Queueing Theory: Queues in continuous time

EBB074A05

Nicky D. van Foreest

2020:12:17

1 General info

This file contains the code and the results that go with this youtube movie: `https://youtu.be/h1OTvdLs9ik`

1.1 TODO Set theme and font size

Set the theme and font size so that it is easier to read on youbute

1.2 Load standard modules

```
import numpy as np
import matplotlib.pylab as plt
from matplotlib import style

style.use('ggplot')

np.random.seed(3)
```

2 Computing waiting times

You should read the relevant section of my queueing book to understand what it going on here. It's very easy, but without background a bit cryptic (I believe).

2.1 Interarrival times

```
labda = 3
X = np.random.exponential(scale=labda, size=10)
print(X)
```

```
[2.40084716 3.69452354 1.03129621 2.14512092 6.70329244 6.79855956
 0.40260163 0.69671515 0.15851674 1.74379707]
```

2.2 Arrival times

```
A = X.cumsum()
print(A)
```

```
[ 2.40084716  6.0953707   7.12666691  9.27178782 15.97508026 22.77363983
 23.17624146 23.8729566  24.03147334 25.77527041]
```

Ex 2.1. Why do we generate first random interarrival times, and use these to compute the arrival times? Why not directly generate random arrival times?

Actually, I am not happy about this construction. In an exercise below I ask you to explain why.

```
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
print(A)
```

```
[ 0.          2.40084716  6.0953707   7.12666691  9.27178782 15.97508026
 22.77363983 23.17624146 23.8729566  24.03147334 25.77527041]
```

This is better!

2.3 Service times

```
mu = 1.2 * labda
S = np.random.exponential(scale=mu,size=len(A))
print(S)
```

```
[0.10919375 2.19721993 3.77056631 1.17505899 4.06007571 3.21733717
 0.08738687 2.94616653 1.08034342 1.93073916 1.20028334]
```

Note, `S[0]` remains unused; it corresponds to job 0, but there is no job 0.

2.4 Departure times

```
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k-1], A[k]) + S[k]

print(D)
```

```
[ 0.          4.59806709  9.86593702 11.040996   15.10107171 19.19241743
 22.86102669 26.12240799 27.20275141 29.13349057 30.33377391]
```

Ex 2.2. Explain now why I was unhappy with the first construction of the arrival times.

2.5 Sojourn times

```
J = D - A
print(J)
```

```
[0.         2.19721993 3.77056631 3.9143291  5.82928389 3.21733717
 0.08738687 2.94616653 3.32979481 5.10201723 4.5585035 ]
```

2.6 Waiting times

```
W = J - S
```

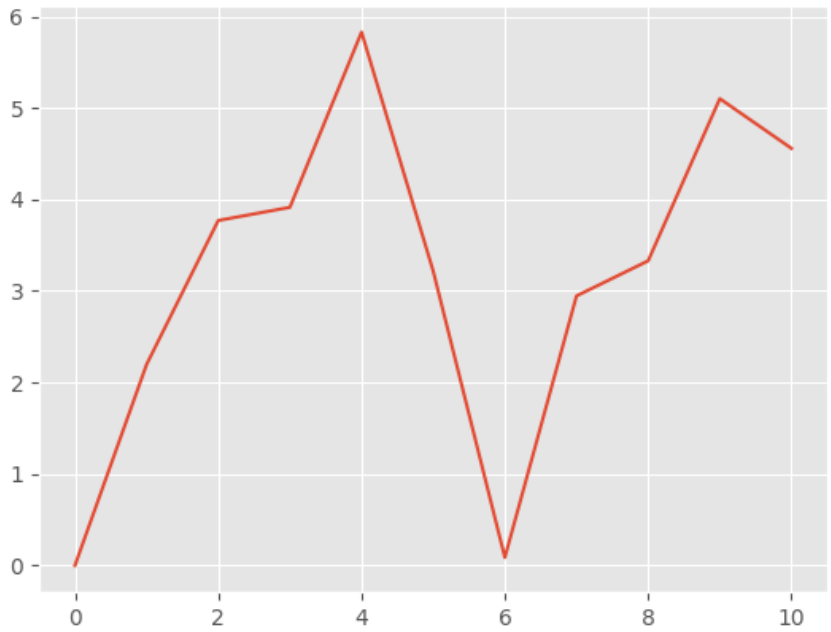
Ex 2.3. Why does the code above compute the waiting times?

