PS 07 – Monte Carlo integration & simulations – solutions

George McNinch

2024-03-29

1. Consider the function $f(x) = \dfrac{1}{x}$ defined on the interval $I = \left[\dfrac{1}{2}, 1\right]$. Note that f is a decreasing function on the interval, and in particular

$$\frac{1}{x} \leq 4$$

for each $x \in I$. Recall that

$$\int_{1/2}^{1} \frac{1}{x} dx = \ln(x)\Big|_{1/2}^{1} = -\ln(1/2) = \ln(2).$$

a. If X and Y are random variables uniformly distributed respectively on the intervals $[1/2, 1]$ and $[0, 4]$, explain why

$$P\left(\frac{1}{2} \leq X \leq 1, 0 \leq Y \leq \frac{1}{X}\right) = \frac{\ln(2)}{2}.$$

Solution:

The value (X, Y) of the random variables represents the choice of a "random" point in the rectangle R defined by $1/2 \leq x \leq 1$ and $0 \leq y \leq 4$.

The area of R is $\dfrac{1}{2} \cdot 4 = 2$. The probability that this point is below the curve $y = 1/x$ is the ratio $\dfrac{A}{2}$ where A represents the *area under the curve* $y = 1/x$.

Using the Fundamental Theorem of Calculus, we find this area under the curve as the integral of $y = \dfrac{1}{x}$ over the given interval:

$$\int_{1/2}^{1} \frac{1}{x} dx = \ln x|_{1/2}^{1} = \ln(1) - \ln(1/2) = \ln(2)$$

Thus the probability $P\left(\dfrac{1}{2} \leq X \leq 1, 0 \leq Y \leq \dfrac{1}{X}\right)$ is given by the ratio $\dfrac{A}{2} = \dfrac{\ln 2}{2}$.

---

b. Write a `python` function which takes as argument a whole number `n` and estimates $\ln(2)$ by generating `n` random points `(x,y)` in the region $[1/2, 1] \times [0, 4]$, counting the number `m` of those points `(x,y)` for which y is *below* the graph $y = \dfrac{1}{x}$, and using the ratio `m/n` to produce an estimate of $\ln(2)$.

Include the text of your function in your problem submission, and include a brief explanation of how it works.

Compare your result to `npumpy.log(2)` (note that `numpy.log` is the natural logarithm). How large must `n` be in order that your estimate matches `numpy.log(2)` to 2 decimal places?

SOLUTION:

Here is the required code:

```
import numpy as np

from numpy.random import default_rng
rng = default_rng()

def randomPoint():
    # return a random point in the rectangl [1/2,1] x [0,4]
    return (rng.uniform(1/2,1),rng.uniform(0,4))

def estimate_log_two(n):
    ll = [ randomPoint() for _ in range(n) ]    # make a list of n random points
    lr = [ (x,y) for (x,y) in ll if y <= 1/x ]  # find the points below the curve
    return len(lr)/len(ll)                      # return the fraction of points below the curve
```

Now, we can compute `log(2)/2`

```
np.log(2)/2
=>
0.34657359027997264
```

We consider:

> Compare your result to npumpy.log(2) (note that numpy.log is the natural logarithm). How large must n be in order that your estimate matches numpy.log(2) to 2 decimal places?

```
[ (n,estimate(1000*n)) for n in range(10,40,2)]
=>
[(10, 0.3462),
 (12, 0.34991666666666665),
 (14, 0.34985714285714287),
 (16, 0.3485),
 (18, 0.3448888888888889),
 (20, 0.3505),
 (22, 0.34745454545454546),
 (24, 0.3445416666666667),
 (26, 0.34465384615384614),
 (28, 0.35010714285714284),
 (30, 0.3442),
 (32, 0.346),
 (34, 0.34755882352941175),
 (36, 0.3487222222222222),
 (38, 0.3465526315789474)]
```

The estimate of course depends on *pseudo-random* numbers, but it appears to be more-or-less reliably correct to two decimals places with $n \equiv 30 \cdot 1000 = 30,000$

***Jane's Fish Tank Emporium* (*JFTE*) revisited.**

Recall that in the course notebook, we discussed the operation of *JFTE* by considering the question: what is the optimal ordering strategy for fish tanks?

Is it *on-demand* ordering (where an order is made after a sale)?

Or is it better to have *standing orders* (where an order is made regularly – say, on a particular day of the week)?

