Tellurium & libRoadRunner

COMBINE & de.NBI Tutorial: Modeling and Simulation Tools in Systems Biology
20th International Conference on Systems Biology
OIST, Okinawa, Japan
October 30th, 2019

Veronica L. Porubsky, B.S.
Sauro Lab PhD student - Department of Bioengineering
University of Washington, Seattle, WA, USA
Outreach Coordinator - Center for Reproducible Biomedical Modeling

Contents

Tellurium tutorial solution manual useful resources

Alternate tutorial access

Jupyter notebooks of tutorial + challenges: Github Repository
Tellurium Introduction: Google Colaboratory notebook
Tellurium Challenges: Google Colaboratory notebook

Installation and documentation

Tellurium installation instructions
Tellurium documentation [1]
libRoadRunner documentation [2]
Antimony documentation [3]
PhraSED-ML documentation [4]
Tellurium environment on nanoHUB

Note: Tellurium and libRoadRunner do not currently support Python 3.7. To use a Python 3.6 distribution without altering the distribution on your local machine, download the Tellurium Spyder IDE, notebook, use nanoHUB, or run this Jupyter notebook on Google Colaboratory.

Introduction to kinetic modeling

What is kinetic modeling?

- Chemical kinetics studies the factors that influence the rate of chemical reactions
  - Concentration
  - Temperature
  - Light
  - Catalysts
- Chemical reaction networks are the framework for building all types of dynamical models
  - Genetic circuits
  - Cell signaling pathways
  - Metabolic networks
- Types of kinetic models:
  - Agent-based
  - Algebraic
  - Boolean
  - Constraint based
  - Differential equations
  - Statistical and machine learning methods
  - Stochastic

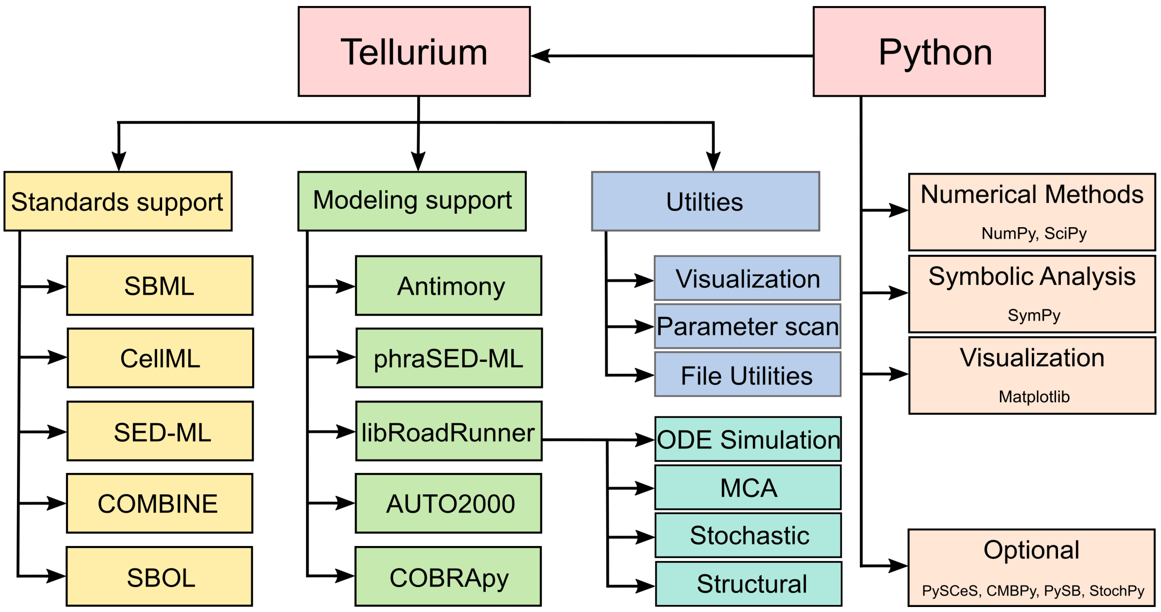
How does numerical simulation help us model kinetics?

- Provides a method to approximate analytical solutions for complex (often non-linear) systems
- Kinetic laws describe the rates of change of species in the system, which can be modeled mathematically
- For a sufficiently large network, simulators must be efficient and perform rapid numerical integration

What are differential equations models?

- Models which describe how variables in a system evolve over time
  - e.g. floating species concentrations
- Quantities are derived from the variables
  - e.g. pathway flux
- Some parameters of the model are fixed by the modeler
  - e.g. rate constants
  - e.g. enzyme concentrations
  - e.g. boundary species concentrations
- Deterministic differential equations models are useful when we can assume there are a large number of participants in the chemical reactions
- Stochastic models are useful for dilute systems in which reactions may not occur at every timepoint

Tellurium and libRoadRunner support rapid simulation and analysis of kinetic models



Introduction to modeling with Tellurium

Installing Tellurium with pip

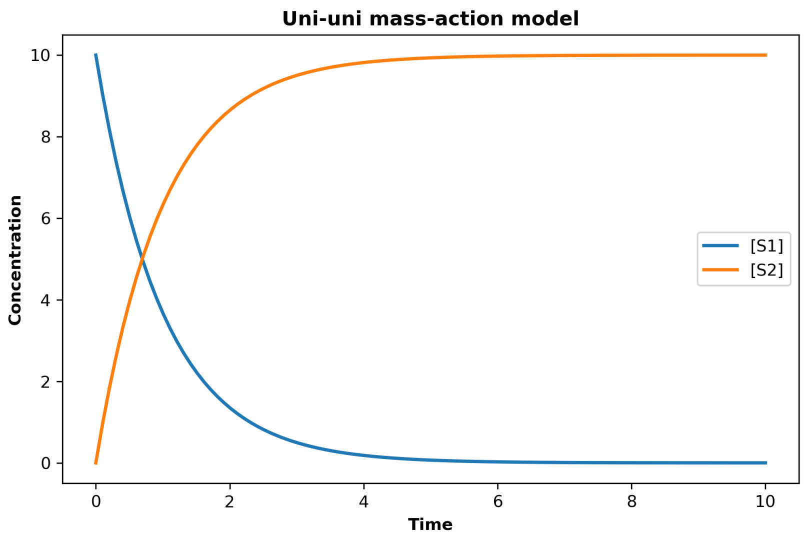
```
# First, install Tellurium, which comes with libRoadRunner
!pip install tellurium
```

Importing relevant packages

```
import tellurium as te # Python-based modeling environment for kinetic models
import roadrunner as rr # High-performance simulation and analysis library
import numpy as np # Scientific computing package
import random # Generate random numbers
import matplotlib.pylab as plt # Additional Python plotting utilities
```

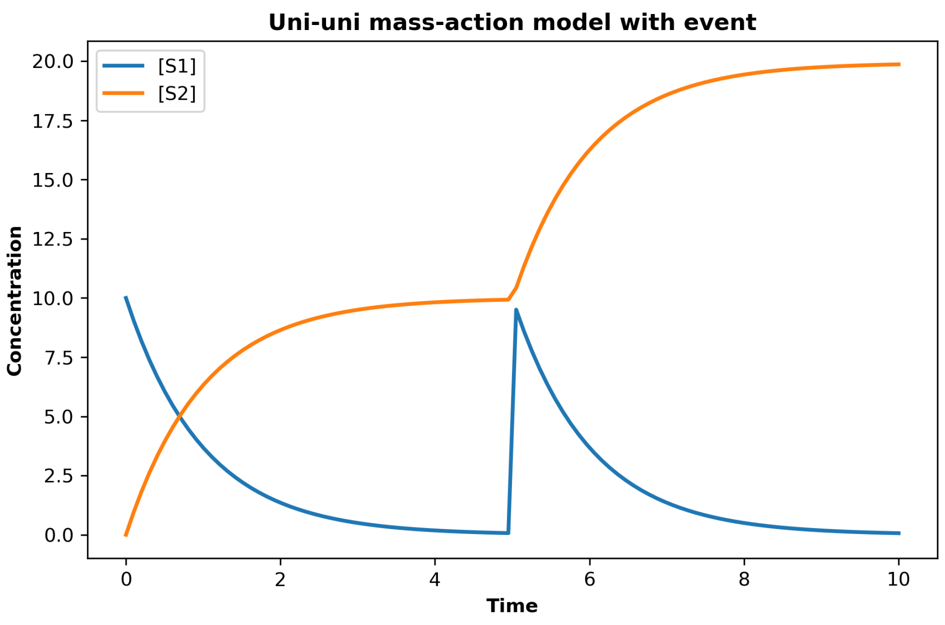
Writing a simple Antimony model

```
Ant_str = """
model test                  # name the model
    compartment C1;         # specify compartments
    C1 = 1.0;               # assign compartment volume
    species S1, S2;

    S1 = 10.0;              # assign species initial conditions
    S2 = 0.0;
    S1 in C1; S2 in C1;     # allocate species to appropriate compartment
    J1: S1 -> S2; k1*S1;    # reaction; reaction rate law;

    k1 = 1.0;               # assign constant values to global parameters
end
"""

r = te.loada(Ant_str)  # create an executable model by loading the string to a RoadRunner object instance
r.simulate(0, 10, 100) # simulate(time_start, time_end, number_of_points)
r.plot(title = 'Uni-uni mass-action model', xtitle = 'Time', ytitle = 'Concentration')
```



Writing an Antimony model with an event

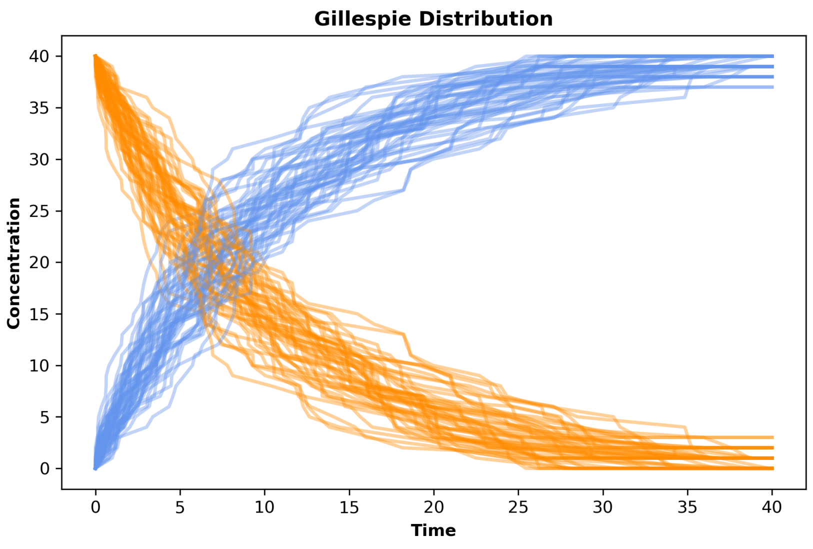
```
Ant_str = """
model test                  # name the model
    compartment C1;         # specify compartments
    C1 = 1.0;               # assign compartment volume
    species S1, S2;

    S1 = 10.0;              # assign species initial conditions
    S2 = 0.0;
    S1 in C1; S2 in C1;     # allocate species to appropriate compartment
    J1: S1 -> S2; k1*S1;  # reaction; reaction rate law;

    k1 = 1.0;               # assign constant values to global parameters

    E1: at (time > 5): S1 = 10; # add an event - spike in S1
end
"""
r = te.loada(Ant_str)  # create an executable model by loading the string to a RoadRunner object instance
r.simulate(0, 10, 100) # simulate(time_start, time_end, number_of_points)
r.plot(title = 'Uni-uni mass-action model with event', xtitle = 'Time', ytitle = 'Concentration')
```



