Queueing theory assignment: Simulation of Queueing processes in continuous time

EBB074A05

Nicky D. van Foreest

2022:01:19

1 Computing waiting times

Here we just follow the steps of the queueing book to construct a single server FIFO queue in continuous time and compute the waiting and sojourn times.

1.1 Load standard modules

We need the standard libraries for numerical work and plotting.

Python Code

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

np.random.seed(3)
```

1.2 Inter-arrival times

Simulate random interarrival times that are $\sim \text{Exp}(\lambda)$, with $\lambda = 3$. First I take just three jobs, so that I can print out all intermediate results and check how things work. Once I am convinced about the correctness, I run a simulation for many jobs.

Python Code

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

Here is an important check (I always forget the meaning of λ when I provide it to the simulator)

Python Code

```
num = 100
labda = 3
X = np.random.exponential(scale=labda, size=num)
print(X.mean())
```

Ex 1.1. Explain that `scale=labda` sets the interarrival times to 3, but that in our queueing models, λ should correspond to the arrival rate. Why is the code below in line with what we want?

Python Code

```
num = 3
labda = 3
X = np.random.exponential(scale=1/labda, size=num)
```

1.3 Arrival times

Ex 1.2. Why do we generate first random inter-arrival times, and use these to compute the arrival times? Why not directly generate random arrival times?

Python Code

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

Check the output to see that the arrival time of job 0 is $A_0 > 0$. But I want time to start at time $A_0 = 0$. Here is the trick to achieve that.

Python Code

```
X[0] = 0
A = X.cumsum()
print(A)
```

This is better!

Ex 1.3. Why can we remove `X[0]` without fundementally changing the analysis?

1.4 Service times

We have arrival times. We next need the service times of the jobs. Assume they are $\sim \text{Exp}(\mu)$ with μ somewhat larger than λ. (Recall this means that jobs can be served faster than that they arrive.)

Python Code

```
mu = 1.2 * labda
S = np.random.exponential(scale=1/mu,size=len(A))
S[0] = 0
print(S)
```

Note, `S[0]` remains unused; it should correspond to job 0, but we neglect this job 0 in the remainder.

Ex 1.4. Why do I use `size=len(A)` in the definition of `S`?

Ex 1.5. Why do we set `scale=1/mu`?

Ex 1.6. It's easy to compute the mean service time like this

Python Code

```
print(S.mean())
```

Explain that this is not exactly equal to $\mathsf{E}[S]$.

1.5 Departure times

The standard recursion to compute the departure times.

Python Code

```python
D = np.zeros_like(A)

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

print(D)
```

Ex 1.7. Explain now why it is practical to have $A_0 = 0$.

1.6 Sojourn times

How long do you stay in the system if you arrive at some time A_n and you depart at D_n?

Python Code

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

1.7 Waiting times

If your sojourn time is 10, say, and your service time at the server is 3 (and there is just one server and the service discipline is FIFO), then what was your time in queue?

Python Code

```python
W = J - S
print(W)
```

Ex 1.8. Recall that we set `S[0] = 0`. Suppose that we wouldn't have done this, and we would run the simulation for a small number of cases, why can `W.mean()` be negative?

2 KPIs and plotting

2.1 Relevant averages

Python Code

```python
print(W.mean(), W.std())
```

Python Code

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

Python Code

```python
plt.clf()
plt.plot(J)
plt.savefig("sojourn.pdf")
```

Ex 2.1. Change the simulation length to 1000 jobs. Do one run for $\mu = 3.5$ and another for 2.8. Compute the KPIs, make a plot, and include that in your assignment. Comment on what you see.

2.2 Server KPI: idle time

This code computes the total time the server is idle, and then computes the fraction of time the server is idle.

Python Code

```python
rho = S.sum() / D[-1]
idle = (D[-1] - S.sum()) / D[-1]
print(idle)
```

Ex 2.2. Explain the code above. Some specific points:

1. Why is S.sum() the total busy time of the server?
2. Why do we divide by D[-1] in the computation of ρ?
3. Explain the computation of the idle variable.

The next code computes the separate idle times.

Python Code

