# A quick guide to Tellurium and libRoadRunner[1]

## What is libRoadRunner?

libRoadRunner is a C/C++ software library that supports the simulation and analysis of biochemical pathways. Models are read into libRoadRunner in the Systems Biology Markup Language (SBML) format and can be modeled either using differential equations or stochastic approaches. libRoadRunner can be accessed through a number of APIs, including a C API, a C++ API and a Python API. A simple standalone application also exists that enables a user to run simulations from the command line. libRoadRunner is available on Windows, Mac and Linux. See `tellurium.analogmachine.org` and `libroadrunner.org`.

## Special Features of libRoadRunner

libRoadRunner has a number of unique features: 1) The API has been designed for modelers with particular emphasis on ease of use; 2) libRoadRunner compiles model using LLVM, this allows us to achieve maximum performance.

## What is Tellurium?

Tellurium is an integrated platform based on Python that includes libRoadRunner as one of its supporting libraries. In addition to libRoadRunner, Tellurium comes with spyder2 as the cross-platform IDE (giving a Matlab like experience), Antimony, that allows user to write models in a more human readable form, SBML2Matlab, that allows user to export models in Matlab format, and libSBML that allow users detailed access to model particulars. In addition Tellurium comes preloaded with the Python plotting library Matplotlib and the array package numpy. Tellurium also comes with a small number of helper subroutines to make it easier for the average modeler.

## SBML

The Systems Biology Markup Language (SBML) is a representation format, based on XML, for communicating and storing computational models of biological processes. It is a free and open standard with widespread software support. SBML can represent many different classes of biological phenomena, including metabolic networks, cell signaling pathways, regulatory networks, infectious diseases, and many others. As an XML format, SBML is not meant to be read or written by Humans.

## Antimony

The Antimony language provides a way for researchers to use simple text statements to create, import, and combine biological models, allowing complex models to be built from simpler models. It is fully compatible with SBML such that SBML and Antimony can be converted from one to the other.

[1] Version 1.04

### Examples of Antimony Models

```
1.
# Simple Decay process
S1 -> S2; k1*S1;

k1 = 0.1; S1 = 10; S2 = 0
```

```
2.
# Consecutive reactions
S1 -> S2; k1*S1;
S2 -> S3; k2*S2;
k1 = 0.1; k2 = 0.2;
S1 = 10; S2 = 0; S3 = 0;
```

```
3.
# Bimolecular reactions
S1 + S2 -> S3;  k1*S1*S2;
S3 -> S1 + S2;  k2*S3;
# Branched System
S1 -> S2; k1*S1;
S2 -> S3; k2*S2;
S2 -> S4; k3*S2;
```

```
4.
# Open system with fixed boundaries
# '$' indicates a fixed species
$S1 -> S2;  k1*S1;
 S2 -> S3;  k2*S2;
 S3 -> $S4; k3*S3;
```

```
5.
# Open system with empty boundaries
   -> S2;  k1*S1;
 S2 -> S3; k2*S2;
 S3 -> ;   k3*S3;
```

```
6.
# Simple Feedback System
   -> S2;  k1*S1/(k2 + S1 + S3/Ki);
 S2 -> S3; k2*S2 - k3*S3;
 S3 -> ;   k4*S3;
```

```
7.
# Named reactions and events
J1: $S1 -> S2;  k1*S1;
J2:  S2 -> $S3; k2*S2;
k1 = 0.1; k2 = 0.3;
S1 = 10;

at (time > 10): k2 = 0.6
```

```
8.
# Modeling Gene expression

# Modeling an activator P0 that
# results in expression of P1
G1:   -> P1;  Vm1*P0^n/(K1 + P0^n);

# Modeling a repressor P0
J2:  S2 -> S3;  Vm1/(K1 + P0^n);
```

### Loading SBML and Antimony Models

The easiest way to load a SBML model is to use the `loada` function:

```
# Load the model from a SBML file
r = te.loada ('mymodel.xml')
# Load a model from an antimony string
r = te.loada (antimonyString)
```

`loada` is simply a short-cut for `loadAntimonyModel`. The methods return a reference to a copy of libRoadRunner.

## Simulation

libRoadRunner supports two kinds of simulation, differential equation based and stochastic based simulations. After a model has been loaded into libRoadRunner, the user has the option to either carry out a deterministic simulation based on solving differential equations, or a stochastic simulation based on the Gillespie Algorithm.

### Differential Equation Based Simulations

To simulate model based on differential equations use the `simulate` command:

```
result = r.simulate (0, 10, 100)
```

In its basic form, simulate takes three arguments, **time start**, **time end** and the **number of points** to generate. If the arguments are omitted default values are chosen. The simulate command returns an array that contains the results of the simulation. The first column holds data that represents the time axis, all subsequent columns represent the floating species in the model. The particular columns in the result array can be changed with a selection list, for example:

```
result = r.simulate (0, 10, 100, ['Time', 'S1', 'k1'])
```

means that the first column will hold time, the second column S1 and the third column k1.