Performing a stochastic simulation with the Gillespie algorithm

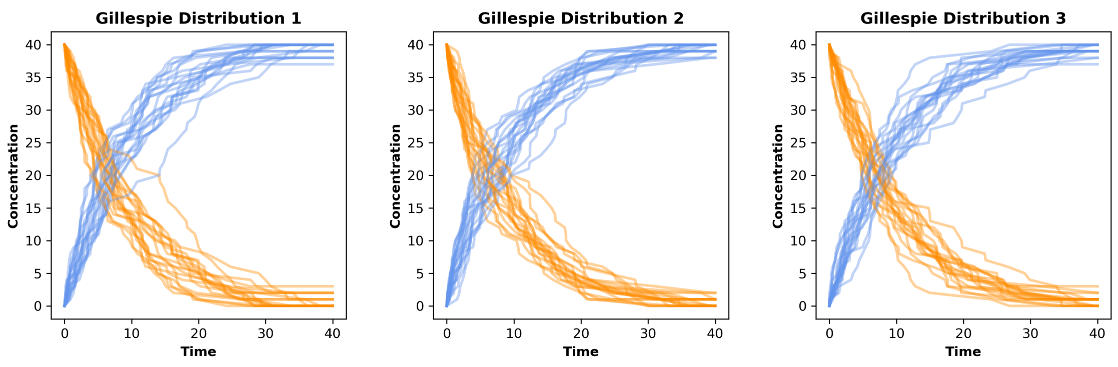
Generating a stochastic distribution with set seed value

```
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 40')
r.integrator = 'gillespie' # select the stochastic integrator - which uses the Gillespie algorithm

# Can set the seed value for the random number generator to obtain a deterministic distribution:
r.integrator.seed = 1234

multi_run_result = []
for k in range(1, 50):
    r.reset()
    result = r.simulate(0, 40)
    multi_run_result.append(result)
    te.plotArray(result, title = 'Gillespie Distribution',\
                 xlabel = 'Time', ylabel = 'Concentration',\
                 show=False, alpha=0.4) # setting alpha value changes trajectory opacity
te.show()
```

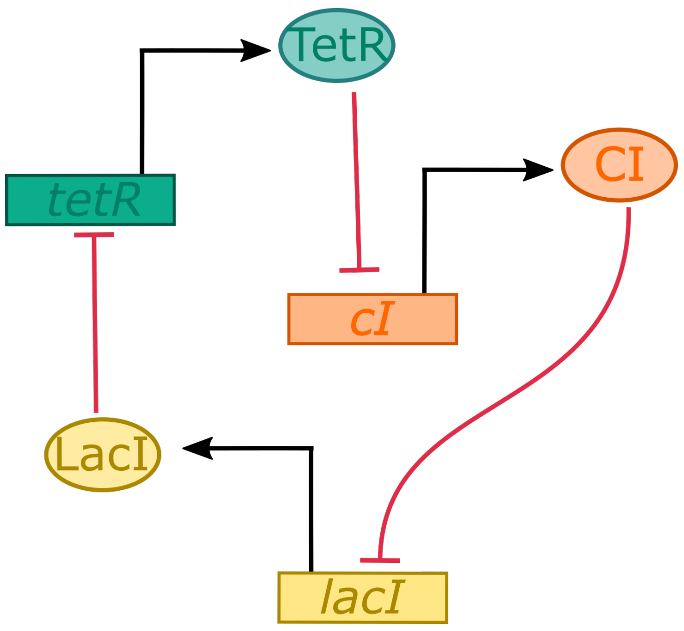


Generating stochastic distributions with variable seed values

```
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 40')
# if a seed value is not set, each run will return
# slightly different results that approximate the same distribution

for i in range(1,4):
    plt.figure(i)
    results = []
    for k in range(1, 20):
        r.reset()
        s = r.gillespie(0, 40)
        results.append(s)
        te.plotArray(s, title = 'Gillespie Distribution ' + str(i),\
                     xlabel = 'Time', ylabel = 'Concentration', show=False, alpha=0.4)
    te.show()
```



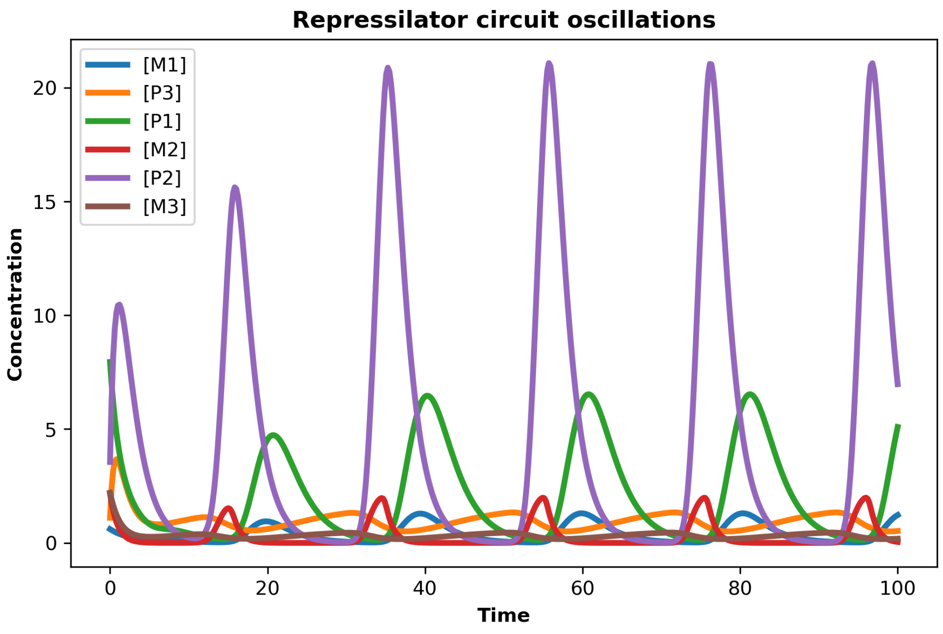
Writing an Antimony model with interactions



Repressilator circuit from Elowitz & Liebler (2000) [5]

```
repressilator_str = """
# Compartments and Species:
species M1, P3, P1, M2, P2, M3;

# Reactions:
J0:  -> M1; a_m1*(Kr_P3^n1/(Kr_P3^n1 + P3^n1)) + leak1;
J1: M1 -> ; d_m1*M1;
J2:  -> P1; a_p1*M1;
J3: P1 -> ; d_p1*P1;
J4:  -> M2; a_m2*(Kr_P1^n2/(Kr_P1^n2 + P1^n2)) + leak2;
J5: M2 -> ; d_m2*M2;
J6:  -> P2; a_p2*M2;
J7: P2 -> ; d_p2*P2;
J8:  -> M3; a_m3*(Kr_P2^n3/(Kr_P2^n3 + P2^n3)) + leak3;
J9: M3 -> ; d_m3*M3;
J10:  -> P3; a_p3*M3;
J11: P3 -> ; d_p3*P3;

# Species initializations:
M1 = 0.604016261711246;
P3 = 1.10433330559171;
P1 = 7.94746428021418;
M2 = 2.16464969760648;
P2 = 3.55413750091507;
M3 = 2.20471854765531;

# Variable initializations:
a_m1 = 1.13504504342841;
Kr_P3 = 0.537411795656332;
n1 = 7.75907326833983;
leak1 = 2.59839004225795e-07;
d_m1 = 0.360168301619141;
a_p1 = 5.91755684808254;
d_p1 = 1.11075218613419;
a_m2 = 2.57306185467814;
```

```
Kr_P1 = 0.190085253528206;
n2 = 6.89140262856765;
leak2 = 1.51282707494481e-06;
d_m2 = 1.05773721506759;
a_p2 = 8.35628834784826;
d_p2 = 0.520562081730298;
a_m3 = 0.417889543691157;
Kr_P2 = 2.71031378955001;
n3 = 0.44365980532785;
leak3 = 3.63586125130783e-11;
d_m3 = 0.805873530762994;
a_p3 = 4.61276807677109;
d_p3 = 1.54954108126666;

# Other declarations:
const a_m1, Kr_P3, n1, leak1, d_m1, a_p1, d_p1, a_m2, Kr_P1, n2, leak2, d_m2;
const a_p2, d_p2, a_m3, Kr_P2, n3, leak3, d_m3, a_p3, d_p3;
"""
repressilator_ant = te.loada(repressilator_str)
repressilator_ant.simulate(0, 100, 500)
repressilator_ant.plot(figsize = (8, 5), linewidth = 3)
```



Repressilator circuit parameters were optimized for oscillatory dynamics using bifurcation optimization software [6].

