COMBINE & de.NBI Tutorial: Modelling and Simulation Tools in Systems Biology

Kinetic Modeling with Tellurium Tutorial/ Challenges

Author/ tutor: Veronica L. Porubsky (verosky@uw.edu), PhD student at University of Washington

Resources:

Tellurium introduction (Jupyter notebook on Google Colaboratory)

Solutions to the Tellurium tutorial/ challenges (Jupyter notebook on Google Colaboratory)

GitHub repository for the tutorial

Note 1: The challenges and tips in this tutorial assume you have imported the following:

Tellurium: `import tellurium as te`
Numpy: `import numpy as np`
Pyplot: `import matplotlib.pyplot as plt`

Note 2: Tellurium uses RoadRunner object instances to hold the executable models obtained by loading an SBML or Antimony string with the `te.loadSBMLModel(SBML_string)` or `te.loada(antimony_string)` commands. In the challenges and tips below, the example RoadRunner object name will be named `model`, such that simulations can be executed and utilities can be accessed from the object: ie. `model.simulate()`, `model.gillespie()`, `model.plot()`, `model.getFloatingSpeciesIds()`, etc.

BASIC UTILITIES

Challenge 1: Create an Antimony string for the following reaction network:

$$\begin{tikzcd} S1 \arrow[r, "\nu_1"] & S2 \arrow[r, "\nu_2"] & S3 \end{tikzcd}$$

$$\nu_1 = k_1 \cdot [S1]$$
$$\nu_2 = k_2 \cdot [S2]$$

$$k_1 = 0.15, \quad k_2 = 0.45$$

$$\text{at time} = 0: \quad [S1] = 1$$
$$[S2] = 0$$
$$[S3] = 0$$

Tips:

- Antimony strings can be saved to a variable like any string in Python
- Antimony strings should be specified within three quotations, ie. **`antimony_string =  ‘’’…‘’’`**
- Each reaction in the Antimony string should take the form:

```
reaction_name: reactants → products; rate law;
```

- Initialize species concentrations and assign global parameter values in the string

Challenge 2: Create a RoadRunner object instance with your Antimony string and simulate your model from time = 0 to time = 20 with 50 data points.

Tips:

- Use **`te.loada(antimony_string)`** to load your Antimony string and create an executable model (saved as a RoadRunner object instance)
- Assign your RoadRunner object with a memorable name:
  `model = te.loada(antimony_string_variable_name)`
- Simulate your model with:
  `result = model.simulate(time_start, time_end, number_of_points)`

Challenge 3: Plot the simulation with a title and x and y axis labels.

Tips:

- Use **`model.plot()`** to plot the simulation results from the previous challenge
- Type **`?model.plot()`** to access the Python Help documentation for the function, review input arguments and determine how to add title and axis labels
- You could also use the matplotlib.pyplot package – simply index the numpy array saved in **`result`** to plot time vs. the species concentrations.

Challenge 4: Reset your model, simulate, and plot only the species S2 concentration time-course.

Tips:

- You can specify “selections” within the **`model.simulate()`** command to determine which simulation data should be saved for plotting
- If you wish to return the time column, you must specify **`‘time’`** and **`‘S2’`** in your selections list
- Use **`model.selections()`** to investigate what species you are recording
- You could also use matplotlib.pyplot and index your simulation results to plot a single time-course from the default, **`result = model.simulate(time_start, time_end, number_of_points)`** with no selections specified, which returns all the floating (or variable) species concentrations for the time-course
- If you use matplotlib.pyplot, you can index the data saved in **`result`** using the column headings, ie. **`‘S2’`**

Challenge 5: Add an event to your Antimony string to set k2 = 0.01 after 10 seconds of simulation time. Load the string to create a RoadRunner instance, simulate the model and plot the time-course of S2 concentration and k2 values throughout the simulation.

Tips:

- You can specify **`‘k2’`** as one of your selections within the **`model.simulate()`** command to record the value of the parameter at each time step along with the **`‘time’`** and **`‘S2’`** selections

Challenge 6: Perform and plot a parameter scan of 'k1' from k1 = 0.1 to k1 = 1.0 in steps of 0.15, using your model from Challenge 1 (which does not include events). Plot the simulation trajectory of [S2] only, and ensure that all the simulation trajectories for the scan are displayed on a single plot.

