Applications of Sakasegawa's formula

EBB074A05

Nicky D. van Foreest

2022:01:19

1 Various models compared

We have (at least) three different types of models for a queueing system: simulation in discrete time, simulation in continuous, and Sakasegawa's formula to compute the expected waiting time in queue. Let's see how these models compare.

1.1 Discrete time simulation

We have a machine that can serve c_k jobs on day k. When the period length is T, then $c_k \sim \text{Pois}(\mu T)$. The number jobs that arrive on k is $a_k \sim \text{Pois}(\lambda T)$. The arrivals in period k cannot be served on day k. The code to simulate this should be familiar to you by now.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 1.2 * labda
T = 8  # period length, 8 hours in a day

num = 1000

a = np.random.poisson(labda * T, size=num)
c = np.random.poisson(mu * T, size=num)
L = np.zeros_like(a, dtype=int)

L[0] = 5
for i in range(1, num):
    d = min(c[i], L[i - 1])
    L[i] = L[i - 1] + a[i] - d


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

Ex 1.1. Run this code (and write down the results). Then change the code such that the arrivals can be served on the day they arrive. Rerun and compare the results.

Ex 1.2. Now change the period time, which was 8 hours in a day, to $T = 1$ (so that we concentrate on what happens during an hour instead of a day). Rerun the code with and without the arrivals being served on the period of arrival, and compare to the previous exercise. Explain the difference.

Ex 1.3. What are the advantages and disadvantages of using small values for T?

1.2 Continuous time simulation

Here is the code.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 1.2 * labda
num = 1000
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)

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

J = D - A
EL = labda * J.mean()
print(EL)
```

Ex 1.4. Which result did we use to compute $\mathsf{E}[L]$? Can we use that to estimate $\mathsf{V}[L]$?

Ex 1.5. Run the code and compare with the discrete time simulation.

1.3 Sakasegawa's formula

Here we use Sakasegawa's formula to compute $\mathsf{E}[L]$.

Python Code

```python
import numpy as np


def sakasegawa(labda, ES, Ca2=1, Cs2=1, c=1):
    rho = labda * ES
    V = (Ca2 + Cs2) / 2
    U = rho ** (np.sqrt(2 * (c + 1)) - 1) / (1 - rho)
    T = ES / c
    return V * U * T


labda = 3
mu = 1.2 * labda
ES = 1 / mu
W = sakasegawa(labda, 1 / mu, 1, 1, 1)
L = labda * (W + ES)
print(L)
```

Ex 1.6. Explain the code, in particular the values of the parameters.

Ex 1.7. Run the code, and compare the value with the discrete and continuous time simulations.

Ex 1.8. What are advantages and disadvantages of using Sakasegawa's formula?

2 Server setup and batching

We can now setup a model in which jobs arrive in batches of size $B and in between batches the server needs a constant setup time R. Check the queueing book for further background; we are going to build the model of the related section.

2.1 Sakasegawa's formula

I build up the code in small blocks. You should put the code blocks underneath each other as you progress.

Since we add setup times, it's easy to have too large loads.

Python Code

```python
import numpy as np

np.random.seed(3)

B = 13
labda = 3
mu = 2 * labda
R = 2

rho = labda * (1 / mu + R / B)
assert rho < 1, f"{rho=} >= 1"
```

Ex 2.1. What does the `assert` statement? Why do I put it before doing any other work? What happens if you would set `labda=10` and `mu=3`.

Next, we need to get the parameters correct for the batches. I just follow the book.

Python Code

```python
W1 = (B - 1) / 2 / labda
```

Ex 2.2. What is `W1` conceptually:?

For the queueing time, we have this:

Python Code

```python
labdaB = labda / B
VR = 0  # constant R
ca2 = 1 / B
ES0 = 1 / mu
VS0 = ES0 * ES0
VSB = VR + B * VS0
ES = ES0 + R / B
ESB = R + B * ES0
cs2 = VSB / ESB ** 2
```

Ex 2.3. We set `ca2 = 1`. What is the assumption about the distribution of the interarrival times X?

Ex 2.4. Print `cs2`. Why is that smaller than 1? #+end$_{src}$

For the average queueing time:

Python Code
```python
def sakasegawa(labda, ES, Ca2, Cs2, c=1):
    rho = labda * ES
    V = (Ca2 + Cs2) / 2
    U = rho ** (np.sqrt(2 * (c + 1)) - 1) / (1 - rho)
    T = ES / c
    return V * U * T

