CS 237 Lab Six: Poisson Process Simulation

Due date: PDF file due Thursday October 18th @ 11:59PM (10% off if up to 24 hours late) in GradeScope

General Instructions

Please complete this notebook by filling in solutions where indicated. Be sure to “Run All” from the Cell menu before submitting.

You may use ordinary ASCII text to write your solutions, or (preferably) LaTeX. A nice introduction to LaTeX in Jupyter notebooks may be found here: http://data-blog.udacity.com/posts/2016/10/latex-primer/

As with previous homeworks, just upload a PDF file of this notebook. Instructions for converting to PDF may be found on the class web page right under the link for homework 1.
Lab 06 Instructions

In this lab we will again simulate a random experiment, but in this case it is somewhat more complex than our previous simulations. We will simulate the behavior of a simple queueing system, which has tasks arriving for service, waiting in a FIFO queue, receiving service, and leaving the system:

```
FIFO Queue
-----------------------------
tasks arrive ==> | | | | | | | =>> Server ==> tasks exit the system
```

You should be familiar in a general way with this idea, since queues are commonly used to schedule programs for execution, packets to processed in a network, and probably you have waited at Starbucks at some time in your life!

Here is the standard terminology we will use:

- A **server** is anything that performs some useful service, such as executing a program in a computer system;
- A **task** is anything that requires service;
- The **service time** of a task is the amount of time the task needs to be serviced; and
- The **queue length** is...wait for it... the number of tasks in the queue.

In this lab, we will consider a simulation of Professor Snyder’s daily processing of emails. Hence,

- Tasks are **email messages**;
- The server is **Snyder reading his email**; and
- The queue is **Snyder’s inbox**, with the assumption that he answers emails in the order he receives them.

We will make the following assumptions in this simulation, which will involve only the Poisson and Geometric distributions:

- Tasks are represented simply by an **integer** (the number of minutes it would take to answer the email);
- The queue is just a **list of integers**;
- The simulation will run **num_days** number of days;
- Tasks (emails) arrive each day following a **Poisson distribution with rate parameter lam = the number of tasks arriving on average in a given day**;
- The service time of tasks follows a **geometric distribution with probability p**, so $1/p$ is the average number of minutes it takes to answer an email (but remember the actual number of minutes for each task is an integer);
- Snyder devotes one hour minimum a day to answering email, and continues to answer emails in order of arrival in his Inbox; he will start to answer an email if he has spent less than one hour so far, and will always finish answering an email. For example, if he has answered 20 emails and has spent 59 minutes doing so, and the next email will take 10 minutes, then he answers it, running over his hour by 9 minutes. But then he would stop because he is out of time. In general, of course, Snyder will spend slightly more than one hour on his email.

A pseudo-code version of the algorithm used to run the simulation is as follows:
# basic parameters of the simulation

\[ \text{lam} = ... \]  
\text{avg\_service} \]  
\[ p = 1/\text{avg\_service} \]  
\[ \text{num\_days} = ... \]  

Initialize the queue

for day in range(num\_days):

    # process daily arrivals
    \[ \text{num\_arrivals} = \text{a random variate from Poi(lam)} \]  
    for k in range(num\_arrivals):
        \[ \text{task} = \text{a random variate from G(p)} \]  
        dequeue the task
In [1]: # Here are some imports which will be used in code that we write for CS 237

# Jupyter notebook specific

from IPython.display import Image
from IPython.core.display import HTML
from IPython.display import display_html
from IPython.display import display
from IPython.display import Math
from IPython.display import Latex
from IPython.display import HTML

# Imports potentially used for this lab

import matplotlib.pyplot as plt  # normal plotting
import math
from random import seed, random, uniform, randint
import numpy as np
from collections import Counter

%matplotlib inline

# Calculating permutations and combinations efficiently

def P(N, K):
    res = 1
    for i in range(K):
        res *= N
        N = N - 1
    return res

def C(N, K):
    if (K < N/2):
        K = N-K
    X = [1]*(K+1)
    for row in range(1,N-K+1):
        X[row] *= 2
        for col in range(row+1,K+1):
    return X[K]

# Useful code from lab 01

# This function takes a list of outcomes and a list of probabilities and
# draws a chart of the probability distribution.