Tips:

- To set the value of parameters in your model without manually editing the Antimony string and reloading to a RoadRunner instance use **`model.k1 = 0.1`**
- Be sure to reset your floating species concentrations on each iteration through the parameter scan using **`model.reset()`**
- Instead of **`model.plot()`**, try using **`te.plot()`** to refrain from displaying separate plots for each simulation – investigate the input arguments to do this
- After you have finished the scan, show the simulation trajectories with **`te.show()`**

Challenge 7: Set the initial conditions of your model to: [S1] = 20, [S2] = 10, [S3] = 10. Perform and plot a stochastic simulation of your model. Next, produce a distribution of stochastic simulation trajectories.

Tips:

- Use **`model.gillespie(start_time, end_time, number_of_points)`** to perform a stochastic simulation
- The Gillespie algorithm will randomly select which reaction to execute at randomly selected time intervals over the course of your simulation. Therefore, you may want to increase the initial concentrations of your species, to ensure there are sufficient amounts of the reactants in each reaction to see interesting behavior.

Antimony → SBML, phraSEDML → SEDML: IMPORT, EXPORT, CONVERSION

Challenge 8: Export an SBML file of your model from the previous challenges to your current working directory. Use the initial model settings from Challenge 1. Then import and load the SBML file you have just saved, simulate using the default deterministic integrator (CVODE), and ensure your simulation results match those in Challenge 3.

Saving your models as declarative SBML files will allow you to exchange your model between simulation, validation, and visualization platforms. To demonstrate the exchangeability of the SBML standard, launch this simulator in your browser and locate and drag your saved SBML file into the simulator to load and simulate the model.

Tips:

- Use **`model.exportToSBML()`** to export an SBML file – specify the path for export
- **`model.getCurrentSBML()`** can be used to obtain the SBML string, which can be written to a file

Challenge 9: Use the PhraSED-ML documentation to write a PhraSED-ML string which specifies a simple simulation experiment in which the reactions in your model from Challenge 1 are numerically integrated (using the default integrator) from time = 0 to time = 20 and plots only species [S1] and [S3]. Convert the PhraSED-ML to SED-ML, execute, and export the SED-ML to your current working directory. Execute the SED-ML script using the file which you exported.

Tips:

- Use **`sedml_str = phrasedml.convertString(phrasedml_str)`** to obtain the SED-ML string
- Use **`te.executeSEDML(sedml_str)`** to run the simulation experiment
- Use **`te.saveToFile()`** to save the SED-ML string as a .xml file for exchange across platforms

PARAMETER FITTING AND UNCERTAINTY QUANTIFICATION

Challenge 10: Download the data (saved as a numpy array file) simple_model_training_data.npy on GitHub to your working directory. These data represent noisy experimental time-course results for [S1], [S2], [S3] in the system you are modeling from Challenge 1. The four columns in the data correspond to 'time', '[S1]', '[S2]', and '[S3]', respectively.

Plot the experimental data.

Tips:

- Once you have saved this file, you can load the array to a variable with:
  `training_data = np.load('simple_model_training_data.npy')`

Challenge 11: Use all the species experimental data from Challenge 8 to perform parameter fitting with your model from Challenge 1 and determine values for k1 and k2. Report your optimized values for k1 and k2.

Tips:

- You can import any Python optimization library for the parameter fitting. Here are a couple suggestions:
  - scipy.optimize package
    - **`import scipy.optimize`**
    - You can import a specific optimizer:
      - Ex. **`from scipy.optimize import differential_evolution`**
    - The differential_evolution method is an effective global optimizer which can be used to minimize a scalar objective function. Try using this if you are less familiar with other optimization methods
  - lmfit package
    - **`import lmfit`**
    - The default optimization method is Levenberg-Marquardt
    - Make the objective function return the residuals, or the difference between your model simulation results and the experimental result for each timepoint
- Write an objective function for the optimization algorithm which will minimize the deviations between your model and the experimental data – consider residuals or root mean square error

Challenge 12: Determine which single species time-course is most important to fit the parameter values effectively. The ground truth values are: k1 = 0.55 and k2 = 0.15.

Tips:

- Ensure that your objective function from Challenge 9 allows you to change which species data are used during optimization, and perform fitting using only a single species at a time

Challenge 13: Plot the uncertainty in species S1, S2, and S3 concentrations in response to changes in the parameters, k1 and k2.

