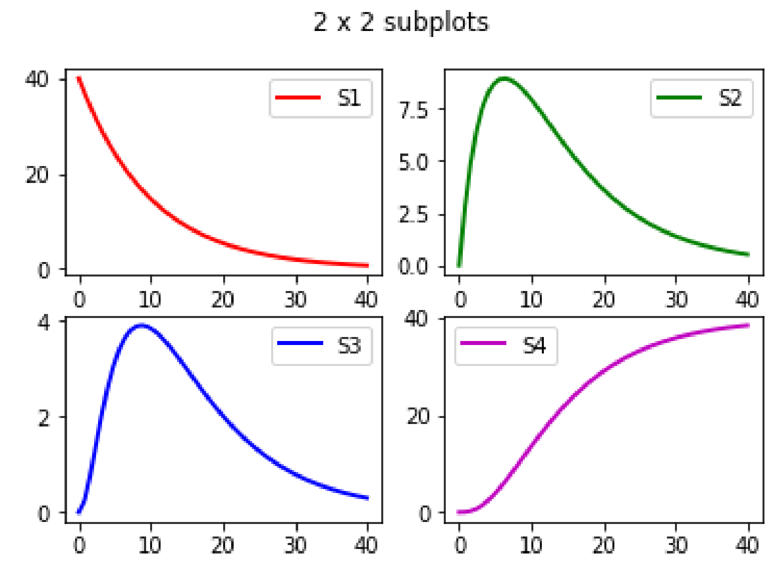
The above example also shows how to change the size of the subplot group.



The following example shows 2 by 2 matrix of plots.

```
r = te.loada("""

   S1 -> S2; k1*S1;
   S2 -> S3; k2*S2;
   S3 -> S4; k3*S3;
   k1 = 0.1; k2 = 0.24; k3 = 0.52
   S1 = 40
""")

m = r.simulate (0, 40)

fig, axs = plt.subplots(2, 2)
fig.figsize(12,10)
fig.suptitle('2 x 2 subplots')
axs[0,0].plot(m['time'], m['[S1]'], 'r', linewidth=2, label="S1")
axs[0,1].plot(m['time'], m['[S2]'], 'g', linewidth=2, label="S2")
axs[1,0].plot(m['time'], m['[S3]'], 'b', linewidth=2, label="S3")
axs[1,1].plot(m['time'], m['[S4]'], 'm', linewidth=2, label="S4")
axs[0,0].legend(); axs[0,1].legend()
```

```
axs[1,0].legend(); axs[1,1].legend()
plt.show()
```


2 x 2 subplots
S1
S2
S3
S4