Standard SBML model description format

```python
#%% repressilator model SBML
print(repressilator_ant.getCurrentSBML())
```

```
<?xml version="1.0" encoding="UTF-8"?>
<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="M1" compartment="default_compartment" initialConcentration="1.22326489649967" hasOnlySubstanceUnits="fa
lse" boundaryCondition="false" constant="false"/>
      <species id="P3" compartment="default_compartment" initialConcentration="0.523696242501221" hasOnlySubstanceUnits="f
alse" boundaryCondition="false" constant="false"/>
      <species id="P1" compartment="default_compartment" initialConcentration="5.09251992154855" hasOnlySubstanceUnits="fa
lse" boundaryCondition="false" constant="false"/>
      <species id="M2" compartment="default_compartment" initialConcentration="0.0419666930233048" hasOnlySubstanceUnits
="false" boundaryCondition="false" constant="false"/>
      <species id="P2" compartment="default_compartment" initialConcentration="6.97093485510087" hasOnlySubstanceUnits="fa
lse" boundaryCondition="false" constant="false"/>
      <species id="M3" compartment="default_compartment" initialConcentration="0.183610489926808" hasOnlySubstanceUnits="f
alse" boundaryCondition="false" constant="false"/>
```

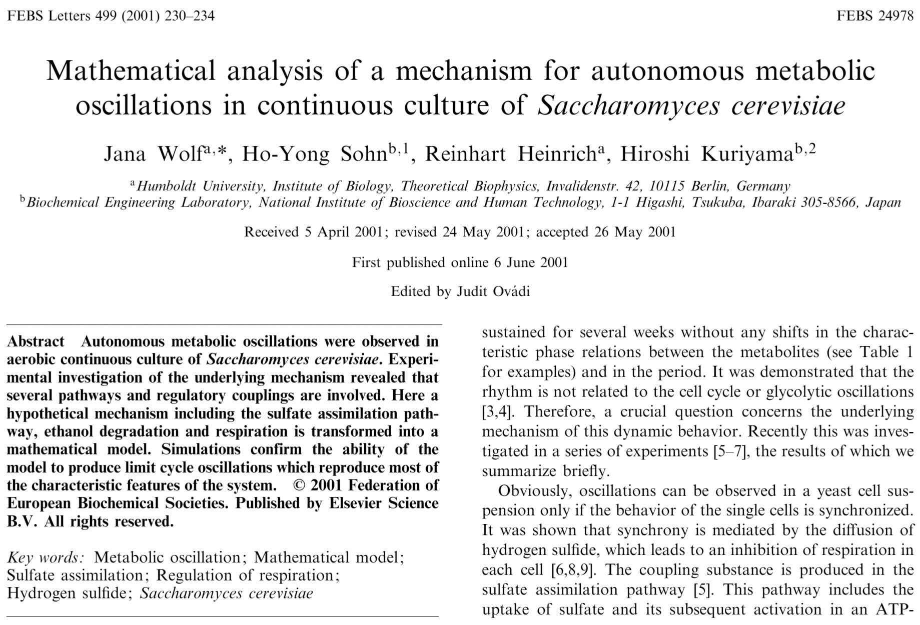
Human-readable Antimony String

```python
#%% repressilator model Antimony string
print(repressilator_ant.getAntimony())
```

```
// Created by libAntimony v2.9.4
// Compartments and Species:
species M1, P3, P1, M2, P2, M3;

// Reactions:
J0:  -> M1; a_m1*(Kr_P3^n1/(Kr_P3^n1 + P3^n1)) + leak1;
J1: M1 -> ; d_m1*M1;
J2:  -> P1; a_p1*M1;
J3: P1 -> ; d_p1*P1;
J4:  -> M2; a_m2*(Kr_P1^n2/(Kr_P1^n2 + P1^n2)) + leak2;
J5: M2 -> ; d_m2*M2;
J6:  -> P2; a_p2*M2;
J7: P2 -> ; d_p2*P2;
J8:  -> M3; a_m3*(Kr_P2^n3/(Kr_P2^n3 + P2^n3)) + leak3;
J9: M3 -> ; d_m3*M3;
J10:  -> P3; a_p3*M3;
J11: P3 -> ; d_p3*P3;

// Species initializations:
M1 = 0.604016261711246;
P3 = 1.10433330559171;
P1 = 7.94746428021418;
M2 = 2.16464969760648;
P2 = 3.55413750091507;
M3 = 2.20471854765531;
```

Import and export capabilities with Tellurium

Models can be imported from the BioModels Database, given the appropriate BioModel ID. We can import this model of respiratory oscillations in Saccharomyces cerevisae by Wolf et al. 2001 [7]:



FEBS Letters 499 (2001) 230–234 FEBS 24978

Mathematical analysis of a mechanism for autonomous metabolic oscillations in continuous culture of *Saccharomyces cerevisiae*

Jana Wolf[a,*], Ho-Yong Sohn[b,1], Reinhart Heinrich[a], Hiroshi Kuriyama[b,2]

[a] *Humboldt University, Institute of Biology, Theoretical Biophysics, Invalidenstr. 42, 10115 Berlin, Germany*
[b] *Biochemical Engineering Laboratory, National Institute of Bioscience and Human Technology, 1-1 Higashi, Tsukuba, Ibaraki 305-8566, Japan*

Received 5 April 2001; revised 24 May 2001; accepted 26 May 2001

First published online 6 June 2001

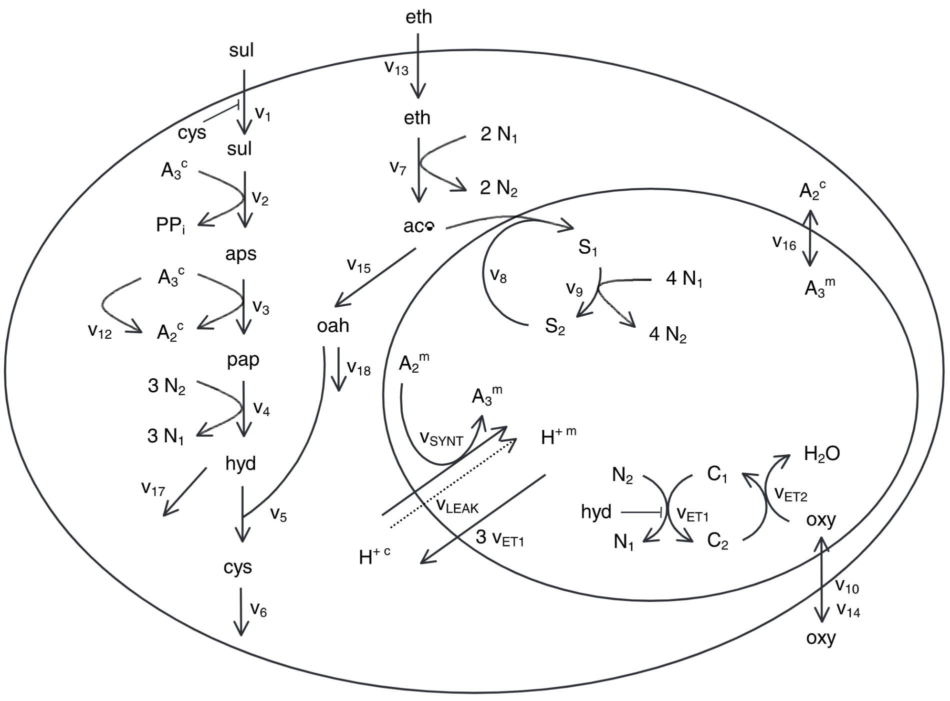
Edited by Judit Ovádi

Abstract **Autonomous metabolic oscillations were observed in aerobic continuous culture of *Saccharomyces cerevisiae*. Experimental investigation of the underlying mechanism revealed that several pathways and regulatory couplings are involved. Here a hypothetical mechanism including the sulfate assimilation pathway, ethanol degradation and respiration is transformed into a mathematical model. Simulations confirm the ability of the model to produce limit cycle oscillations which reproduce most of the characteristic features of the system. © 2001 Federation of European Biochemical Societies. Published by Elsevier Science B.V. All rights reserved.**

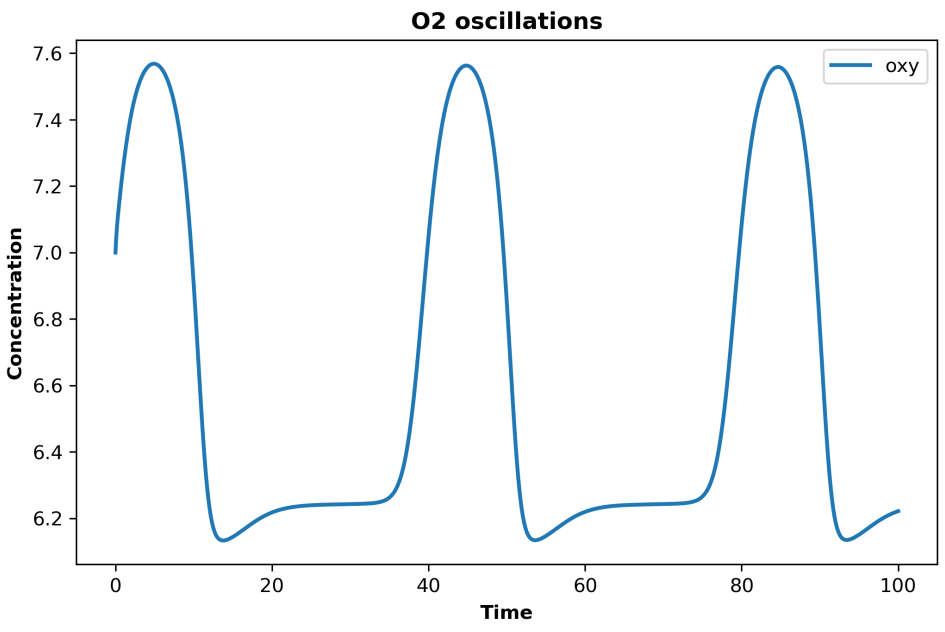
Key words: Metabolic oscillation; Mathematical model; Sulfate assimilation; Regulation of respiration; Hydrogen sulfide; *Saccharomyces cerevisiae*

sustained for several weeks without any shifts in the characteristic phase relations between the metabolites (see Table 1 for examples) and in the period. It was demonstrated that the rhythm is not related to the cell cycle or glycolytic oscillations [3,4]. Therefore, a crucial question concerns the underlying mechanism of this dynamic behavior. Recently this was investigated in a series of experiments [5–7], the results of which we summarize briefly.

