Queueing theory assignment: Queues with blocking rules

EBB074A05

Nicky D. van Foreest

2022:01:19

1 Blocking on waiting time

Suppose we don't allow jobs to enter the system when the waiting time becomes too long. A simple rule is the block job k when $W_k \geq T$, for some threshold $T.

How to simulate that?

1.1 A start with no blocking

Before doing something difficult, I tend to start from a situation that I do understand, which, in this case, is the single server queue without blocking.

Here is our standard code. With this we can find parameters that are suitable to see that blocking will have an impact. (Suppose $\lambda = 1$, $\mu = 1000$, $T = 100$, we will see no job being blocked during any simulation, at least not in this universe.)

Python Code

```python
import numpy as np


np.random.seed(3)

num = 10000
labda = 1
mu = 1.1 * labda
T = 5
X = np.random.exponential(scale=1 / labda, size=num)
S = np.random.exponential(scale=1 / mu, size=len(X))
S[0] = 0

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.mean(), W.max())
print((W >= T).mean()) # this
```

Ex 1.1. Run the code and check that indeed long waiting times do occur. What does the `this` line do?

Ex 1.2. Give the Kendall notation for the queueing model that we simulate here.

1.2 A hack to implement blocking

Here is a dirty hack to implement blocking. When $W_k \geq T$, job k should not enter. That means that its service time should not be added to the waiting time. But not adding the service time can be achieved by setting $S_k = 0$. To realize this, change the for loop to

Python Code

```python
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] - X[k], 0)
    if W[k] >= T:
        S[k] = 0
```

Ex 1.3. Run this code and check its effect on $\mathsf{E}[W]$, $\mathsf{V}[W]$, and $\max\{W\}$. Explain how it can occur that the maximum can still be larger than T.

1.3 A better method

As a matter of principle, I don't like the code of the previous section. In my opinion such hacks are a guarantee on bugs that can be very hard to find later. Mind, with this trick I am changing my primary data, in this case the service times. Reuse these service times at a later point in the code, for instance for a comparison with other models or for testing, has become impossible. And if I forget this (when I use this code maybe half a year later), then finding the bug will be very hard. Hence, as a golden rule: don't touch the primary data.

However, modifying this code so that I can block seems not straightforward.

Ex 1.4. The recursion we use, i.e.,

Python Code

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

assumes that job $k-1$ is accepted. Suppose that `W[k] >= T` so that job k is blocked. Explain that with this recursion, we now have a problem to compute the waiting time for job $k+1$.

Here is better code.

Python Code

```python
W = np.zeros_like(S)
I = np.ones_like(S)
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] * I[k - 1] - X[k], 0)
    if W[k] >= T:
        I[k] = 0

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

Ex 1.5. What does the vector `I` represent?

1.4 Some other blocking rules

There are other rules to block jobs.

Ex 1.6. In this code,

Python Code

```python
W = np.zeros_like(S)
I = np.ones_like(S)
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] * I[k - 1] - X[k], 0)
    if W[k] + S[k] >= T:
        I[k] = 0

print(W.mean(), W.max(), I.mean())
```

how does the rule below block jobs?

Ex 1.7. Likewise,

Python Code

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

print(W.mean(), W.max(), S.mean() - V.mean())
```

how does the rule below block jobs? What is the meaning of V?

2 Batch queues and blocking on waiting time

Let us now set up a simulation to see the combined effect of batch arrivals and blocking on waiting time.

Recall, in the queueing book we discuss some methods to block jobs in the $M^X/M/1$ queue when the queue length (not the waiting time) is too long. We tackle blocking on queue length in a separate section below.

2.1 Again start without blocking

We need a slightly different way to generate service times. When a batch of B_k jobs arrives at time A_k, then the service time added to the waiting is the sum of the service times of all B_k jobs in the batch.

Python Code

```python
import numpy as np
from scipy.stats import expon

np.random.seed(3)

num = 100000
labda = 1
mu = 1.1 * labda
X = np.random.exponential(scale=1 / labda, size=num)
B = np.random.randint(1, 2, size=num)
S = expon(scale=1 / mu)

W = np.zeros_like(X)
for k in range(1, len(W)):
    W[k] = max(W[k - 1] + S.rvs(B[k]).sum() - X[k], 0)

```

```
print(S.mean(), W.mean(), W.max())
rho = labda / mu
print(rho**2 / (1 - rho))
```

Ex 2.1. Explain how this code works.

Ex 2.2. Run the code. Why do I take `B` as it is here? Why should `W.mean()` and $\rho^2/(1-\rho)$ be approximately equal

2.2 Include blocking

Here is the code with a blocking rule.

