***Part 1***

***Stochastic models – the story so far….***

We built a toy system to model protein degradation with green and red balls in a bag.

* green balls represent function protein molecules
* red balls represent non-functional degraded protein molecules

We modelled the collisions of an enzyme that will degrade any protein it “meets”.

**Model procedure**

At intervals of 10 seconds we take a ball from the bag simulating the enzyme encountering a protein in the system.

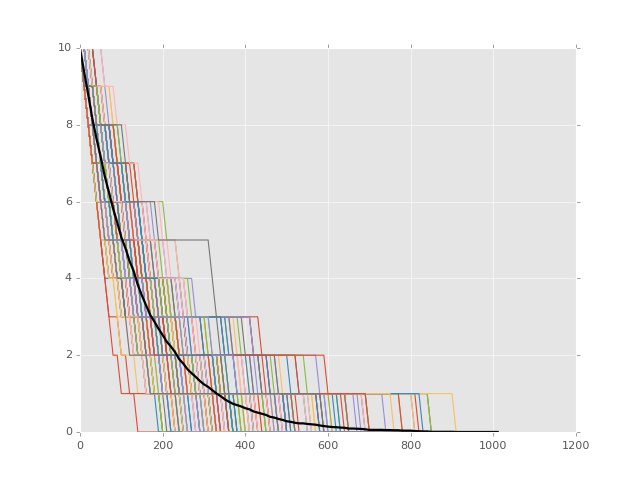
* If it is green the protein 'degrades' and is replaced it with a red ball.
* If it is red we put it back.

The code used to run this is below (and can be found in the Dropbox folder as s7\_code.py).

1. **from** myreport **import** html\_report
2. **import** random
3. **from** matplotlib **import** pyplot as plt
4. **import** numpy as np
5. plt.style.use('ggplot')
7. # PUT ALL FUNCTIONS HERE
9. **def** draw\_ball(G,R):
10. pG=float(G)/float(G+R)
11. rand\_num=random.random()
12. **if** rand\_num<pG:
13. **return** 'G'
14. **else**:
15. **return** 'R'
17. **def** simulate\_system():
18. #balls
19. G=10
20. R=5
21. T=G+R
23. #time
24. t=0
26. G\_record=[]
27. R\_record=[]
28. t\_record=[]
30. G\_record.append(G)
31. R\_record.append(R)
32. t\_record.append(t)
34. **while** t<=1000:
35. # increment time
36. t=t+10
37. **if** G+R!=0:
38. # select ball from bag
39. ball=draw\_ball(G,R)
40. # update number of balls
41. **if** ball=='G':
42. G=G-1
43. R=R+1
44. # record system state
45. G\_record.append(G)
46. R\_record.append(R)
47. t\_record.append(t)
48. **return** (G\_record, R\_record, t\_record)
50. #results= simulate\_system()
51. #G\_record, R\_record, t\_record = results
52. #plt.plot(t\_record, G\_record, 'g-')
53. #plt.plot(t\_record, R\_record, 'r-')
54. #plt.show()
56. my\_runs=[]
57. t\_record=None
58. **for** i **in** range(200):
59. results= simulate\_system()
60. G\_record, R\_record, t\_record = results
61. my\_runs.append(G\_record)
63. **print** "length t\_record:",len(t\_record)
64. **print** "length my\_runs:",len(my\_runs)
65. **print** "length my\_runs[0]:",len(my\_runs[0])
67. plt.close('all')
68. my\_runs=np.array(my\_runs)
69. my\_runs=np.transpose(my\_runs)
70. fig1=plt.figure()
71. ax=fig1.add\_subplot(1,1,1)
72. ax.plot(t\_record, my\_runs, '-')
74. av\_run=np.average(my\_runs,axis=1)
75. # axis=0 averages colums
76. # axis=1 averages rows
77. ax.plot(t\_record, av\_run, 'k-', lw=2)
78. fig1.show()
80. fig2=plt.figure()
81. ax=fig2.add\_subplot(1,1,1)
82. ax.hist(my\_runs[10,:], bins=np.arange(-0.5,11.5,1.0))
83. fig2.show()

86. # now write results
87. myreport=html\_report("s6\_task1.html")
88. myreport.add\_subheading('Decay trajectories')
89. myreport.add\_figure(fig1)
90. myreport.add\_subheading('Distribution 100 runs t=100')
91. myreport.add\_figure(fig2)
93. myreport.add\_subheading('Python Code')
94. myreport.add\_source(\_\_file\_\_)
95. myreport.write()
96. myreport.view()

The following figure shows a plot resulting from 200 trajectories. The averaged trajectory is marked with a black line:



*Plot of # green balls vs time.*

*200 trajectories and averaged*

*trajectory shown (black).*

It is apparent that the average trajectory looks like an exponential decay process, which we might expect for the deterministic model of protein degradation:

Where:

k = rate of degradation for an individual protein

= rate of collisions between the enzyme and an individual protein

We can calculate this, by noting that this rate can be found using:

rate that a given protein collide with enzyme

= rate of collisions probability collision involves a given protein molecule

In this case we have:

rate of collisions = 1/10sec

*(collisions every 10 seconds)*

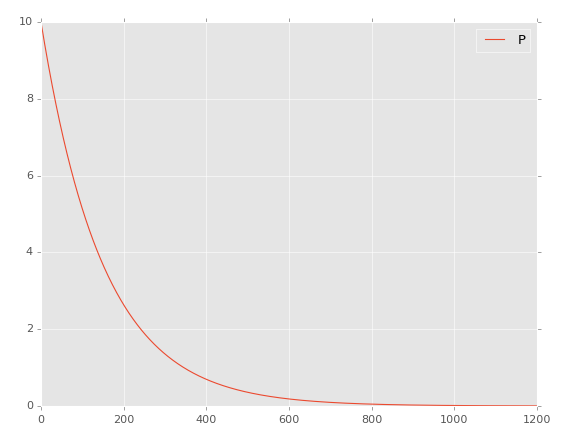
probability that enzyme to collide with a given protein = 1 / 15

( *this is 1 / total number of proteins*)

This system can be simulated using odeint:

1. **import** numpy as np
2. **import** matplotlib.pyplot as plt
3. **from** scipy.integrate **import** odeint
5. k=(1.0/10.)\*(1.0/15.0)
6. P=10.0
7. s0=(P,)
8. params=(k,)
10. # as we only have 1 species / parameter
11. # we need to add commas to force them
12. # to be packed and unpacked as tuples
14. **def** sdot(s,t,params):
15. k,=params
16. P,=s
17. dP = -k\*P
18. **return** (dP,)
20. plt.close("all")
22. fig1 = plt.figure()
23. ax=fig1.add\_subplot(1,1,1)
24. t\_max=1200
25. t\_obs=np.linspace(0,t\_max,1201)
26. s\_obs=odeint(sdot,s0,t\_obs,args=(params,))
27. ax.plot(t\_obs, s\_obs, '-',label='P')
28. ax.legend()
29. fig1.show()

Output of above code:



*Example plot showing result of deterministic model.*

*# proteins vs time*

***Comparing results of the stochastic and deterministic models***

To compare the results of our stochastic and deterministic model it is informative to look at the following:

*i) the average trajectory for stochastic model vs trajectory of deterministic model.*

In this case in the limit that we average over a large number of trajectories from our stochastic model we observe similar behaviour to the deterministic model.

**Note this result does not hold generally. In some cases the average trajectory from the stochastic system does not correspond to the behaviour of a deterministic model for the same system.**

*ii) the behaviour of both systems in the limit that the model involves large numbers*

In the following task we will look at the behaviour of the stochastic system when we increase the numbers in the system to large values.

***Tasks***

**1a) Run a single simulation (not 200!) of the stochastic model with the following changes to the parameters:**

Number of green balls: 10 changed to 1000

Number of red balls: 5 changed to 500

Collision time: 10 sec changed to 0.1 sec

**1b) Compare this to the result of the deterministic model with the same changes made .**

Hint:

Recall that degradation rate: k = 1/[collision time] x 1/[total number of balls]

*\*\*\* Save both plots (e.g. in a word document) before continuing \*\*\**

You should find that in the limit of large numbers the stochastic system behaves in the same way as the deterministic system.

***This result holds in general.***

***Part 2***

***Analysis of the stochastic distributions.***

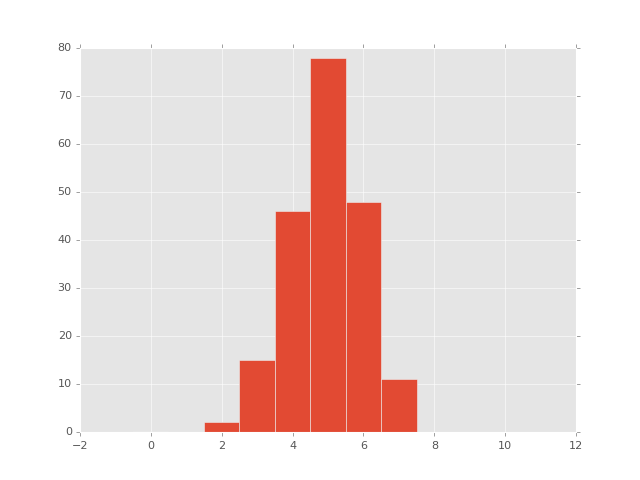
Revert your code to its original form (10 green 5 red balls and 10s collision interval.)