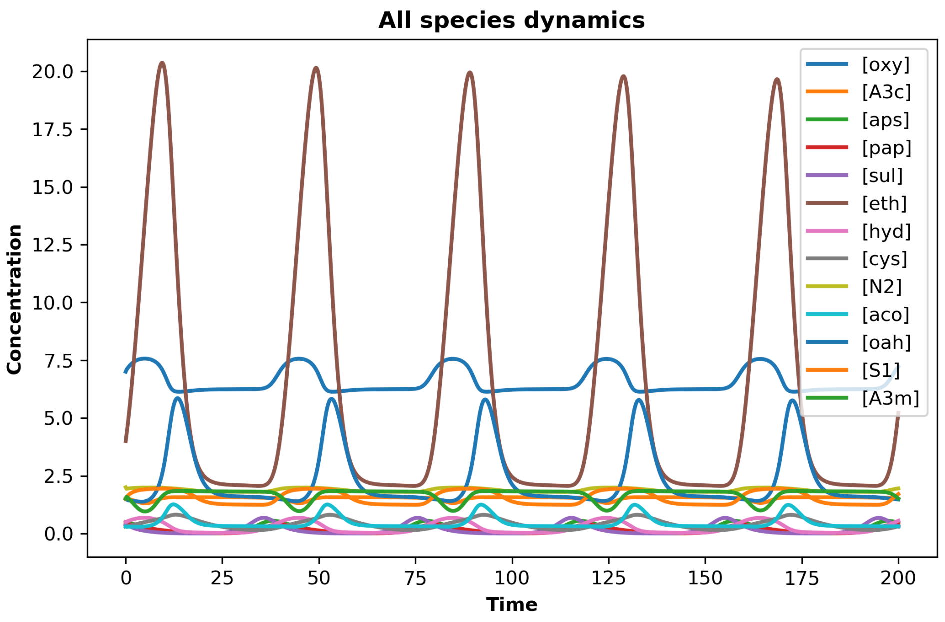
Obviously, oscillations can be observed in a yeast cell suspension only if the behavior of the single cells is synchronized. It was shown that synchrony is mediated by the diffusion of hydrogen sulfide, which leads to an inhibition of respiration in each cell [6,8,9]. The coupling substance is produced in the sulfate assimilation pathway [5]. This pathway includes the uptake of sulfate and its subsequent activation in an ATP-



```
#%% You can load an SBML model directly from the BioModels Database, given the BioModel ID
wolf = te.loadSBMLModel("http://www.ebi.ac.uk/biomodels-main/download?mid=BIOMD0000000090")
wolf.simulate(0, 100, 1000, ['time', 'oxy'])
wolf.plot(figsize = (8, 5), title = 'O2 oscillations', xtitle = 'Time', ytitle = 'Concentration')
```



```
# Export the model you just accessed from BioModels to the current directory as an SBML string
wolf.reset()
wolf.getFloatingSpeciesConcentrationIds()
wolf.exportToAntimony('wolf.txt', current = True)
# wolf.exportToSBML('wolf.xml', current = True)

wolf = te.loada('wolf.txt') # load the Antimony string you just saved to the working directory
wolf.simulate(0, 200, 1000)
wolf.plot(figsize = (15, 10), xtitle = 'Time', ytitle = 'Concentration')
```



Accessing useful matrices and vectors in Tellurium

Stoichiometry matrix

```
r = te.loada('S1 -> S2; k1*S1; k1 = 0.1; S1 = 40') # simple model
# The stoichiometry matrix
r.getFullStoichiometryMatrix()
# short-cut
r.sm()
```

Jacobian matrix

```
# The Jacobian matrix
r.getFullJacobian()
# short-cut
r.fjac()
```

Reaction rates

```
# reaction rate vector
r.getReactionRates()
# short-cut
r.rv()
```

Rates of change

```
# the rate of change vector
r.getRatesOfChange()
# short-cut
r.dv()
```

Floating species concentrations

```
# the species concentration vector
r.getFloatingSpeciesConcentrations()
# short-cut
r.sv()
```

Floating species IDs

```
# the names of all floating species
r.getFloatingSpeciesIds()
# short-cut
r.fs()
```

Global parameter values and IDs

```
# the global parameter value vector
r.getGlobalParameterValues()
```

```
# the names of all kinetic parameters in the model
r.getGlobalParameterIds()
# short-cut
r.ps()
```

Advanced analysis with Tellurium

Computing the steady state

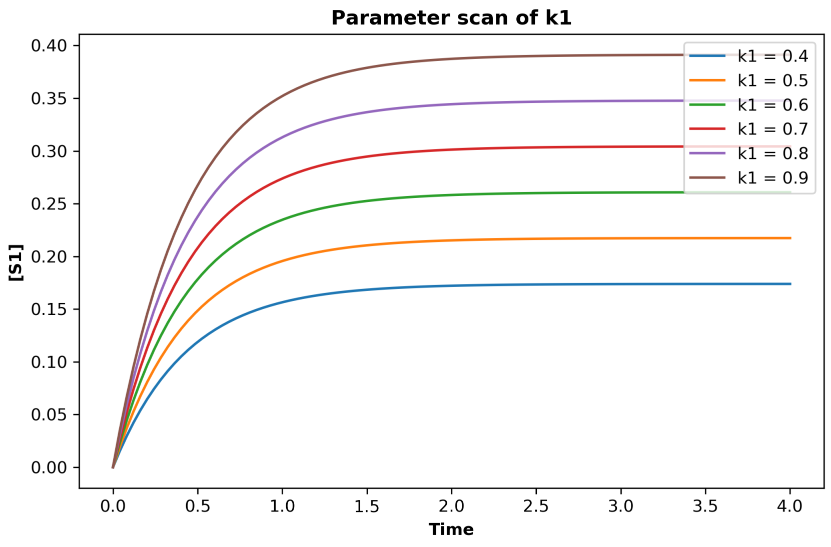
```
r.reset()
r.steadyState() # Bring the model to steady state using a steady state solver
print(r.getFloatingSpeciesConcentrations())
```

Computing eigenvalues for stability analysis

```
r.reset()
print(r.getFullEigenValues()) # compute the eigenvalues at the current state of the system
```

Parameter scans and plotting

```
#%%  Example of flexibility with Python programming
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;
''')

for k1 in np.arange(0.4, 1, 0.1):
    r.k1 = k1
    r.reset()
    result = r.simulate(0, 4, 100, ['time', 'S1'])
    plt.plot(result['time'], result['S1'], label = 'k1 = ' + format(k1, ".1f"))

plt.legend(loc = 'upper right')
plt.xlabel('Time'); plt.ylabel('[S1]')
plt.show()
```



Changing integrator and steady state solver settings with libRoadRunner

```
print(rr.integrators) # shows the integrators implemented in tellurium

# cvode, from the SUNDIALS suit, is the default and performs stiff and non-stiff numerical integration
```

```
['cvode', 'gillespie', 'rk4', 'rk45', 'euler']
```

```
print(rr.steadyStateSolvers) # shows the steady state solvers currently implemented

# nleq1 and nleq2 both implement variants of the Newton method to solve for the steady state concentrations
```

```
['nleq1', 'nleq2']
```

```
print(r.getSteadyStateSolver()) # shows the settings for the steady state solver
```

```
 < roadrunner.SteadyStateSolver() >
   name: nleq2
   settings:
      allow_presimulation: true
     presimulation_maximum_steps: 100
       presimulation_time: 100
             allow_approx: true
         approx_tolerance: 0.000001
     approx_maximum_steps: 10000
              approx_time: 10000
       relative_tolerance: 0.000000000001
       maximum_iterations: 100
          minimum_damping: 1e-20
           broyden_method: 0
                linearity: 3
```

```
r.steadyStateSolver.approx_maximum_steps = 12000 # change the settings for the steady state solver
```

Tutorial Challenges

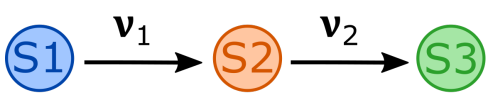
Note 1: The challenges and tips in this tutorial assume you have imported the following:

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.

Challenge 1

Create an Antimony string for the following reaction network:



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

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

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

Tips:

- Antimony strings can be saved to a variable like any string in Python
- Antimony strings should be specified with triple quotation marks, 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

Solution:

```
# Write a simple antimony string
antStr = '''

    J1: S1 -> S2; k1*S1; # Reaction names 'J1', 'J2'
    J2: S2 -> S3; k2*S2;

    k1 = 0.15; k2 = 0.45; # Specify parameter values
    S1 = 1; S2 = 0; S3 = 0; # Specify initial conditions
'''
```

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)

Solution:

```
# Write a simple antimony string
# Load the Antimony string to a RoadRunner object instance 'r'
r = te.loada(antStr)

# Simulate the model (numerically integrate the differential equations)
result = r.simulate(0, 20, 50)
```

Challenge 3

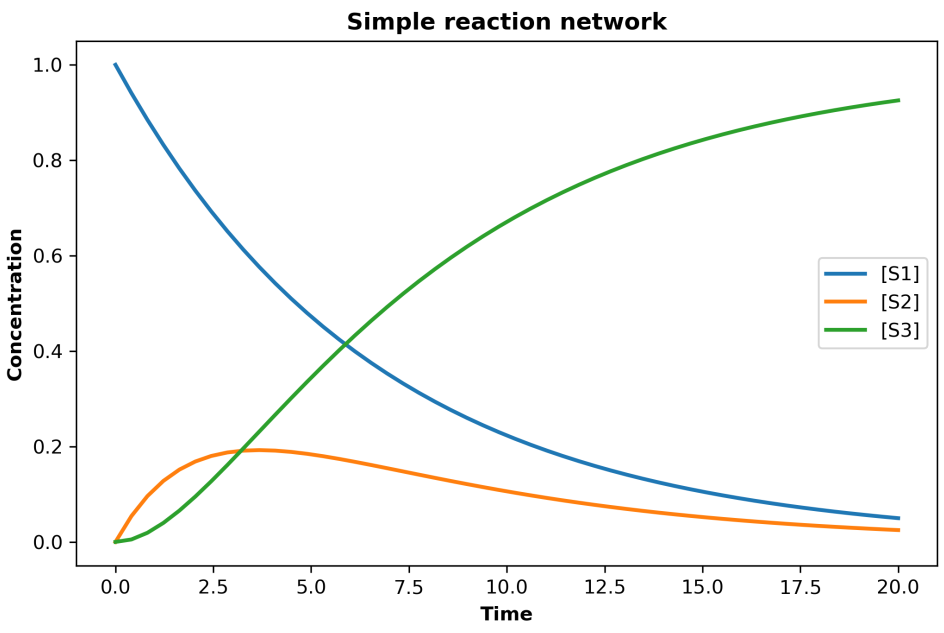
Plot the simulation with a title and x and y axis labels.

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.