W2 = sakasegawa(labdaB, ESB, ca2, cs2)
```

Ex 2.5. In view of the previous exercise, explain that

Python Code
```python
sakasegawa(labda, ES, 1, 1)
```

gives the wrong result, but

Python Code
```python
sakasegawa(labda, ES, 1, cs2)
```

agrees with

Python Code
```python
sakasegawa(labdaB, ESB, ca2, cs2)
```

The last step:

Python Code
```python
W3 = R + (B + 1) / 2 * ES0
```

Ex 2.6. What is the meaning of `W3`?

The sojourn time:

Python Code
```python
J = W1 + W2 + W3
print(J)
```

2.2 Simulation

To set up the simulation requires a bit fiddling with slicing. It took me a bit of time, and print statements, to get the details right. Here is the code, with the print statements so that you can figure out how it works.

Python Code

```python
# Don't forget the copy the parameters like B so that you work with
# the same numbers.

num = 10
num = B * (num // B)  # get multiple of B
X = np.ones(num)
X[0] = 0
A = X.cumsum()
A = np.zeros_like(A0)
for i in range(num // B):
    st = i * B + 1  # start
    fi = (i + 1) * B  # finish
    print(st, fi, A0[fi])
    A[st : fi + 1] = A0[fi]
    print(A)

S0 = np.ones_like(A0)
S0[0] = 0
S = S0.copy()
S[1::B] += R
print(S)
```

Ex 2.7. Use the print statements to explain how the slicing, i.e., notation like `A[st : fi +1]`, works. Explain how A and S correspond to batch arrivals and services with setup times.

Now that we know how to construct batch arrivals and job service times that include regular setups, the rest of the simulation is standard.

Python Code

```python
# put the parameters here, or glue this code after the code for
# Sakasegawa's formula

num = 1000
num = B * (num // B)  # get multiple of B
X = np.random.exponential(scale=1 / labda, size=num)
A0 = np.zeros(len(X) + 1)
A0[1:] = X.cumsum()
A = np.zeros_like(A0)
for i in range(num // B):
    st = i * B + 1  # start
    fi = (i + 1) * B  # finish
    A[st : fi + 1] = A0[fi]


S0 = np.random.exponential(scale=1 / mu, size=len(A))
S0[0] = 0
S = S0.copy()
S[1::B] += R

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

J = D - A0
print(J.mean())
```

Ex 2.8. Why is `D - A` not the sojourn time?

Ex 2.9. Run the code for `num = 1000` and compare the results of the formulas and the simulation. (Ensure that both models use the same data.) Then extend to `num = 1_000_000` and check again. What do you see, and conclude?

2.3 Getting things right

While making the above code, I made several (tens of) errors, so that the simulation and the formulas gave different results. Here are the steps that I followed to get things right. Only after one step was correct, I moved on to the next.

1. Check with $B = 1$ and $R = 0$, since $B = 1$ is the single job case.
2. Keep $B = 1$, set $R = 0.2$. I had to change μ so that still $\rho < 1$.
3. Set $B = 2$, $R = 0$. Compare `ES` (input for Sakasegawa's formula) to `S.mean()` (input for simulation).
4. In the previous step I did not get corresponding results for `num = 10000`. Changing it to 1 million helped.

After these four steps, the simulation and the model gave similar results. However, from a higher level of abstraction, I am not quite happy about this. It is not realistic to wait until we have seen a million or so arrivals in any practical setting. My personal way to deal with this situation is like this (but not all people agree on what to do though):

- Simple formulas are tremendously useful to get *insight* into the main drivers of the behavior of a system. In other words, there is not better way to get *qualitative* understanding than with simple formulas.
- The quantitative quality of a formula need to not be too good.
- Building a simulator is intellectually rewarding as it helps understand the *dynamics* of a system.
- Building a simulator is difficult; it's easy to make mistakes, in the code, in the model, in the data...
- Simulation depends on large quantities of data. It's very hard (next to impossible) to *understand* the output.
- The simple formulas can be used to check the output of a simulator when applied to simple cases.

All in all, I think that simulation and theoretical models should go hand in hand, as they offer different type of insight, and have different strengths and weaknesses.

2.4 Random setup times

3 Server Adjustments

Consider now a queueing system in which the server needs an adjustments with probability p (see the section on server adjustments in the book). The repair times are assumed constant, at first, with mean 2. Here is the simulator.

3.1 Check work

First we should check the formulas for $\mathsf{E}[S]$ and $\mathsf{V}[S]$.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 2 * labda
ES0 = 1 / mu
p = 1 / 20
num = 10000


S0 = np.random.exponential(scale=ES0, size=num)
R = 2 * np.ones_like(S0)
I = np.random.uniform(size=len(R)) <= p
S = S0 + R * I
print(S.mean(), S.var())

ER = R.mean()
ES = ES0 + p * ER
VS0 = ES0 * ES0
VR = R.var()
VS = VS0 + p * VR + p * (1 - p) * ER * ER
print(ES, VS)
```

Ex 3.1. Explain what is `I`. Then explain the line `S0 + R * I` works.