Python Code
```python
import numpy as np
from scipy.stats import expon

np.random.seed(3)

num = 1000
labda = 1
mu = 3.1 * labda
T = 5
X = np.random.exponential(scale=1 / labda, size=num)
B = np.random.randint(1, 5, size=num)
S = expon(scale=1 / mu)

W = np.zeros_like(X)
V = np.zeros_like(W)
for k in range(1, len(W)):
    W[k] = max(W[k - 1] + V[k - 1] - X[k], 0)
    V[k] = S.rvs(B[k]).sum() if W[k] < T else 0

print(S.mean() * B.mean() - V.mean())
print(W.mean(), W.max())
print(np.isclose(V, 0).mean())
print((V <= 0).mean())  # this
```

Ex 2.3. Explain how the code works. What do the printed KPIs denote? Finally, why is the `this` line very dangerous to use, hence to be avoided? (Use `np.isclose` instead!)

3 Blocking on queue length

Blocking on queue length is quite a bit harder with a simulation in continuous time because we need to keep track of the number of jobs in the system. (Recall in discrete time the recursions to compute $\{L\}$ are easy, while in continuous time the recursions for $\{W\}$ or $\{J\}$ are easy.)

3.1 Start without blocking

As before, I start from a code that I really understand, and then I extend it to a situation that I find more difficult. So, here is code to find the system length L at *arrival* epochs $\{A_k\}$.

Python Code

```python
import numpy as np

np.random.seed(3)

num = 10000
labda = 1
mu = 1.5 * labda
X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
S[0] = 0
D = np.zeros_like(A)
L = np.zeros_like(A, dtype=int)

idx = 0
for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]
    while D[idx] < A[k]:
        idx += 1
    L[k] = k - idx

rho = labda / mu
print(L.mean(), rho/(1-rho), L.max())
print((L == 0).mean(), 1 - rho)
print((L == 1).mean(), (1 - rho)*rho)
```

Ex 3.1. Explain how this computes `L[k]`. Do we count the system length as seen upon arrival, or does `L[k]` include job k, i.e., the job that just arrived?

Ex 3.2. Just to check that you really understand: why is there no problem with the statement `(L == 0)`?

Ex 3.3. Why do I compare `L.mean()` to $\rho/(1-\rho)$ and not to $\rho^2/1-\rho)$?

Ex 3.4. Change μ to 1.05λ. Now the results of the simulation are not very good if `num=1000` or so. Making `num` much larger does the job, though.

3.2 Include blocking

It might seem that we are now ready to implement a continuous time queueing system with blocking on the queue length. Why not merge the ideas we developed above? Well, because this does not work.

(If you like a challenge, stop reading here, and try to see how far you can get with developing a simulation for this situation.)

Only after having worked for 3 hours I finally saw 'the light'. As a matter of fact, I needed a new data structure, a deque from which we can pop and append jobs at either end of a list. Here is the code.

Python Code

```python
from collections import deque
import numpy as np

np.random.seed(3)
```

```python

num = 10000
labda = 1
mu = 1.2 * labda
T = 5
X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
S[0] = 0
D = np.zeros_like(A)
L = np.zeros_like(A, dtype=int)

Q = deque(maxlen=T + 1)
for k in range(1, len(A)):
    while Q and D[Q[0]] < A[k]:
        Q.popleft()
    L[k] = len(Q)
    if len(Q) == 0:
        D[k] = A[k] + S[k]
        Q.append(k)
    elif len(Q) < T:
        D[k] = D[Q[-1]] + S[k]
        Q.append(k)
    else:
        D[k] = A[k]
```

Ex 3.5. Read the documentation of how a `deque` works, then explain the code.

Ex 3.6. At first I had these lines:

Python Code

```python
elif len(Q) < T:
    Q.append(k)
    D[k] = D[Q[-1]] + S[k]
```

Why is that wrong?

Ex 3.7. What goes wrong with this code:

Python Code

```python
if len(Q) < T:
    D[k] = max(D[Q[-1]], A[k]) + S[k]
    Q.append(k)
```

Ex 3.8. What queueing discipline would result if we would use the `pop()` and `appendleft()` methods of a deque?

Ex 3.9. What queueing discipline would result if we would use the `pop()` and `append()` methods of a deque?

Ex 3.10. Run this code with T=100 and compare this with the queueing system without blocking. Why should you get the same results? (Realize that this is a check on the correctness of our code.)

Glue the next code (for the theoretical model) at the end of the previous code.

Python Code

```python
rho = labda / mu
p = np.ones(T + 1)
for i in range(1, T + 1):
    p[i] = rho * p[i - 1]
p /= p.sum()
for i in range(T + 1):
    print((L == i).mean(), p[i])
```

Ex 3.11. Now set T=5 and `num = 10000` or so. Run the code. Why do the result agree with the theoretical model? Why is this the $M/M/1$ queue?