Solution:

```
# Plot the simulation results
r.plot(title  = 'Simple reaction network', xtitle = 'Time', ytitle = 'Concentration')
```



Challenge 4

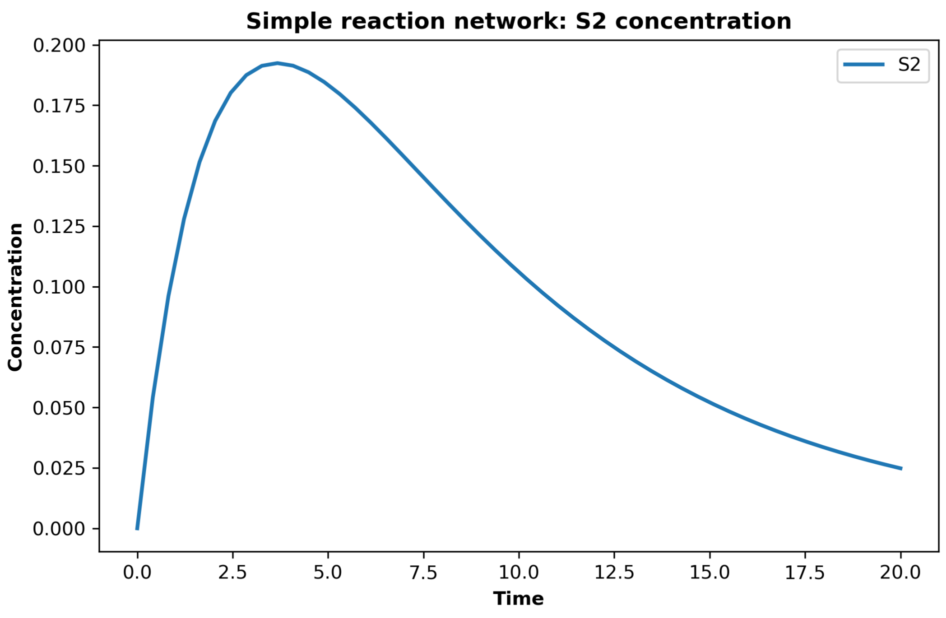
Reset your model, simulate, and plot only the species S2 concentration time-course.

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'

Solution:

```
# Reset the model species concentrations to the initial conditions
r.reset()

# Simulate the model with selections 'time' and 'S2'
r.simulate(0, 20, 50, ['time', 'S2'])
r.plot(title  = 'Simple reaction network: S2 concentration', xtitle = 'Time', ytitle = 'Concentration')
```



Challenge 5

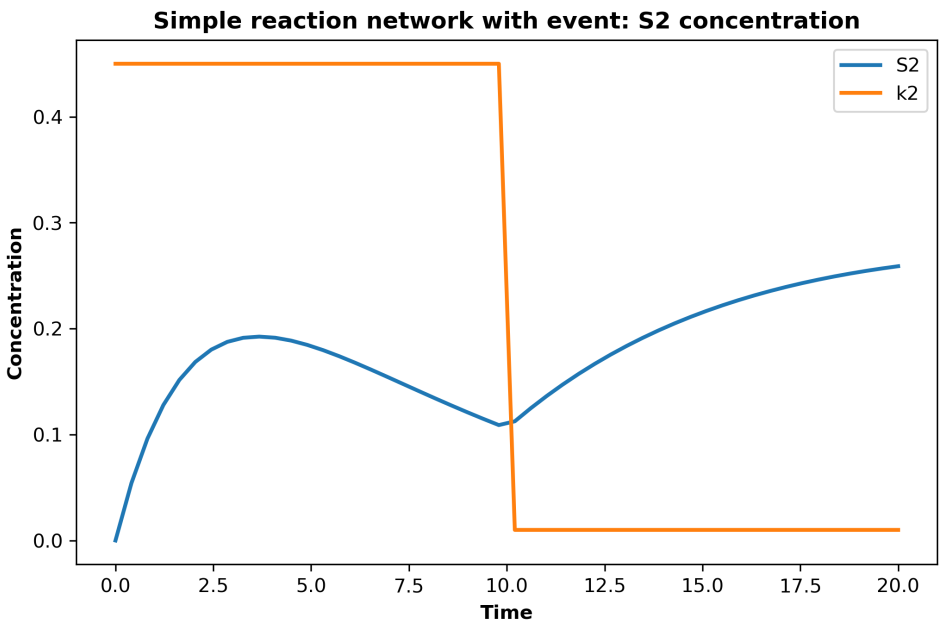
Add an event to your Antimony string to set k2 = 0.01 after 10 seconds of simulation time. Load the string to create a RoadRunner instance, simulate the model and plot the time-course of S2 concentration and k2 values throughout the simulation.

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

Solution:

```
# Add an event to the Antimony string
antStr_with_event = antStr + '''E1: at (time > 10): k2 = 0.01'''
r = te.loada(antStr_with_event)

# Simulate the model with an event, and record the value of 'k2'
r.simulate(0, 20, 50, ['time', 'S2', 'k2'])
r.plot(title  = 'Simple reaction network with event: S2 concentration', xtitle = 'Time', ytitle = 'Concentration')
```



Challenge 6

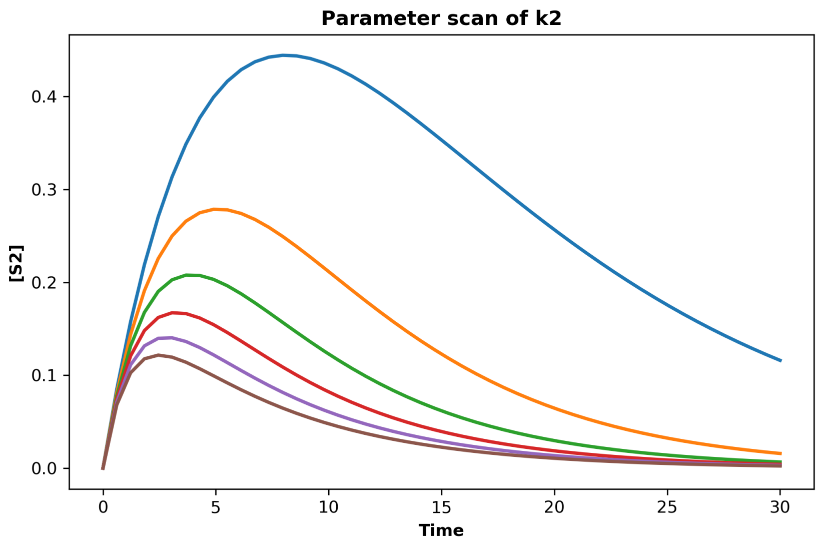
Perform and plot a parameter scan of ’k1’ from k1 = 0.1 to k1 = 1.0 in steps of 0.15, using your model from Challenge 1 (which does not include events). Plot the simulation trajectory of [S2] only, and ensure that all the simulation trajectories for the scan are displayed on a single plot.

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()

Solution:

```
r = te.loada(antStr)
k2_vals = np.arange(0.1, 1.0, 0.15) # choose the value of k2 from 0.1 to 1.0 in 0.15 increments
for k2 in k2_vals:
    r.k2 = k2 # set the value of k2 at each loop iteration
    r.reset() # reset the variable species concentrations
    result = r.simulate(0, 30, 50, ['time', 'S2'])

    # store plot specifications, but do not show plot at each iteration of for loop
    te.plotArray(result, title = 'Parameter scan of k2', xlabel = 'Time', ylabel = '[S2]',\
            show = False, resetColorCycle= False)
te.show()
```



Challenge 7

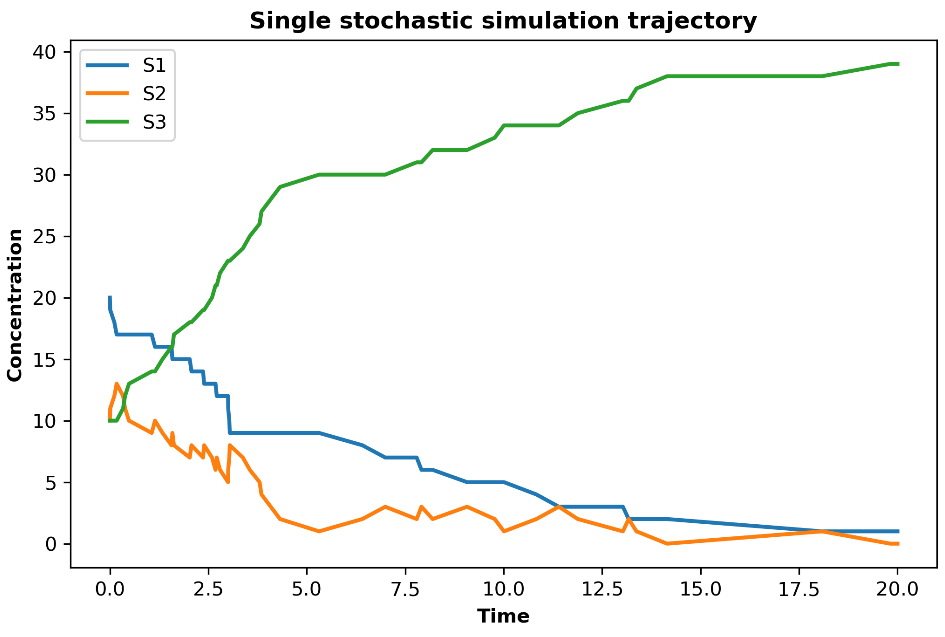
Set the initial conditions of your model to: [S1] = 20, [S2] = 10, [S3] = 10. Perform and plot a stochastic simulation of your model. Next, produce a distribution of stochastic simulation trajectories.