```python
idle_times = np.maximum(A[1:] - D[:-1], 0)
print(idle_times)
print(idle_times.sum())
print(D[-1] - S.sum())
```

Ex 2.3. Run this code for a simulation with 10 or so jobs (some other small number). Explain how this code works. Which line is a check on the computations?

2.3 Server KPI: busy time

We also like to know how a long the server has to work uninterruptedly. Finding the busy times is quite a bit harder than the idle times. (A busy time starts when a job arrives at an empty system and it stops when the server becomes free again.)

Ex 2.4. To help you understand the code, let's first do a numerical example. Suppose jobs $1,4,8$ find an empty system upon *arrival*. The simulation contains 10 jobs. Why do jobs $3,7,10$ leave an empty system behind upon *departure*?

With this idea, we can compute the idle times in another way (as a check on earlier work), and then we extend the approach to the busy times.

Python Code

```python
import numpy as np

np.set_printoptions(suppress=True)
np.random.seed(3)

num = 10
labda = 3
X = np.random.exponential(scale=1 / labda, size=num)
X[0] = 0
A = X.cumsum()
mu = 1.2 * labda
S = np.random.exponential(scale=1 / mu, size=len(A))
```

```
S[0] = 0
D = np.zeros_like(A)

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


W = D - S - A # waiting times
idx = np.argwhere(np.isclose(W, 0))
idx = idx[1:]  # strip A[0]
idle_times = np.maximum(A[idx] - D[idx - 1], 0)
print(idle_times.sum())
```

Ex 2.5. What is stored in idx? Why do we strip A[0]? Why do we subtract D[idx-1] and not D[idx]? (Print out the variables to understand what they mean, e.g., print(idx).)

Now put the next piece of code behind the previous code so that we can compute the busy times.

Python Code

```
busy_times = D[idx - 1][1:] - A[idx][:-1]
last_busy = D[-1] - A[idx[-1]]
print(busy_times.sum() + last_busy, S.sum())
```

Ex 2.6. Explain these lines. About the last line, explain why this acts as a check.

2.4 Virtual waiting time

Plotting the virtual waiting time is subtle.

Ex 2.7. Make a plot of the virtual waiting time by hand. Figure out which points are the most important ones to characterize the virtual waiting times, and explain why this is so.

Here is the code to plot the virtual waiting time.

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(3)

num = 40
labda = 1
mu = 1.1 * labda
T = 10  # this acts as the threshold
X = np.random.exponential(scale=1 / labda, size=num)
X[0] = 0
A = np.zeros_like(X)
A = X.cumsum()
S = np.ones(len(A)) / mu
S[0] = 0
D = np.zeros_like(A)

W = np.zeros_like(A)
for k in range(1, len(X)):
    W[k] = max(W[k - 1] + S[k - 1] - X[k], 0)
    D[k] = A[k] + W[k] + S[k]
```

```python

idx = np.where(W <= 0)[0] # this
empty = D[idx[1:] - 1]

E = np.zeros((2 * len(A) + len(empty), 2))  # epochs
E[: len(A), 0] = A
E[: len(A), 1] = W
E[len(A) : 2 * len(A), 0] = A
E[len(A) : 2 * len(A), 1] = W + S
E[2 * len(A) : 2 * len(A) + len(empty), 0] = empty
E[2 * len(A) : 2 * len(A) + len(empty), 1] = 0
E = E[np.lexsort((E[:, 1], E[:, 0]))]

plt.plot(E[:, 0], E[:, 1])
plt.savefig("virtual-waiting-time.pdf")
```

The `this` line is perhaps somewhat strange. First of all, `np.where(W <= 0)` returns a tuple of which the first element is an array of the indices where `W` is zero. To get that element we add the extra `[0]`.

Ex 2.8. Explain how we store the relevant epochs in `E`. Why do we use `np.lexsort`? (Check the documentation to see how lexical sorting works. It is important to know what lexical sorting is.)

3 Computing Queue length

We have the waiting times, but not the number of jobs in queue. What if we would like to plot the queue length process?

A simple, but inefficient, algorithm to construct the queue length process is to walk backwards in time.

Python Code

```python
import numpy as np
np.random.seed(3)

num = 10
X = np.random.exponential(scale=labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
mu = 0.8 * 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]

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)
```

Ex 3.1. Explain how this code works. At what points in time do we sample the queue length?

Ex 3.2. The above procedure to compute the number of jobs in the system is pretty inefficient. Why is that so?

Ex 3.3. Try to find a(more efficient algorithm to compute L. If you cannot solve this yourself, explain my code that is provided in the hint.

---

4 Multi-server queue

Let us now generalize the simulation to a queue that is served by multiple servers.

4.1 A single fast server

While you are *still in queue* of a multi-server queue, the rate at which jobs are served is the same whether there are c servers or just one server working at a rate of c, i.e., c times as fast as a server in the multi-server. As a first simple case, we model the multi-server queueing system as if there is one fast server working at rate c.