2.7 KPIs and plot

```
print(J.mean(), J.std())
```

```
3.177509575645523 1.7638308028443408
```

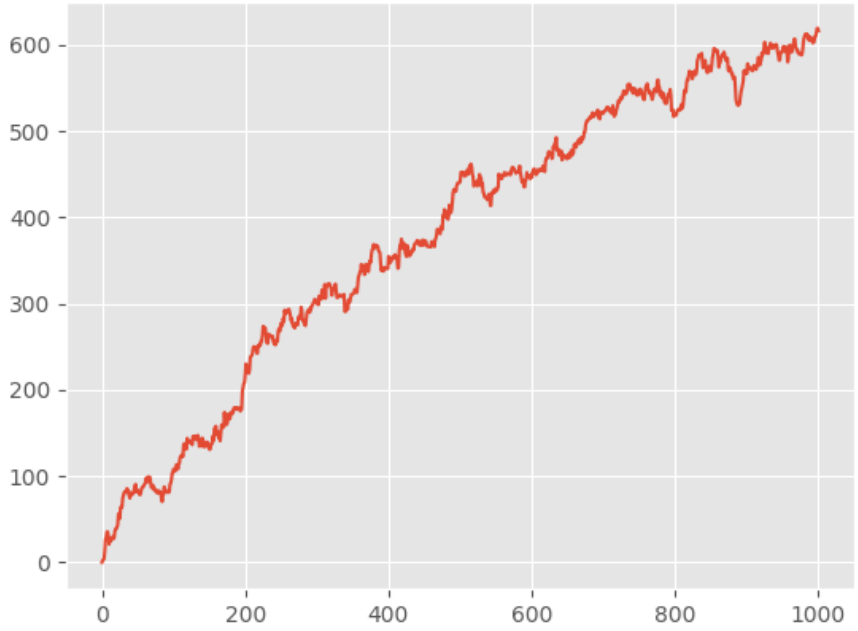
```
plt.clf()
plt.plot(J)
plt.savefig("wait.png")
"wait.png"
```



2.8 Is the queue stable?

Let's do a longer simulation

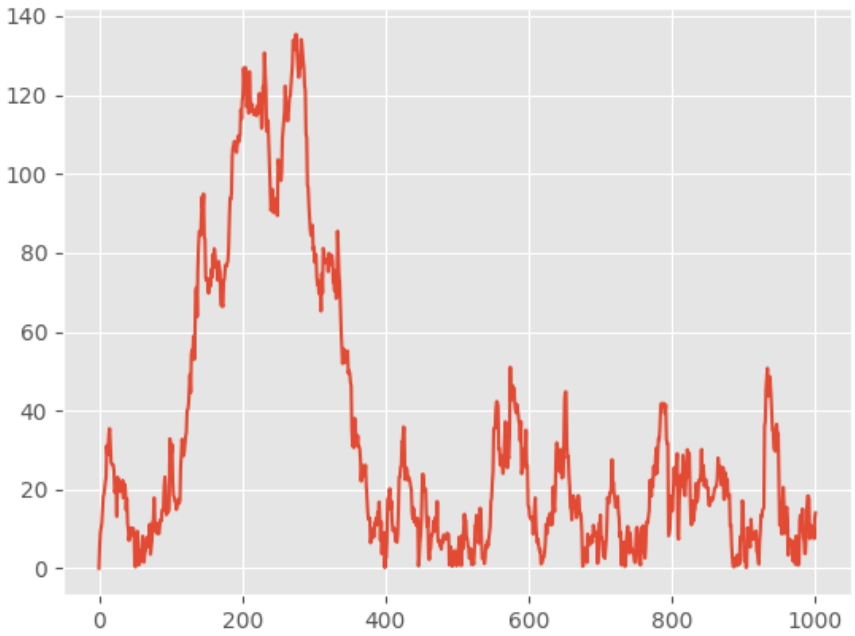
```
num = 1000
labda = 3
X = np.random.exponential(scale=labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
mu = 1.2 * labda
S = np.random.exponential(scale=mu,size=len(A))
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k-1], A[k]) + S[k]

J = D - A

plt.clf()
plt.plot(J)
plt.savefig("wait2.png")
"wait2.png"
```



It's increasing. Let's try for another mu.

```
num = 1000
labda = 3
X = np.random.exponential(scale=labda, size=num)
```

```
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
mu = 0.9 * labda  # changed 1.2 to 0.9
S = np.random.exponential(scale=mu,size=len(A))
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k-1], A[k]) + S[k]

J = D - A

plt.clf()
plt.plot(J)
plt.savefig("wait3.png")
"wait3.png"
```



Ex 2.4. Explain why the new choice for μ is more interesting.

Ex 2.5. Compute the total busy time of the server, and derive from this the total idle time.

Ex 2.6. Here is a hard exercise, and a challenge. Find a way to compute the busy times and the idles times and characterize the emperical distribution of each of these random variables. Note, a busy time starts when a job arrives at an empty system and it stops when the server becomes free again. An idle period starts when a job leaves an empty system behind and it stops when a new job arrives.