Tips:

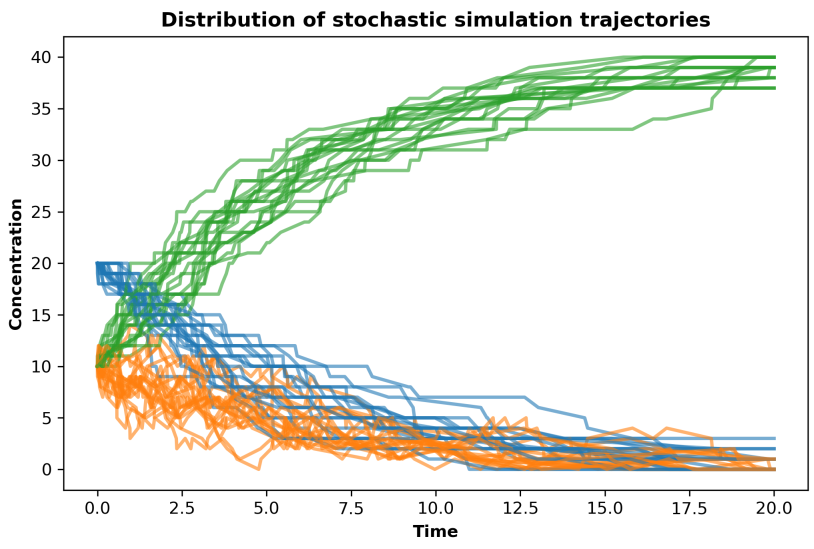
- Use model.gillespie(start_time, end_time, number_of_points) to perform a stochastic simulation
- The Gillespie algorithm will randomly select which reaction to execute at randomly selected time intervals over the course of your simulation. Therefore, you may want to increase the initial concentrations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

Solution:

```
r.reset()
r.S1 = 20 # set the initial value for [S1]
r.S2 = 10 # set the initial value for [S2]
r.S3 = 10 # set the initial value for [S3]
all_species_selections = ['time'] + r.getFloatingSpeciesIds()

# Use the Gillespie algorithm to perform the simulation
r.gillespie(0, 20, selections = all_species_selections)
r.plot(title = 'Single stochastic simulation trajectory', xtitle = 'Time', ytitle = 'Concentration')
```



```
r = te.loada(antStr)
results = []

# Create a distribution of stochastic trajectories
for k in range(1, 20):
    r.reset()
    r.S1 = 20
    r.S2 = 10
    r.S3 = 10
    s = r.gillespie(0, 20)
    results.append(s)
    te.plotArray(s, title = 'Distribution of stochastic simulation trajectories',\
                    xlabel = 'Time', ylabel = 'Concentration', show=False, alpha=0.6)
te.show()
```



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.

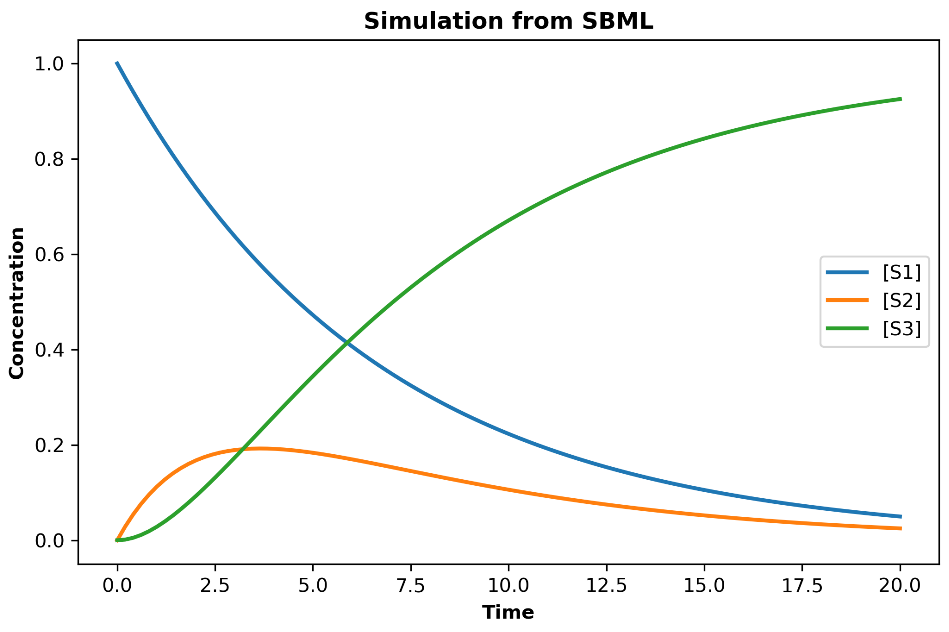
Saving your models as declarative SBML files will allow you to exchange your model between simulation, validation, and visualization platforms. To demonstrate the exchangeability of the SBML standard, launch this web-based simulator in your browser and locate and drag your saved SBML file from its current directory 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

Solution:

```
# Export the model to an SBML description format, set current = False to reset to the Antimony string values
r.exportToSBML(os.getcwd() + '\\three_species_sbml.xml', current = False)

# Load the saved SBML file to a RoadRunner instance
r = te.loadSBMLModel('three_species_sbml.xml')
r.simulate(0, 20, 100)
r.plot()
```



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.

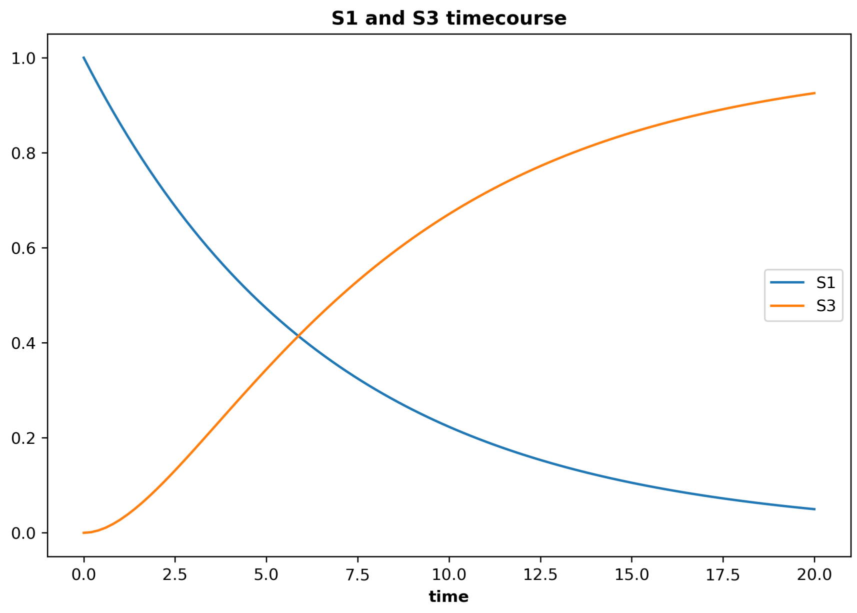
Tips:

- 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

Solution:

```
# Write your simulation experiment using a PhraSED-ML string
phrasedml_str = '''
  model1 = model "three_species_model.xml"
  sim1 = simulate uniform(0, 20, 100)
  task1 = run sim1 on model1
  plot "S1 and S3 timecourse" time vs S1, S3
'''

# convert the Antimony string from Challeng 1 to an SBML string
sbml_str = te.antimonyToSBML(antStr)

# Set your saved SBML three-species model as the SBML file that will be referenced
phrasedml.setReferencedSBML("three_species_model.xml", sbml_str)

# Create the SED-ML xml string from the phrasedml
sedml_str = phrasedml.convertString(phrasedml_str)
if sedml_str == None:
    print(phrasedml.getLastPhrasedError())

# Visualize the SED-ML file
print(sedml_str)

# Execute the simulation experiment specified with SED-ML
te.executeSEDML(sedml_str)

# Save the SED-ML simulation experiment to your current working directory
te.saveToFile(str(os.getcwd()) + '\\three_species_model_simulation_experiment.xml', sedml_str)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Created by phraSED-ML version v1.0.9 with libSBML version 5.15.0. -->
<sedML xmlns="http://sed-ml.org/sed-ml/level1/version2" level="1" version="2">
  <listOfSimulations>
    <uniformTimeCourse id="sim1" initialTime="0" outputStartTime="0" outputEndTime="20" numberOfPoints="100">
      <algorithm kisaoID="KISAO:0000019"/>
    </uniformTimeCourse>
  </listOfSimulations>
  <listOfModels>
    <model id="model1" language="urn:sedml:language:sbml.level-3.version-1" source="three_species_model.xml"/>
  </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" mod
elReference="model1"/>
      </listOfVariables>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <ci> S1 </ci>
      </math>
    </dataGenerator>
    <dataGenerator id="plot_0_1_1" name="S3">
      <listOfVariables>
        <variable id="S3" target="/sbml:sbml/sbml:model/sbml:listOfSpecies/sbml:species[@id='S3']" taskReference="task1" mod
elReference="model1"/>
      </listOfVariables>
      <math xmlns="http://www.w3.org/1998/Math/MathML">
        <ci> S3 </ci>
      </math>
    </dataGenerator>
  </listOfDataGenerators>
  <listOfOutputs>
    <plot2D id="plot_0" name="S1 and S3 timecourse">
      <listOfCurves>
        <curve id="plot_0__plot_0_0_0__plot_0_0_1" logX="false" logY="false" xDataReference="plot_0_0_0" yDataReference="plo
t_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="plo
t_0_1_1"/>
      </listOfCurves>
    </plot2D>
  </listOfOutputs>
</sedML>
```



```python
# Execute the SED-ML directly from the file you saved to the current working directory
te.executeSEDML('three_species_model_simulation_experiment.xml')
```

Challenge 10

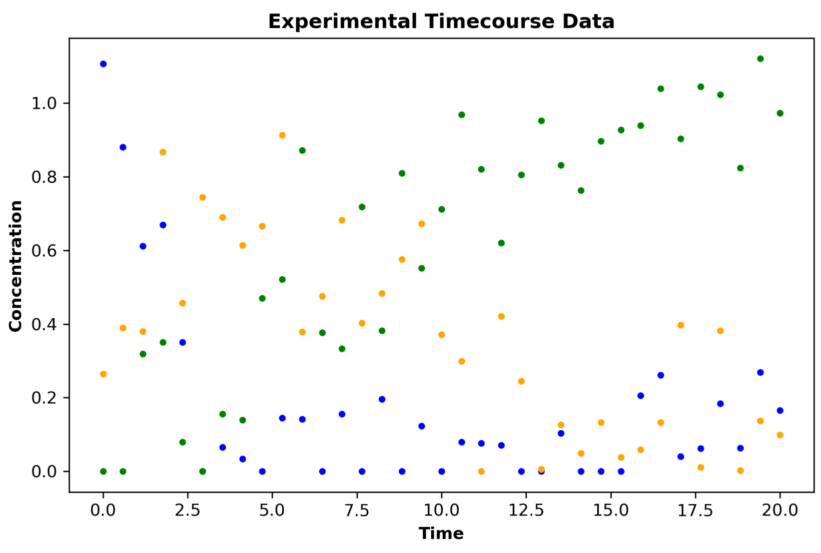
Download the data (saved as a numpy array file) simple_model_training_data.npy on GitHub to your working directory. These data represent noisy experimental time-course results for [S1], [S2], [S3] in the system you are modeling from Challenge 1. The four columns in the data correspond to 'time', '[S1]',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