Ex 4.1. Explain that we can implement a fast server by specifying the number of servers c, and change the service times as follows:

Python Code

```python
c = 3
S = np.random.exponential(scale=1 / (c * mu), size=len(A))
```

4.2 A real multi-server queue

Here is the code to implement a real multi-server queue; see the queueing book to see how it works.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 4
num = 3

X = np.random.exponential(scale=1 / labda, size=num + 1)
S = np.random.exponential(scale=1 / mu, size=num)

# single server queue
W = np.zeros_like(S)
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] - X[k], 0)

print(W)
print(W.mean(), W.std())

# code for multi server queue
c = np.array([1.0])
```

```python
W = np.zeros_like(S)
w = np.zeros_like(c)
for k in range(len(S)):
    s = w.argmin()  # server with smallest waiting time
    W[k] = w[s]
    w[s] += S[k]  # assign arrival to this server
    w = np.maximum(w - X[k + 1] * c, 0)

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

Ex 4.2. First a test, we set the vector of server capacities c=[1] so that we reduce our multi-server queue to a single-server queue. Run the code and check that both simulations give the same result.

BTW: such 'dumb' corner cases are necessary to test code. In fact, it has happened many times that I tested code of which I was convinced it was correct, but I still managed to make bugs. A bit of paranoia is a good state of mind when it comes to coding.

Ex 4.3. We can modify the code for the single server queue such that it represents a single fast server working at rate c, rather than at 1.

Python Code

```python
W = np.zeros_like(S)
for k in range(len(S)):
    W[k] = max(W[k - 1] + S[k - 1]/c - X[k], 0)
```

How does this work?

Now that we have tested the implementation (in part), here is the code for a queue served by three servers, all working at the same speed.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 1.1
N = 1000

X = np.random.exponential(scale=1 / labda, size=N + 1)
S = np.random.exponential(scale=1 / mu, size=N)

c = np.array([1.0, 1.0, 1.0])
W = np.zeros_like(S)
w = np.zeros_like(c)
for k in range(len(S)):
    s = w.argmin()  # server with smallest waiting time
    W[k] = w[s]
    w[s] += S[k]  # assign arrival to this server
    w = np.maximum(w - X[k + 1] * c, 0)

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

Ex 4.4. Compare $\mathsf{E}[W]$ for two cases. The first is a model with single fast server working at rate $c = 3$. The second is a model with three servers each working at rate 1. Include your numerical results and discuss the differences.

Ex 4.5. Change the code for the multi-server such that the individual servers have different speeds but such that the total service capacity remains the same. What is the impact on $\mathsf{E}[W]$ and $\mathsf{V}[W]$ as compared to the case with equal servers? Include your numerical results.

Ex 4.6. 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.

5 Hints

h.1.3. The X_k are iid.

h.1.4. If I would not do this, and I would want to change the simulation length (the number of jobs), at how many places should I change this number?

h.1.6. Did we really serve job 0? If `num` is big number, does it matter that we set `S[0]=0`?

h.1.7. Observe that for D_1 we need D_0. If A_0 would be the arrival time of the first job, then what would we take for D_{-1}?

h.1.8. We subtract `S[0]` as if we served the corresponding job, but did we actually serve it?

h.2.7. The crucial points are (A_k, W_k), $(A_k, W_k + S_k)$, and $(D_k, 0)$ when $W_{k-1} = 0$. Then connect these points with straight lines.

h.3.3. Here is the code.

Python Code

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(3)

num = 4
labda = 3
X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
mu = 0.3 * labda
S = np.random.exponential(scale=1 / mu, size=len(A))
S[0] = 0
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((len(A) + len(D), 2))
L[: len(A), 0] = A
L[1 : len(A), 1] = 1
L[len(D) :, 0] = D
L[len(D) + 1 :, 1] = -1
N = np.argsort(L[:, 0], axis=0)
```

```
L = L[N]
L[:, 1] = L[:, 1].cumsum()
print(L)

plt.clf()
plt.step(L[:, 0], L[:, 1], where='post', color='k')
plt.plot(A[1:], np.full_like(A[1:], -0.3), '^b', markeredgewidth=1)
plt.plot(D[1:], np.full_like(D[1:], -0.3), 'vr', markeredgewidth=1)
plt.savefig("wait4.pdf")
```

h.4.5. For instance, set `c=np.array([2, 0.5, 0.5])`.