### Stochastic Based Simulations

To simulate a model based on Gillespie stochastic method use the `gillespie` command:

```
  result = r.gillespie (0, 10)
```

In its basic form, gillespie takes two arguments, time start, and time end. If the arguments are omitted default values are chosen. The gillespie command returns an array that contains the results of the simulation. The first column hold data that represents the time axis, all subsequent columns represent the floating species in the model. The particular columns in the result array can be changed with a selection list, for example:

```
  result = r.gillespie (0, 10, 50, ['Time', 'S1', 'S4'])
```

means that the first column will hold time, the second column S1 and the third column S4. The Gillespie method by its nature returns values in the time column at irregular intervals. We provide a variant where the output is returned on a regular time grid which is why the 50 is present in the call above.

## Changing Values

In a simulation it is often necessary to make changes to values in a model and rerun the simulation. Consider the following code:

```
  r = te.loada ('''
      S1 -> S2; k1*S1;
      S2 -> S3; k2*S2;
      S3 -> S4; k3*S3;
      k1 = 0.1; k2 = 0.2; k3 = 0.3;
      S1 = 10
  ''')

  # Modify the k1 rate constant
  r.k1 = 12.0

  # Changing species levels
  r.S1 = 24.5
```

## Plotting

A common need is to be able to plot simulation results. To do this we use the `plot` command

```
  import tellurium as te
  r = te.loada ('''
      S1 -> S2; k1*S1;
      S2 -> S3; k2*S2;
      S3 -> S4; k3*S3;
      k1 = 0.1; k2 = 0.2; k3 = 0.3;
      S1 = 10
  ''')
  result = r.simulate (0, 50, 50)
  r.plot ()
  # Or:
  r.plot(ytitle='Concentration', xtitle='Time',
        title='A simple simulation',
        xlim=[0, 50], ylim=[0, 20])
```

For more complex plots use the Python library matplotlib directly.

```
import tellurium as te, pylab

r = te.loada ('''
      S1 -> S2; k1*S1;
      k1 = 0.1; S1 = 10
''')

result = r.simulate (0, 50, 50)
pylab.ylabel ('Concentration')
pylab.xlabel ('Time')
pylab.plot (result[:,0], result[:,1], linewidth=2)
pylab.plot (result[:,0], result[:,2], linewidth=2)
```

## Resetting the Model

A common operation when doing interactive simulation is resetting a model back to some initial state.

```
   # Reset a model back to the state it was
   # when it was first loaded or created
   r.resetToOrigin()

   # Reset the current species values back
   # to their initial conditions
   r.reset()

   # Reset the current species values back
   # to their initial conditions and reset
   # all parameters back to when the model was
   # first loaded or created
   r.resetAll()
```

## Exporting a Model

Models can be exported in threes different formats: SBML, Antimony or Matlab

```
  # Export the model as SBML
  print r.getCurrentSBML()

  # Export the model as Matlab
  print r.getMatlab()

  # Export the mode as Antimony
  print r.getAntimony()
```

## Computing the Steady State

To compute the steady state for a model use:

```
  r.steadyState()
  print r.getFloatingSpeciesConcentrations()
  # Short-cut:
  print r.sv()  # Species vector
```

The steadyState function returns a value indicating how close the the solution is to the steady state. The smaller the value the better. Values less than $10^{-4}$ usually indicate that the steady state was found.

## Useful Matrices

There are a variety of matrices that can be obtained from the model, only two will be described here.
The stoichiometry matrix:

```
  print r.getFullStoichiometryMatrix()
  # Short-cut:
  print r.sm()
```

The Jacobian matrix:

```
  print r.getFullJacobian()
  # Short-cut:
  print r.fjac()
```

## Useful Vectors

The reaction rate vector:

```
  print r.getReactionRates()
  # Short-cut:
  print r.rv()
```

The rates of change vector:

```
  print r.getRatesOfChange()
  # Short-cut:
  print r.dv()
```

The species concentration vector:

```
  print r.getFloatingSpeciesConcentrations()
  # Short-cut:
  print r.sv()
```

The names of all floating species

```
  print r.getFloatingSpeciesIds()
  # Short-cut:
  print r.fs()
```

The names of all kinetic parameters in the model

```
  print r.getGlobalParameterIds
  # Short-cut:
  print r.ps()
```

## Resources

https://tellurium.readthedocs.io/en/latest/index.html
http://libroadrunner.org/
http://antimony.sourceforge.net/
http://tellurium.analogmachine.org/

1. Tellurium: An extensible python-based modeling environment for systems and synthetic biology, Biosystems, 171, 74-79 (2018) Choi et al.
https://doi.org/10.1016/j.biosystems.2018.07.006
2. Tellurium notebooksAn environment for reproducible dynamical modeling in systems biology, PLoS Comp Bio, June 2018, https://doi.org/10.1371/journal.pcbi.1006220J, Kyle et al