Solution:

```
# Download the data from GitHub

import urllib.request # use this library to download file from GitHub
import os # use this library to determine current working directory to download the data from Github to

# URL to the data
url = 'https://github.com/vporubsky/tellurium-libroadrunner-tutorial/blob/master/simple_model_training_data.npy?raw=true'

# Obtain the path for the current working directory
working_dir = os.getcwd()

# Save the data from the url
filename, headers = urllib.request.urlretrieve(url, filename= working_dir + "\\simple_model_training_data.npy")

# Load the experimental data
training_data = np.load('simple_model_training_data.npy') # col 1 = time, col 2 = [S1], col 3 = [S2], col 4 = [S3]

# Plot the experimental data
color_list = ['blue', 'orange', 'green']
for i in range(3):
    plt.plot(training_data[:, 0], training_data[:, i + 1], '.', color = color_list[i], linewidth = 3)
plt.title('Experimental Timecourse Data', fontweight="bold")
plt.xlabel('Time', fontweight="bold")
plt.ylabel('Concentration', fontweight="bold")
plt.legend(['[S1]', '[S2]', '[S3]'])
```



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

Solution:

```
# Choose an optimization method and import relevant libraries/functions
from scipy.optimize import differential_evolution

# Load your Antimony string model from Challenge 1 and examine the parameters
r = te.loada(antStr)
parameters = r.getGlobalParameterIds() # parameters which will be optimized

# Enter column index to determine which species data to use for fitting
def select_species(species_indices = [1, 2, 3]):
    selections = species_indices
    names = []
    for i in selections:
        names.append(r.getFloatingSpeciesIds()[i-1])
    return selections, names

# Write an objective function to minimize with differential evolution, such as the root mean square error (RMSE)

def sim_model(s): # function will produce simulation results for the input parameter values
    r.reset()

    # Set parameter value in model using values in s, which will be determined by the
    # scipy.optimize.differential_evolution algorithm

    for index, p in enumerate(r.getGlobalParameterIds()):
        r.setValue (p, s[index])
    simulation_data = r.simulate(training_data[0, 0], training_data[len(training_data)-1 , 0],\
            len(training_data)) # (start, end, timepoints)
    return simulation_data

def rmse(s):
    model_prediction = sim_model(s)
    species_sum = (sum((training_data[:, species_fitting_selections]\
                    - model_prediction[:, species_fitting_selections])**2))\
                    /len(training_data[:, species_fitting_selections])
    return sum(species_sum)

    # Perform the optimization/ parameter fitting using scipy.optimize.differential_evolution

# use all species' data
species_fitting_selections, species_fitting_names = select_species(species_indices = [1,2,3])
# perform parameter fitting
model_fit_params = differential_evolution(rmse, [(0.00001, 20)]*len(parameters), maxiter=100)

print(model_fit_params) # examine the results of optimization with scipy.optimize.differential_evolution
print('')
print('Parameter fitting was performed using training data from the following species:')
print(species_fitting_names)
print('')
print('The optimized value for k1 is: ' + str(model_fit_params.x[0]))
print('The optimized value for k2 is: ' + str(model_fit_params.x[1]))
print('The ground truth values values are k1 = 0.55 and k2 = 0.15')
```

```
      fun: 0.06807607820677919
      jac: array([ 1.37112544e-06, -4.84473572e-06])
  message: 'Optimization terminated successfully.'
     nfev: 657
      nit: 20
  success: True
        x: array([0.5662429 , 0.14585021])

 Parameter fitting was performed using training data from the following species:
 ['S1', 'S2', 'S3']

 The optimized value for k1 is: 0.5662428983973844
 The optimized value for k2 is: 0.1458502059502918
 The ground truth values values are k1 = 0.55 and k2 = 0.15
```

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

Solution:

```
r.reset()
print('The ground truth values are k1 = 0.55 and k2 = 0.15')
for i in [1, 2, 3]:
    species_fitting_selections, species_fitting_names = select_species(species_indices = [i])
    model_fit_params = differential_evolution(rmse, [(0.00001, 20)]*len(parameters), maxiter=100) # parameter fitting
    print('')
    print('Parameter fitting was performed using training data from the following species:')
    print(species_fitting_names)
    print('')
    print('The optimized value for k1 is: ' + str(model_fit_params.x[0]))
    print('The optimized value for k2 is: ' + str(model_fit_params.x[1]))

print('')
print('When fitting with a single species timecourse, species S2 typically provides the best fit.')
```

```
The ground truth values are k1 = 0.55 and k2 = 0.15

Parameter fitting was performed using training data from the following species:
['S1']

The optimized value for k1 is: 0.4677657719782111
The optimized value for k2 is: 1.7099422434020015

Parameter fitting was performed using training data from the following species:
['S2']

The optimized value for k1 is: 0.7021260468810441
The optimized value for k2 is: 0.12489883228632653

Parameter fitting was performed using training data from the following species:
['S3']

The optimized value for k1 is: 0.2649520086641491
The optimized value for k2 is: 0.262067479853958

When fitting with a single species timecourse, species S2 typically provides the best fit.
```

Challenge 13

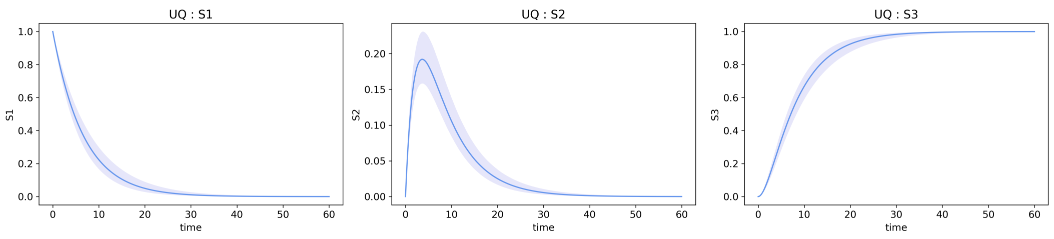
Plot the uncertainty in species S1, S2, and S3 concentrations in response to changes in the parameters, k1 and k2.

Tips:

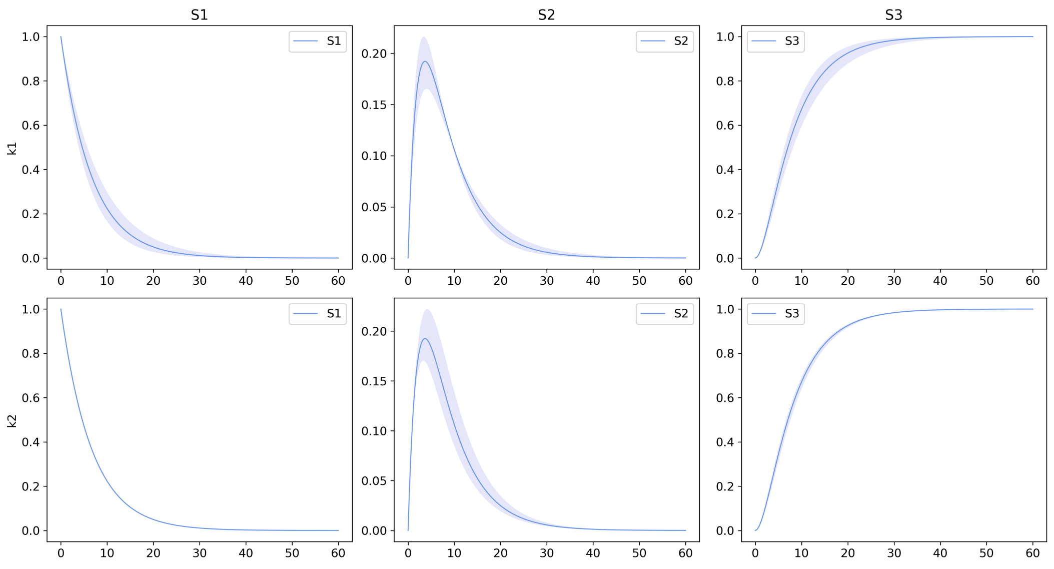
- Review the Python Help documentation for uncertainty quantification in Tellurium:
  - ?te.utils.uncertainty.UncertaintyAllP()
  - ?te.utils.uncertainty.UncertaintySingleP()

Solution:

```
# Sample all parameter values from a normal distribution and propagate the uncertainty
te.utils.uncertainty.UncertaintyAllP(r, variables = ['S1', 'S2', 'S3'], parameters = ['k1', 'k2'])
```



```
# Sample a single parameter value at a time using a normal distribution and propagate the uncertainty
te.utils.uncertainty.UncertaintySingleP(r, variables = ['S1', 'S2', 'S3'], parameters = ['k1', 'k2'])
```

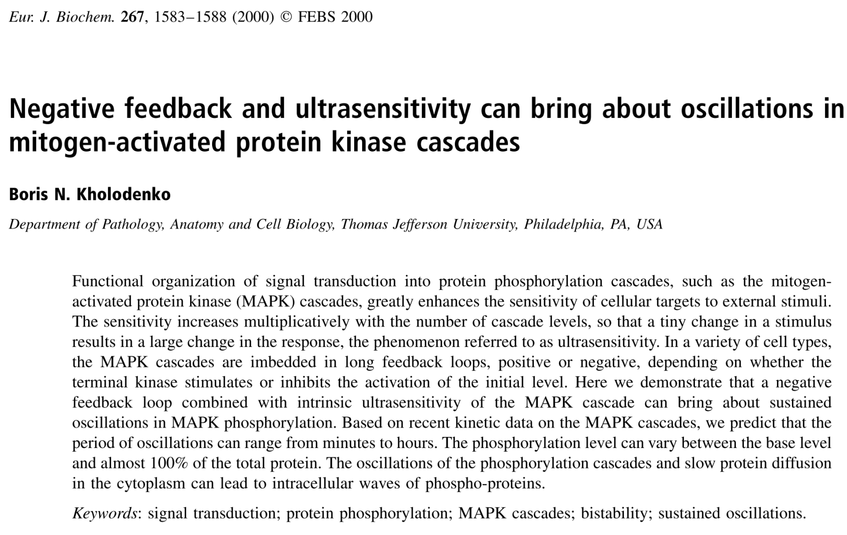


Challenge 14

Load, simulate, and plot the simulation results of the Kholodenko 2000 model of ultrasensitivity and negative feedback oscillations in the MAPK cascade [8] from the BioModels Database.