Ex 3.2. Run the code; include the numbers in your assignment. Are the numbers nearly the same?

3.2 The simulations

Now that we checked the formulas to compute $\mathsf{E}[S]$ and $\mathsf{V}[S]$, we can see how well Sakasegawa's formula applies for a queueing system in which a server requires regular, but random, adjustments.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 2 * labda
p = 1 / 20
num = 10000

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))
R = 2 * np.ones(len(S))                              # this
I = np.random.uniform(size=len(S)) <= p
D = np.zeros_like(S)

```

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

W = D - A - S
print(W.mean())
```

Ex 3.3. Explain how D is computed.

To see how the approximation works, glue the next code below the previous code.

Python Code

```
def sakasegawa(labda, ES, Ca2=1, Cs2=1, c=1):
    rho = labda * ES
    V = (Ca2 + Cs2) / 2
    U = rho ** (np.sqrt(2 * (c + 1)) - 1) / (1 - rho)
    T = ES / c
    return V * U * T


ES0 = 1 / mu
VS0 = ES0 * ES0
ER = R.mean()
ES = ES0 + p * ER
rho = labda * ES
assert rho < 1, "rho >= 1"
VR = R.var()
VS = VS0 + p * VR + p * (1 - p) * ER * ER
Cs2 = VS / ES / ES
W = sakasegawa(labda, ES, 1, Cs2, 1)
print(W)
```

Ex 3.4. To test the code I set at first `R = 0 * np.ones(len(A))` in the line marked as `this`. Why is this a good test?

Ex 3.5. Now run the code, with R as in the code (not set as 0 such as in the previous exercise). Compare the answers. Then set `num = 100000`, i.e., 10 times as large. What is the effect?

Ex 3.6. Now set `R = np.random.exponential(scale=2, size=len(A))`. What is the effect on $\mathsf{E}[W]$? In general, do you see that indeed $\mathsf{E}[W]$ increases with the variability of the adjustments?

Ex 3.7. What is the model behind this code?

Python Code

```
import numpy as np

np.random.seed(3)

labda = 3
mu = 4
N = 1000

X = np.random.exponential(scale=1 / labda, size=N)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
```

```python
R = np.random.uniform(0, 0.1, 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] + R[k]

W = D - A - S
print(W.mean(), W.std())
```

Ex 3.8. In stead of

Python Code
```python
for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k] + R[k] * I[k]
```

we could write

Python Code
```python
for k in range(1, len(A)):
    D[k] = max(D[k - 1] + R[k] * I[k], A[k]) + S[k]
```

What modeling choice would this change reflect? Which of these two models makes the sojourn smaller?

---

4 Server failures

This time we focus on a server that can fail; again check the queueing book for the formulas. Here we just implement them.

4.1 Check work

Again, first we need to check that our (implementation of the) formulas for $\mathsf{E}[S]$ and $\mathsf{V}[S]$ are correct.

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

np.random.seed(3)

labda = 3
mu = 2 * labda
ES0 = 1 / mu
labda_f = 2
ER = 0.5
num = 10000

S0 = np.random.exponential(scale=ES0, size=num + 1)
N = np.random.poisson(labda_f * ES0, len(S0))
R = expon(scale=ER)
S = np.zeros_like(S0)
for i in range(len(S0)):
    S[i] = S0[i] + R.rvs(N[i]).sum()
```

```python

A = 1 / (1 + labda_f * ER)
ES = ES0 / A
print(ES, S.mean(), S0.mean() + N.mean() * R.mean())

C02 = 1
Cs2 = C02 + 2 * A * (1 - A) * ER / ES
print(Cs2, S.var() / (S.mean() ** 2))
```

Ex 4.1. Run this code, and check the result. Then chance `num` to 100000 to see that the estimate improves.

Ex 4.2. Explain how we compute `S[i]`.

4.2 The simulations

Here is all the code to compare the results of a simulation to Sakasegawa's formula.

Python Code

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

np.random.seed(3)

labda = 2
mu = 6
ES0 = 1 / mu
labda_f = 2
ER = 0.5
num = 10000

S0 = np.random.exponential(scale=ES0, size=num + 1)
N = np.random.poisson(labda_f * ES0, len(S0))
R = expon(scale=ER)
S = np.zeros_like(S0)
for i in range(len(S0)):
    S[i] = S0[i] + R.rvs(N[i]).sum()


X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
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 - S
print(W.mean())


def sakasegawa(labda, ES, Ca2=1, Cs2=1, c=1):
    rho = labda * ES
    V = (Ca2 + Cs2) / 2
    U = rho ** (np.sqrt(2 * (c + 1)) - 1) / (1 - rho)
    T = ES / c
    return V * U * T
```

```python


A = 1 / (1 + labda_f * ER)
ES = ES0 / A
C02 = 1
Cs2 = C02 + 2 * A * (1 - A) * ER / ES
rho = labda * ES
assert rho < 1, "rho >= 1"
W = sakasegawa(labda, ES, 1, Cs2, 1)
print(W)
```