In fact, I used the above theoretical model to check whether the simulation was correct. (My first 20 or so attempts weren't.)

4 An algorithm for the $M/G/1$ with blocking

In the queueing book we develop an algoritm to compute $\pi(n)$. Here we implement this, use this as another test on the simulator, and improve our understanding of queueing systems.

4.1 The algorithm

This is the code.

Python Code

```python
import numpy as np
from scipy.integrate import quad
from scipy.stats import expon

np.random.seed(3)

labda = 1
mu = 1.2 * labda
T = 5
S = expon(scale=1 / mu)


def g(j, x):
    res = np.exp(-labda * x) * (labda * x) ** j * S.pdf(x)
    return res / np.math.factorial(j)


f = np.zeros(T + 1)
for j in range(T + 1):
    f[j] = quad(lambda x: g(j, x), 0, np.inf)[0]

F = f.cumsum()
G = 1 - F

pi = np.ones(T + 1)
for n in range(T):
    pi[n + 1] = pi[0] * G[n]
    pi[n + 1] += sum(pi[m] * G[n + 1 - m] for m in range(1, n + 1))
    pi[n + 1] /= f[0]

pi /= pi.sum()
print(pi)
```

Ex 4.1. Which formulas (give the numbers) of the queueing book have we implemented?

Ex 4.2. Run this code after the computation of `f`.

Python Code
```
j = 2
print(mu / (mu + labda) * (labda / (labda + mu)) ** j, f[j])
```

Why should these numbers be the same?

Ex 4.3. Run the code for $\mu = 0.3$ and compare the numerical results to what you get from:

Python Code
```
rho = labda / mu
p = np.ones(T + 1)
for i in range(1, T + 1):
    p[i] = rho * p[i - 1]
p /= p.sum()
print(pi)
```

Explain why you should get the same numbers.

Ex 4.4. When the service times are contant, explain that this code computes `f` correctly:

Python Code
```
from scipy.stats import expon, uniform, poisson
# include useful code here
f = poisson(labda / mu).pmf(range(T + 1))
```

Then change the S in the simulation part to

Python Code
```
S = np.ones(len(A)) / mu
```

Run the code and include your results; of course the simulation and the algoritm should give more or less the same results.

Ex 4.5. As another good example, take $S \sim U(0, 2/\mu)$. The relevant code changes are this:

Python Code
```
from scipy.stats import expon, uniform, poisson
# other stuff for the model
S = uniform(0, 2 / mu)
```

and for the simulator:

Python Code
```
S = np.random.uniform(0, 2 / mu, size=len(A))
```

Run the code, and include your output.

4.2 Effect of blocking on performance

Ex 4.6. Take $\lambda = 1$ and $\mu = 1.1$. Use the algorithm to compute the loss probability and $\mathsf{E}[L]$ for $T = 5$, $T = 10$ and $T = 15$. Include the numbers.

Ex 4.7. Do the same computations for $\mu = 0.5\lambda$. Why is the loss probability not so sensitive to T?

Ex 4.8. Set $\mu = 1.2\lambda$ again. Then compare the loss probability for $T = 5, 10, 15$ for $S \sim \text{Exp}(\mu)$ and $S \sim \text{U}(0, 2/\mu)$. What is the influence of service time variability on the loss when $T = 5$, $T = 10$, $T = 15$? Why is this influence relatively more important for larger T?

5 Hints

h.1.2. It's not the $D/D/1$ queue.

h.1.3. Check the section in the queueing book on the $M^X/M/1$ queue, in particular the section on blocking. Which blocking method do we include here?

h.1.4. We want to block job k when $W_k \geq T$. That means that job $k+1$ should not see S_k, but (as far as I see) we cannot convert that into an elegant recursion for W_{k+1}

h.1.5. If `I[k] == 1`, then what happens to job k?

h.2.2. Why is this the $M/M/1$ queue when the batches `B = np.random.randint(1, 2, size=num)`? (Recall my obsession with testing code.)

h.3.1. When the while loop terminates, is `idx` the index of the last departure, or does it point to the job that is the first to leave?

h.3.2. Is `L` a float?

h.3.3. What is $\rho^2/(1-\rho)$?

h.3.6. To what job does `Q[-1]` point to? It's one to far!

h.3.7. When `Q` is empty (i.e., `len(Q)` is 0), then what is `Q[-1]`?

h.3.8. Does it matter whether we push jobs from right to left through a quue, rather then from left to right?

h.3.9. It's not FIFO.

h.3.10. Is `L.max()` larger than 100 for this simulation?

h.4.6. Why is the loss probality equal to π_T?