def draw_distribution(Rx, fx, title='Probability Distribution for X'):
    plt.figure(figsize=(10, 6))
    plt.bar(Rx, fx, width=1.0, edgecolor='black')
    plt.ylabel("Probability")
    plt.xlabel("Outcomes")
    if (Rx[-1] - Rx[0] < 30):
        ticks = range(Rx[0],Rx[-1]+1)
    plt.xticks(ticks, ticks)
    plt.title(title)
    plt.show()

# This function takes a list of outcomes, calculates a histogram, and
# then draws the empirical frequency distribution.

def show_distribution(outcomes, title='Empirical Probability Distribution'):
    num_trials = len(outcomes)
    print(num_trials)
    Rx = range( int(min(outcomes)), int(max(outcomes))+1 )
    print(Rx)
    freqs = Counter(outcomes)
    print(freqs)
    fx = [freqs[i]/num_trials for i in Rx]
Problem Zero: Generating Poisson Random Variates

This provides a random variates generator for the Poisson distribution, based on the ideas from last week's lab. However, a subtlety is added: it allows you to specify the rate parameter lambda, but only calculates the CDF when lam is changed. This is MUCH more efficient, since you don't have to calculate the CDF every time you ask for a random variate (which could be as much as 100,000 times!).

Nothing to do here, but uncomment the last two lines and make sure you understand what these functions do.
Problem One

(a) Implement the queue using the (ugly) Python OO paradigm.

(b) Test the queue

```
In [4]: Q = Queue()
Q.enqueue(6)
Q.enqueue(9)
print(Q.size())
print(Q.isEmpty())
print(Q.show())
print(Q.dequeue())
print(Q.dequeue())
print(Q.size())
print(Q.isEmpty())
```

2
False
>[9, 6]>
6
9
0
True

Problem Two

(a) Write the main simulation code, using the pseudo-code given above.

```
In [5]:
```

(b) Test the code
In [7]:

    Optional question to ponder (not to be graded): Why do you think there is a huge spike for a queue length of 0?

Problem Three

Now we will investigate the behavior of the system as we change the parameter lam which represents the average number of emails that arrive each day. We will keep the parameter p fixed.

The thing to keep in mind is the relationship between lam * avg_service (the average cumulative service requests per day) and 60 (the allocated service time).

(a) Set lam = 20 (so lam * avg_service << 60). Run the experiment from the previous problem.
In [8]: test_run(20, p, num_days)

1000
range(0, 9)
Counter({0: 965, 1: 16, 2: 5, 4: 4, 8: 3, 3: 3, 5: 2, 7: 2})
[0.965, 0.016, 0.005, 0.003, 0.004, 0.002, 0.0, 0.002, 0.003]

(b) Set lam = 40 (so lam * avg_service >> 60). Run the experiment from the previous problem.

If you have trouble running this one, try 35 and a smaller number of days.
In [9]: `test_run(40, p, num_days)`
(c) Set $\lambda = 30$ (so $\lambda \times \text{avg. service} = 60$). Run the experiment from the previous problem.

Mean queue length = 4832.202

Out[9]: 4832.2020000000002
Problem Four

Try a few more examples of different values for lam, paying careful attention to what happens to the queue distribution, and the average queue length, as lam approaches and exceeds 30. You may have to set the number of days to something smaller, but probably you need at least 100 to get reasonable results. Do not change avg_service. Then answer the following questions:

(d.1) What does the queue distribution look like in general when lam << 30? Why do you think this happens?

(d.2) What does the queue distribution look like in general when lam >> 30? Why do you think this happens?

(d.3) Why do you think the queue distribution changes so dramatically as lam approaches and exceeds 30?

Answers:

(d.1) The queue is mostly empty, or very short, because Snyder has plenty of time to answer all the emails, on average.

(d.2) In this case, Snyder does not have enough time to finish each days' email requests, so the queue gets longer and longer.

(d.3) When the mean cumulative requested time hits the mean service time available, that's when it blows up.
Problem Five

Run a series of experiments where lam is tested in a range of values around 30, noting that lam can be a floating-point number. Generate a graph showing the mean queue length as a function of lam. You should see a linear increase in the mean queue length when you get to around 30.5, something like this

/  
/  
/  
------------  

You may adjust the num_days parameter and try different ranges, starting with the one given. The main point is to see the shape of the curve.

In [12]:
Out[12]: [<matplotlib.lines.Line2D at 0x11b040eb8>]

![Graph showing the mean queue length as a function of lam.](image)