Last week we looked at how we could plot the distribution of the results over the ensemble of simulation runs. We looked at how many green balls were left at time t=100 seconds. (i.e. the 10th step of the simulation):

1. fig2=plt.figure()
2. ax=fig2.add\_subplot(1,1,1)
3. ax.hist(my\_runs[10,:], bins=np.arange(-0.5,10.6,1.0))
4. fig2.show()

Note that matplotlib will automatically calculate histogram bins, but in this case we know the values will be integer values between 0 and 10, so we set up bins accordingly:

-0.5 to 0.5, 0.5 to 1.5, 1.5 to 2.5 …. 9.5 to 10.5,

 we set an upper limit of 10.6 because arange goes up in increments stopping when it is equal or greater to the upper limit.

*Histogram showing the distribution of G*

*(# green balls) at time t=100 seconds*

We see that in this case after 100 seconds:

- some runs had as few as 2 green balls left

- most runs had 5 balls left

- some runs had as many as 7 balls left

***Tasks***

**2a) Adjust the code to produce the distribution at t=500seconds.**

**How has the distribution changed?**

**Compare this to the trajectory plot – are they in agreement?**

Making these histogram plots is equivalent to taking vertical “slices” of the trajectory plot at a particular time point.

Alternatively we might take a horizontal “slice”. For example we might want to plot the time taken in each run for the number of green balls in the system to fall from 10 to 5, and hence take a measure of the system ‘half-life’.

**2b) Write additional code to determine the measured “half-life” in each run, and histogram the results.**

Hints.

1. Prior to the run loop create an empty list: hlife\_results.
2. Inside the loop, after each simulation, create a loop over the simulation results to finds the index i of the first G\_record which has a value of 5 (or less).
3. Store the corresponding time (t\_record[i]) which is the half life measured in this run.
4. Ensure that you only store one half-life value from each run (i.e. after finding the first value that has fallen to 5, exit the loop over G\_record values.
5. Histogram the results at the end of the code using something like:

fig3=plt.figure()

ax3=fig3.add\_subplot(1,1,1)

ax3.hist(hlife\_results)

fig3.show()

**2c) Compare the mean half-life found averaged over the 200 runs to that predicted by the deterministic model:**

***Part 3***

***The Gillespie Algorithm***

In the above model we used a constant 10s time interval between events.

In general the interval between events in a system will vary as the system evolves and the populations of reactants in each process changes.

The Gillespie algorithm assumes that the system can be modelled as a set of Poisson processes (i.e. a series of independent events which each event having a known probability of occurrence).

The Gillespie algorithm applies the following procedure to simulate a system:

***1. Set up the system in its initial state.***

***2. Simulate the system forward in time on an event by event basis:***

1. ***Construct a probabilistic model for the rates of all processes that might occur, based on the current state of the system.***
2. ***Based on this model randomly generate the time interval, which will elapse before an event of any type occurs.***
3. ***Based on the relative probabilities of each process occurring, randomly select the type of event that occurs next.***
4. ***Calculate the new state of the system after this event has occurred.***

***3. Repeat step (2) until we reach the final simulation time (or the system moves to a state in which no further events are possible)***

To prepare to simulate a system using the Gillespie algorithm we must therefore make the following calculations:

***i) Model the event rates for each process***

The first stage in formulating the stochastic model is to write down the processes that take place, along with their rates. In the above model we have two types of process or events:

1) Collision between enzyme and protein (green ball)

G + E R + E (degradation)

2) Collision between enzyme and degraded protein (red ball)

R + E R+ E (no change)

If we assume mass action kinetics for the rates we can write them as:

Rate (G + E R + E ) = k1 G E

Rate (R + E R+ E ) = k2 R E

We then assume that the E is equally likely to collide with an individual G or R type protein, so k1=k2 and we can combine the enzyme concentration into a shared rate constant k:

Rate (G + E R + E ) = k G

Rate (R + E R+ E ) = k R

***ii) Model the event-to-event interval***

In order to randomly generate event-to-event intervals the algorithm assumes that if the rate of events is ***k*** then the interval time between two events can be described by an exponential distribution:

distribution of intervals = C e –kt where C is a normalisation constant

This means we can model the time interval between two events occurring by generating a random number the above exponential distribution.

However in the algorithm we need to consider the interval between two events in general (not events in general). To do this we need to use the overall rate for events of any type to occur:

ratetotal = sum of rates over all processes

= kG + kR

***iii) Model the probability for events to be of a particular type***

The next step is to write the probability distribution; given an event has occurred what is the probability is it due to each process.

e.g.