Tips:

- The BioModel ID for this model is: BIOMD0000000010



Eur. J. Biochem. **267**, 1583–1588 (2000) © FEBS 2000

Negative feedback and ultrasensitivity can bring about oscillations in the mitogen-activated protein kinase cascades

Boris N. Kholodenko

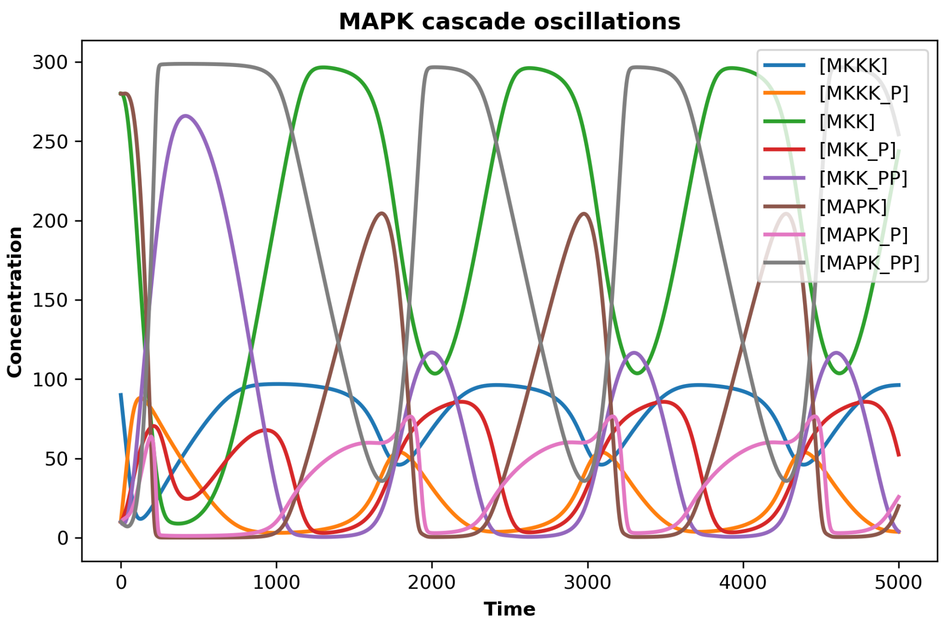
Department of Pathology, Anatomy and Cell Biology, Thomas Jefferson University, Philadelphia, PA, USA

Functional organization of signal transduction into protein phosphorylation cascades, such as the mitogen-activated protein kinase (MAPK) cascades, greatly enhances the sensitivity of cellular targets to external stimuli. The sensitivity increases multiplicatively with the number of cascade levels, so that a tiny change in a stimulus results in a large change in the response, the phenomenon referred to as ultrasensitivity. In a variety of cell types, the MAPK cascades are imbedded in long feedback loops, positive or negative, depending on whether the terminal kinase stimulates or inhibits the activation of the initial level. Here we demonstrate that a negative feedback loop combined with intrinsic ultrasensitivity of the MAPK cascade can bring about sustained oscillations in MAPK phosphorylation. Based on recent kinetic data on the MAPK cascades, we predict that the period of oscillations can range from minutes to hours. The phosphorylation level can vary between the base level and almost 100% of the total protein. The oscillations of the phosphorylation cascades and slow protein diffusion in the cytoplasm can lead to intracellular waves of phospho-proteins.

Keywords: signal transduction; protein phosphorylation; MAPK cascades; bistability; sustained oscillations.

Solution:

```
kholodenko = te.loadSBMLModel("http://www.ebi.ac.uk/biomodels-main/download?mid=BIOMD0000000010")
kholodenko.simulate(0, 5000, 5000)
kholodenko.plot(figsize = (10, 6), title = 'MAPK cascade oscillations',\
                xtitle = 'Time', ytitle = 'Concentration')
```

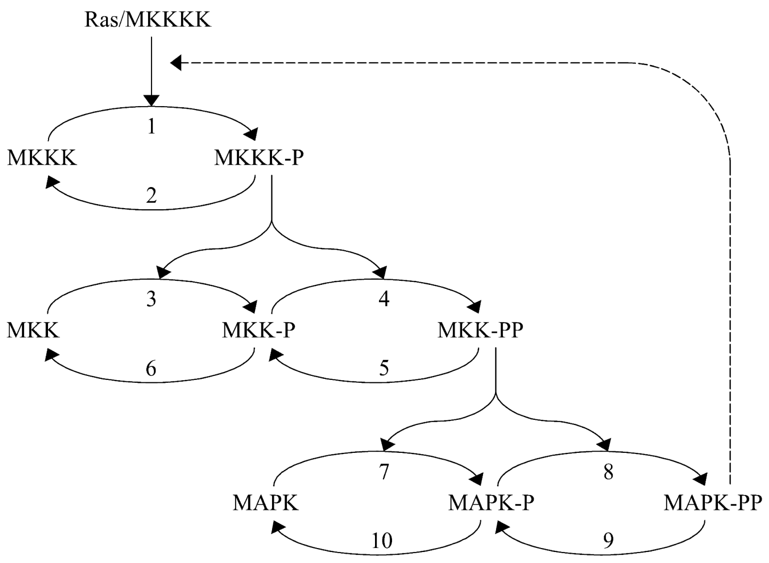


Challenge 15

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 from Kholdenko 2000 publication [8] 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

Solution:

```
# Try to compute the steady state of the Kholodenko 2000 model
# An error is raised because the Jacobian is singular and therefore cannot be inverted
kholodenko.steadyState()
```

```
RuntimeError: Both steady state solver and approximation routine failed: 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.; Failed to converge while running
approximation routine. Try increasing the time or maximum number of iteration via changing
the settings under r.steadyStateSolver where r is an roadrunner instance. Model might not
have a steady state.
```

```
# Examine the Jacobian matrix for the Kholodenko 2000 model
# Note that rows are linearly dependent on one another - this is due to the presence of conserved cycles
print(kholodenko.getFullJacobian())
```

```
                  MKKK,       MKKK_P,          MKK,       MKK_P,       MKK_PP,       MAPK,     MAPK_P,       MAPK_PP
MKKK    [[ -8.88976e-05,  0.00275773,            0,           0,            0,          0,          0,  0.000215073],
MKKK_P   [  8.88976e-05, -0.00275773,            0,           0,            0,          0,          0, -0.000215073],
MKK      [            0,  -0.0220693, -0.000433564,  0.00114779,            0,          0,          0,            0],
MKK_P    [            0, 0.000857124,  0.000433564, -0.00187204,  0.000807372,          0,          0,            0],
MKK_PP   [            0,   0.0212122,            0, 0.000724256, -0.000807372,          0,          0,            0],
MAPK     [            0,           0,            0,           0, -0.000853459, -0.160211, 0.0223849,            0],
MAPK_P   [            0,           0,            0,           0,  -0.00365954,  0.160211, -0.137714,  7.74585e-05],
MAPK_PP  [            0,           0,            0,           0,     0.004513,         0,  0.115329, -7.74585e-05]]
```

```python
# Turn on conserved moiety analysis to account for conserved cyles in the MAPK cascade
# Then you may compute the steady state

kholodenko.conservedMoietyAnalysis = True
kholodenko.steadyState()
print('Steady state computed successfully. Steady state values:')
print('')
kholodenko.getSteadyStateValuesNamedArray()
```

```
    [MKK_P], [MAPK_P],  [MKKK],    [MKK],   [MAPK], [MKK_PP], [MKKK_P], [MAPK_PP]
 [[ 41.1421,  99.9714, 76.6351, 238.914, 102.159,  19.9441,  23.3649,   97.8701]]
```

Acknowledgements

Tellurium	libRoadRunner	Antimony	Funding
Kiri Choi	Endre T. Somogyi	Lucian Smith	NIH R01 GM099138
J. Kyle Medley	Matthias Konig	J. Kyle Medley	NIH U01 GM111243
Matthias Konig	J. Kyle Medley	Herbert M. Sauro	NIH R01 GM076692
Kaylene Stocking	Andrew Hu		EPA RD83500101
Lucian Smith	Faye Yu		NIH P41 EB001978
Stanley Gu	Andrew Briand		NIH U01 GM104604
Wilbert Copeland	Jean-Marie Bouteiller		NIH R01 GM081070
Jayit Biswas	Maciej H. Swat		NIH R01 GM123032
Totte Karlsson	James A. Glazier		
Caroline Cannistra	Herbert M. Sauro		
Alex Darling			
Nasir Elmi			
Herbert M. Sauro			

References

[1] K. Choi, J. K. Medley, M. König, K. Stocking, L. Smith, S. Gu, and H. M. Sauro, “Tellurium: An extensible python-based modeling environment for systems and synthetic biology,” *Biosystems*, vol. 171, pp. 74–79, Sep. 2018, ISSN: 0303-2647. DOI: 10.1016/J.BIOSYSTEMS.2018.07.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0303264718301254?via%3Dihub.

[2] E. T. Somogyi, J.-M. Bouteiller, J. A. Glazier, M. König, J. K. Medley, M. H. Swat, and H. M. Sauro, “libRoadRunner: a high performance SBML simulation and analysis library.,” *Bioinformatics*, vol. 31, no. 20, pp. 3315–21, Oct. 2015, ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btv363. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/26085503http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC4607739.

[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, ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btp401. [Online]. Available: https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btp401.

[4] K. Choi, L. P. Smith, J. K. Medley, and H. M. Sauro, “phraSED-ML: a paraphrased, human-readable adaptation of SED-ML,” *Journal of Bioinformatics and Computational Biology*, vol. 14, no. 06, p. 1 650 035, Dec. 2016, ISSN: 0219-7200. DOI: 10.1142/S0219720016500359. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/27774871http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC5313123https://www.worldscientific.com/doi/abs/10.1142/S0219720016500359.

[5] M. B. Elowitz and S. Leibier, “A synthetic oscillatory network of transcriptional regulators,” *Nature*, vol. 403, no. 6767, pp. 335–338, 2000, ISSN: 00280836. DOI: 10.1038/35002125.

[6] V. L. Porubsky and H. M. Sauro, “Application of Parameter Optimization to Search for Oscillatory Mass-Action Networks Using Python,” *Processes*, vol. 7, no. 3, p. 163, Mar. 2019, ISSN: 2227-9717. DOI: 10.3390/pr7030163. [Online]. Available: https://www.mdpi.com/2227-9717/7/3/163.

[7] 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 Letters*, vol. 499, no. 3, pp. 230–234, Jun. 2001, ISSN: 00145793. DOI: 10.1016/S0014-5793(01)02562-5.

[8] B. N. Kholodenko, “Negative feedback and ultrasensitivity can bring about oscillations in the mitogen-activated protein kinase cascades,” *European Journal of Biochemistry*, vol. 267, no. 6, pp. 1583–1588, Mar. 2000, ISSN: 00142956. DOI: 10.1046/j.1432-1327.2000.01197.x. [Online]. Available: http://doi.wiley.com/10.1046/j.1432-1327.2000.01197.x.