Tips:

- Review the Python Help documentation for uncertainty quantification in Tellurium:
  - **`?te.utils.uncertainty.UncertaintyAllP()`**
  - **`?te.utils.uncertainty.UncertaintySingleP()`**

EXAMINING MODELS FROM THE BIOMODELS DATABASE

Challenge 14: Load, simulate, and plot the simulation results of the Kholodenko 2000 model of ultrasensitivity and negative feedback oscillations in the MAPK cascade from the BioModels Database.

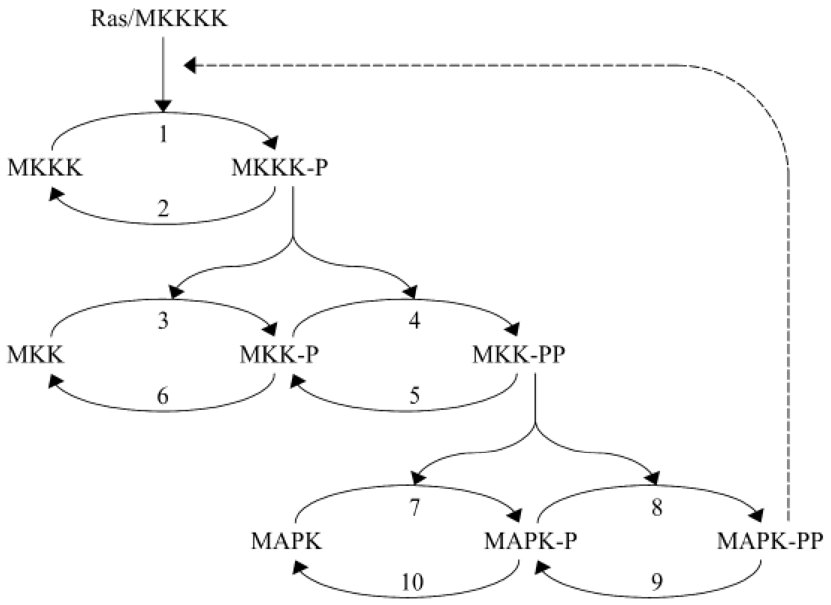
Tips:

- The BioModel ID for this model is: BIOMD0000000010

Challenge 15: Try to compute the steady state of the Kholodenko 2000 model.

You will get an error. What does this error message suggest as the reason the steady state could not be computed? Examine the Jacobian matrix.

Read about the steady state solver in libRoadRunner's documentation and examine the network diagram for the model to diagnose the problem:



Based on the network diagram and RoadRunner documentation linked above, adjust the settings in RoadRunner to be able to compute the steady state without raising an error.

Tips:

- Use **`model.steadyState()`** to use an adapted Newton method to compute the steady state.
- Use **`model.getFullJacobian()`** to investigate the Jacobian matrix

References:

[1] K. Choi *et al.*, “Tellurium: An extensible python-based modeling environment for systems and synthetic biology,” *Biosystems*, vol. 171, pp. 74–79, Sep. 2018.

[2] E. T. Somogyi *et al.*, “libRoadRunner: a high performance SBML simulation and analysis library.,” *Bioinformatics*, vol. 31, no. 20, pp. 3315–21, Oct. 2015.

[3] L. P. Smith, F. T. Bergmann, D. Chandran, and H. M. Sauro, “Antimony: a modular model definition language,” *Bioinformatics*, vol. 25, no. 18, pp. 2452–2454, Sep. 2009.

[4] K. Choi, L. P. Smith, J. K. Medley, and H. M. Sauro, “phraSED-ML: a paraphrased, human-readable adaptation of SED-ML,” *J. Bioinform. Comput. Biol.*, vol. 14, no. 06, p. 1650035, Dec. 2016.

[5] B. N. Kholodenko, “Negative feedback and ultrasensitivity can bring about oscillations in the mitogen-activated protein kinase cascades,” *Eur. J. Biochem.*, vol. 267, no. 6, pp. 1583–1588, Mar. 2000.

[6] J. Wolf, H. Y. Sohn, R. Heinrich, and H. Kuriyama, “Mathematical analysis of a mechanism for autonomous metabolic oscillations in continuous culture of Saccharomyces cerevisiae,” *FEBS Lett.*, vol. 499, no. 3, pp. 230–234, Jun. 2001.