Ex 4.3. Run the code and include the results in your assignment.

Ex 4.4. Suppose you can choose between two alternative ways to improve the system. Increase λ_f, and decrease $\mathsf{E}[R]$, but such that $\lambda_f \mathsf{E}[R]$ remains constant; or the other way around, decrease λ_f and increase $\mathsf{E}[R]$. Which alternative has better influence on $\mathsf{E}[W]$? (Use Sakasegawa's formula for this; you don't have to do the simulations. In general, computing functions is much faster than simulation.)

5 A simple tandem network

We have two queues in tandem. Again we compare the approximations with simulation.

5.1 Simulation in continuous time

This code simulates two queues in tandem in continuous time.

Python Code

```python
import numpy as np

np.random.seed(4)

labda = 3
mu1 = 1.2 * labda
num = 100000
X = np.random.exponential(scale=1 / labda, size=num)
A1 = np.zeros(len(X) + 1)
A1[1:] = X.cumsum()
S1 = np.random.exponential(scale=1 / mu1, size=len(A1))
D1 = np.zeros_like(A1)

for k in range(1, len(A1)):
    D1[k] = max(D1[k - 1], A1[k]) + S1[k]

W1 = D1 - A1 - S1

# queue two
A2 = D1
mu2 = 1.1 * labda
S2 = np.random.exponential(scale=1 / mu2, size=len(A2))
D2 = np.zeros_like(A2)

for k in range(1, len(A2)):
    D2[k] = max(D2[k - 1], A2[k]) + S2[k]

```

```python
W2 = D2 - A2 - S2

J = D2 - A1
print(W1.mean(), S1.mean(), W2.mean(), S2.mean(), J.mean())
```

Ex 5.1. Explain how the code works. Why did I choose these parameters?

5.2 Sakasegawa's formula plus tandem formula

Now we can check the quality of the approximations.

Python Code

```python
import numpy as np


def sakasegawa(labda, ES, Ca2=1, Cs2=1, c=1):
    rho = labda * ES
    V = (Ca2 + Cs2) / 2
    U = rho ** (np.sqrt(2 * (c + 1)) - 1) / (1 - rho)
    T = ES / c
    return V * U * T


labda = 3
mu1 = 1.2 * labda
ES1 = 1 / mu1
rho1 = labda * ES1
Ca2 = 1
Cs2 = 1
W1 = sakasegawa(labda, ES1, Ca2, Cs2, 1)
Cd2 = rho1 ** 2 * Ca2 + (1 - rho1 ** 2) * Cs2

Ca2 = Cd2
mu2 = 1.1 * labda
ES2 = 1 / mu2
Cs2 = 1
W2 = sakasegawa(labda, ES2, Ca2, Cs2, 1)


J = W1 + ES1 + W2 + ES2
print(W1, ES1, W2, ES2, J)
```

Ex 5.2. How does the code work? Compare the results obtained from simulation and by formula.

Ex 5.3. Assume that the service times at the first queue are constant and equal to $1/\mu_1$. What should be the parameter values for Sakasegawa's formula (explain). Then run both codes and comment on the results.

6 Hints

h.1.1. Replace the relevant line by `d = min(c[i], L[i - 1] + a[i])`.

h.1.4. Recall $L = \lambda W$.

h.2.4. When $S = R/B + S_0$, then part of S is constant. Hence, can it be (relatively) as variable as S_0?

h.2.7. Why do I chose `np.ones` to fill `A0` and `S0`? What is the difference between `A0` and `A`, and `S0` and `S`?

h.2.8. What do you think I used first?

h.3.8. What is the influence on the setup? Do we still require that the setup has to be done immediately before a service starts?

h.4.4. For instance, set `labda_f = 4` and `ER = 0.25`.

h.5.1. For what situations are the formulas exact?

h.5.3. In the simulation, replace the line with `S1` by `S1 = np.ones(len(A1)) / mu1`. In the formula's, don't forget to update C_s^2 for the relevant queue.