Suppose we have many more R than G molecules and rate for (R+E R+E ) is 9 times greater that the rate for (G+E R+E ).

Then the chance that when an event occurred it is of type (R+E R+E ) is 9 times greater,

e.g. if 100 events occurring

~90 will be (R+E R+E )

~10 will be (G+E R+E )

In general to find the probability of event type “X”:

P(when an event occurs it is of type X) = rate(process X) / rate (total)

i.e. P(G+E R+E ) = kG / (kG + kR) = G / N

P(R+E R+E ) = kR / (kG + kR) = R / N

where N=G+R

***iv) Update the state of the system***

For example suppose at a given point in the simulation the conditions are:

time t=100.4

# green G=12

# red R=3

Following the above steps suppose we found that the event-to-event interval was randomly generated to be 13.5, and the event type was enzyme-and-green collision (such that green ball is exchanged for a red). In this case we update the system by incrementing the time by 13.5 seconds, and update the ball counts accordingly:

time t=113.9

# green G=11

# red R=4

If we generate the next event and find a time interval of 5.2s and the event type as enzyme-and-red collision (such that no change occurs) we would then update the system state to:

time t=119.1

# green G=11

# red R=4

***Coding the Gillespie Algorithm***

To code our algorithm we can write code that is similar in form to our fixed interval model. However instead of progressing forward through time using a fixed time step, we progress forward through time event by event.

The event-to-event intervals drawn randomly from an exponential function that has rate constant according to the total rate of all possible events.

**3a) Code the Gillespie Algorithm for our model.**

To get you started a template file is provided in the file:

gillespie\_template.py

It contains gaps (marked with three dots … ) which you have to fill using the above information.

**3b) When you have done this increase the number of runs to 200, and compare the overlaid trajectories plot to the one in this document. How does it differ?**

Hint. As the runs all have different lengths you need to plot each result using a loop like:

n\_runs=len(t\_runs)

for i in range(n\_runs):

ax.plot(t\_runs[i], G\_runs[i], '-')

***Part 4.***

***Analysing the results***

Note that when using the Gillespie algorithm the simulation results contain an observation made for every event. For each run the timing and number of events differs (e.g. check the length of t\_runs[0] and t\_runs[1])

This means we cannot directly use the raw results to produce average trajectories and distributions.

To be able to do this we need to resample the runs, so that observations of the variables are made at regular intervals.

To do this you can add the following function into your code:

1. **def** convert\_to\_intervals(t\_obs,s\_obs,interval,t\_start,t\_final):
2. T=t\_start
3. t\_new\_obs=[]
4. s\_new\_obs=[]
5. i=0
6. ti=t\_obs[i]
7. si=s\_obs[i]
8. **while** T<t\_final:
9. **while** ti<T **and** i<len(t\_obs)-1:
10. i+=1
11. ti=t\_obs[i]
12. si=s\_obs[i]
13. t\_new\_obs.append(T)
14. s\_new\_obs.append(si)
15. T=T+interval
16. **return** (t\_new\_obs,s\_new\_obs)

This takes a set of observation arrays t\_obs and s\_obs and resamples them over the period t\_start to t\_end, at the specified interval

***Tasks***

**4a) Use this function to resample the simulation results (0s to 1000s with an interval of 1s),**

To do this add the function in the following way:

G\_record, R\_record, t\_record = results

t\_record,G\_record=convert\_to\_intervals(t\_record,G\_record,…

This means we convert to use identical time intervals before adding the simulating results into the G\_runs and t\_runs collections.

**b) When you have done this recreate the following two plots:**

1. **200 trajectories along with the average trajectory**
2. **the distribution of G for the 200 runs after time=100 seconds**

To do this you can modify the code you used in the first example:

1. t\_record=t\_runs[0]
2. G\_runs=np.array(G\_runs)
3. G\_runs=np.transpose(G\_runs)
4. fig1=plt.figure()
5. ax=fig1.add\_subplot(1,1,1)
6. ax.plot(t\_record, G\_runs, '-')
8. av\_run=np.average(G\_runs,axis=1)
9. # axis=0 averages colums
10. # axis=1 averages rows
11. ax.plot(t\_record, av\_run, 'k-', lw=2)
12. fig1.show()
14. fig2=plt.figure()
15. ax=fig2.add\_subplot(1,1,1)
16. ax.hist(G\_runs[...,:], bins=np.arange(-0.5,11.5,1.0))
17. fig2.show()

**c) Compare this to the plots using a fixed time interval. What do you find?**