2.9 Queue length

We have the waiting times, but not the number of jobs in queue. How to compute the number of jobs in queue? We walk backwards!

```
num = 10
X = np.random.exponential(scale=labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
mu = 0.9 * labda  # changed 1.2 to 0.9
S = np.random.exponential(scale=mu,size=len(A))
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k-1], A[k]) + S[k]

L = np.zeros_like(A)
for k in range(1, len(A)):
    l = k - 1
    while D[l] > A[k]:
        l -= 1
    L[k] = k - l

print(L)
```

```
[0. 1. 1. 2. 3. 2. 3. 4. 3. 1. 1.]
```

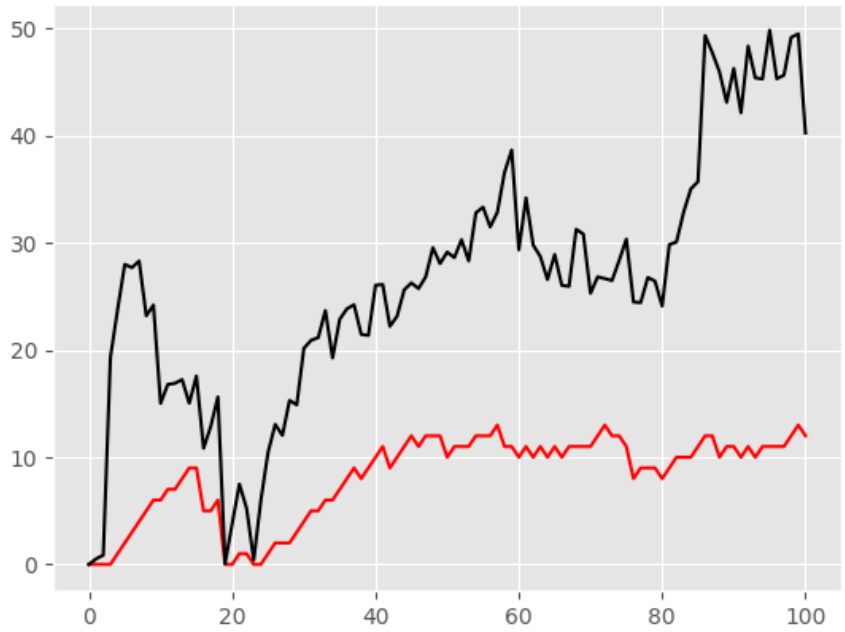
Ex 2.7. Explain the algorithm in your own words. Are you sure it's correct?

Ex 2.8. Are you sure it's correct? Explain in words how you would test this code. (If you can come with python code to test this, then I prefer that. However, explain how your test works, and include the code.)

A longer run.

```
num = 100
X = np.random.exponential(scale=labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
mu = 0.9 * labda  # changed 1.2 to 0.9
S = np.random.exponential(scale=mu,size=len(A))
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k-1], A[k]) + S[k]

W = D - A

L = np.zeros_like(A)
for k in range(1, len(A)):
    l = k
    while D[l] > A[k]:
        l -= 1
    L[k] = k - l - 1

plt.clf()
plt.plot(L, color='red')
plt.plot(W, color='black')
```

```
plt.savefig("wait4.png")
"wait4.png"
```



2.10 Efficiency

Ex 2.9. The above procedure to compute the number of jobs in the system is pretty inefficient. Why is that so? (Can you explain a better (more efficient) way? Again, if you can provide more efficient python code, please do so. However, there is no obligations, and you also don't get extra kudos for it. It's just the challenge, and the reward.)

3 Multiserver queue in continuous time

```
c = 3
N = 10

one = np.ones(c, dtype=int)  #  vector with ones

X = np.ones(N + 1, dtype=int)
S = 5 * np.ones(N, dtype=int)
w = np.zeros(c, dtype=int)

for k in range(1, N):
    s = w.argmin()  # server with smallest waiting time
    w[s] += S[k]  # assign arrival too this server
    w = np.maximum(0, w - X[k + 1] * one)

print(w)
```

```
[6 7 8]
```

4 A bunch of exercises

Ex 4.1. Change the code for the multi-server such that the individual servers have different speeds. For this, change the vector $(1, 1, 1)$ to e.g. $(0.5, 2, 0.5)$ to give the second server twice as much capacity as and the other two half the capacity. Like this the total available capacity remains the same. What happens to the mean and std of the waiting times? (I suspect that the mean stays the same, but I don't know what happens to the variance.)

Ex 4.2. Once you researched the previous exercise, provide some consultancy advice. Is it better to have one fast server and several slow ones, or is it better to have 3 equal servers? What gives the least queueing times and variance? If the variance is affected by changing the server rates, explain the effects based on the intuition you can obtain from Sakasegawa's formula.

Ex 4.3. As a test, set the vector of ones to $(1, 0, 0)$. Like this we have reduced our multi-server queue to a single-server queue. Change the arrival rate to something decent (so that the queue does not explode, i.e, the system is stable again). Then do a few simulations to check whether everything works as it should.

Once again, observe that such 'dumb' corner cases are necessary to test code. As a general rule, it is hard to invent a test that is 'too' dumb. In fact, it has happened quite a few times that I tested on things I was sure not to do wrong, but I still managed to f* things up. It's quite amazing how many mistakes we human beings make, all the time. Hence, a bit of paranoia is a good state of mind when it comes to coding.

Ex 4.4. And now we have assembled all these ideas, can you write a computer program to let a server fail for a certain time? Describe in words how would you do this. Better yet (but again, no obligation), write code to simulate such a case. With this you can analyze the quality of the models with server interruptions we developed in the queueing book.