Statistics and Numerics

Lecture Prof. Dr. Jens Timmer
Exercises Helge Hass, Mirjam Fehling-Kaschek, Joep Vanlier

Exercise Sheet Nr. 11

Exercise 1: The Gillespie algorithm

- In the following exercise, we will attempt to implement the following reaction system:

$$\begin{aligned} S_1 + S_2 &\xrightarrow{k_1} S_3 \\ S_3 &\xrightarrow{k_2} S_2 + S_4 \, , \end{aligned}$$

following the law of mass action and in a way that is suitable for the Gillespie algorithm.

In this exercise, we will try to separate the solver from the actual problem being solved. One method to do this is to specify a lambda function which computes the propensities. A lambda function is defined by specifying lambda, then the function's input arguments and then the result the function is intended to produce. Example (not the correct propensities for the system above):

```
propensityFunction = lambda x,p:[p[0]*x[0]*x[1], p[1]*x[2]]
```

The function can then be called as:

```
x = [1, 1, 1]
p = [1, 1]
result = propensityFunction(x, p)
```

a. Write the lambda function that calculates the propensities for our problem.

b. Write the stoichiometry matrix for this problem. Each row in the stoichiometric matrix represents a reaction. Negative values refer to species which are consumed, while positive values indicate species which are produced. Example:

```
n_rxns = 2
n_states = 4
N = np.zeros([n_rxns,n_states])
N[0,:] = [ -1.0, -1.0,  1.0, 0.0 ]
N[1,:] = [  0.0,  1.0, -1.0, 1.0 ]
```

c. We will first implement integration of the system using deterministic simulation. This will provide us with a reference for what the simulation would look like for large numbers of particles. The following code will simulate the system when provided with a stoichiometric matrix N, propensity function 'propensityFunction' and parameters parsODE. Initial is an array of the initial concentrations of the model components (S_i). The function will then construct an array of differential equations dx from the propensities and N.
Hint: Remember to convert the parameters appropriately between particles and concentrations.
$N_{to\_conc} = 1.0/1.0e-12/1.0e-9/6.022/1.0e23;$
Hint: The ODE integrator odeint can be imported by: from scipy.integrate import odeint.

```
## Deterministic integration
def rhs(y, t, N, fun, pars):
    w = fun(y, pars)
    sizes = np.shape(N)

    dx = N[0,:]*w[0]
    for i in range(1,sizes[0]):
        dx = dx + N[i,:]*w[i]

    return dx

sol = odeint(rhs, initial*N_to_conc, t, args=(N, propensityFunction, parsODE),
      hmax=10, rtol=1e-14, atol=1e-14 )
```

d. Now that we have a reference simulation, write a new function which will solve the problem using the Gillespie algorithm. This function should take the initial condition (number of particles of each species), the stoichiometry of the problem, the propensity function which returns the reaction propensities and the parameters and simulation time.

```
def gillespie(n_s0, N, fun, pars, maxtime, maxsteps)
```

Call this function from the main script.

e. Make the code evaluate the propensities. How can we select the appropriate reaction to perform? (Hint: you can compute a cumulative sum with *np.cumsum*).

f. Implement a mechanism which appropriately selects the reaction to perform according to its propensity and apply it. (Hint: Row or column selection can be done by $X[:,i]$ and $X[i,:]$. Rows can simply be added (provided that you use a numpy array or matrix for storage).

g. Recall the formula to calculate the next reaction time from the lecture. Implement this in your Gillespie simulation routine.

h. Write a loop to simulate over the time steps. Evaluate the propensities, calculate the time increment and store the results after applying the appropriate reaction.

i. For our model system, when is the simulation finished? What will happen to the next reaction time when the step or time threshold is hit? Make sure this does not happen and terminate the simulation appropriately.

j. Simulate the system with this algorithm.

- Simulate the system for a maximum of 20000 steps between 0 and 1000 seconds:
  a) $k_1 = 0.5\,(nMsec)^{-1}$
  b) $k_2 = 0.02\,sec^{-1}$
  c) $V = 1pL$
  d) Initial values: $\#S_1(0) = \#S_2(0) = 10, \;\; \#S_3(0) = \#S_4(0) = 0.$

- Take the average of 20 realizations of the algorithm and compare it with a deterministic simulation of the system. For this purpose, interpolate your solutions to a fixed time vector. Hint: You can use interpolate.interp1d for this which can be imported by:

  ```
  from scipy import interpolate
  ```

  And used as follows:

  ```
  f = interpolate.interp1d(t_simulated, y_sim, kind='zero' )
  y_desired = f(t_desired)
  ```

  You can also copy the function `averagedGillespie` from the online solution.

- Bonus question for the interested. Implement the following system:

$$\begin{aligned} S_2 &\xrightarrow{k_1} 2S_2 \\ S_1 + S_2 &\xrightarrow{k_2} 2S_1 \\ S_1 &\xrightarrow{k_3} \emptyset \\ \emptyset &\xrightarrow{k_4} S_1 \end{aligned}$$

  Simulate this system for a maximum of 2000000 steps between 0 and 100 seconds and compare it to its ODE equivalent:

  a) $k_1 = 1.0 sec^{-1}$
  b) $k_2 = 0.005 (nM sec)^{-1}$
  c) $k_3 = 0.5 sec^{-1}$
  d) $k_4 = 0.3 nM sec^{-1}$
  e) Initial values: $\#S_1(0) = 100, \#S_2(0) = 50$.

  What do you notice?