2. In the notebook, we studied the case for which the probability of a customer arriving at the store on any particular day was `1/7`. Let's now consider the case where the probability of the arrival of a customer to the store depends on the day of the week, as follows:

Day	Sun	Mon	Tue	Wed	Thur	Fri	Sat
DOW	0	1	2	3	4	5	6
Prob	0.16	0.08	0.04	0.08	0.12	0.25	0.27

Here the `DOW` ("day of week") row just indicates that we view `Mon` as day 1 of a week, `Tue` as day 2, etc.

In the notebook, we constructed a `python` class `JFTE` to keep track of our simulations. The *constructor* of the class `JFTE` (i.e. its member function `__init__`) creates the `customer` instance variable; to do this, it invokes the function

```
def customer(prob=1./7):
   return rng.choice([1,0],p=[prob,1-prob])
```

Make an alternative to this function `customer` by creating a new function `customer_alt` taking an integer argument `m` which returns `1` with probability as indicated in the above table (for the `DOW` corresponding to `m`) and otherwise returns `0`.

SOLUTION:

```
# we'll keep the probabilities in a dictionary
dow_probs = { 0: .16,
              1: .08,
              2: .04,
              3: .08,
              4: .12,
              5: .25,
              6: .27
            }

def customer_alt(d,dow_probs):
  pp = dow_probs[mod(d,7)]             # get the probability `pp` for the `dow` of `d`
  return rng.choice([1,0],p=[pp,1-pp]) # return a random choice based on `pp`
```

Now we *copy* the code for analyzing `Janes Fish Tank Emporium`. We have to modify the `__init__` method of the class `JFTE` slightly, in order to use the new function `customer_alt` to generate customers. More precisely, the `__init__` function now takes arguments `N, dow_probs` where `N` is the number of days and `dow_probs` is a dictionary representing the probability of a customer for the given day of the week.

```
class JFTE():
    def __init__(self,N,dow_probs):
        self.customers = [customer_alt(n,dow_probs) for n in range(N)]
        self.num_days = N
        self.reset()

    def reset(self):
        self.stock = 1
        self.sales = 0
```

```
        self.lost_sales = 0
        self.storage_days = 0
        self.max_stock = 1

    def add_stock(self):
        self.stock = self.stock + 1
        if self.stock > self.max_stock:
            self.max_stock = self.stock

    def sale(self):
        self.stock = self.stock - 1
        self.sales = self.sales + 1

    def result(self):
        return { 'number_days': self.num_days,
                 'weeks': self.num_days/7.0,
                 'sales': self.sales,
                 'lost_sales': self.lost_sales,
                 'storage_days': self.storage_days,
                 'max_stock': self.max_stock
               }


def stand_order(J,dow=6):
    ## dow = arrival day-of-week for standing order; should be in [0,1,2,3,4,5,6]
    ## we'll assume that the first day of the ``days`` list is dow=0.

    N = J.num_days
    J.reset()

    # loop through the days
    for i in range(N):
        c = J.customers[i]            ## c is 1 if there is a customer on day i, 0 otherwise

        if dow == np.mod(i,7):        ## add stock on the dow for order arrival
            J.add_stock()

        if c>0 and J.stock == 0:
            J.lost_sales = J.lost_sales + 1   ## lost sale if no stock

        if c>0 and J.stock > 0:               ## sale if adequate stock
            J.sale()

        J.storage_days = J.storage_days + J.stock    ## accumulate total storage costs

    return J.result()

def order_on_demand(J):
    J.reset()
    order_wait = np.inf                        ## order_wait represents wait-time
                                               ## until next order arrival

    ## loop through the customers
    for c in J.customers:
        if c>0 and J.stock==0:                 ## record lost sale if no stock
            J.lost_sales = J.lost_sales + 1
```

```
        if c>0 and J.stock>0:                     ## record sale if adequate stock
            J.sale()

        J.storage_days += J.stock                 ## accumulate storage days

        if  J.stock==0 and order_wait == np.inf:  ## reorder if stock is empty and no current order
            order_wait = 5

        if order_wait == 0:                       ## stock arrives
            J.add_stock()
            order_wait = np.inf

        if order_wait>0:                          ## decrement arrival time for in-transit orders
            order_wait -= 1

    return J.result()
```

Run the simulation 10 times with both strategy functions, as was done in the notebook. Discuss similarities/differences between the results obtained in the notebook.

Now we run the simulation. We make 10 trials, each of length 2 years:

```
import pandas as pd

def make_trials(dow_probs,trial_weeks = 2*52, num_trials = 10):
    return [ JFTE(7*trial_weeks,dow_probs) for _ in range(num_trials) ]

def report_trials(strategy,trials):

    results = [ strategy(t) for t in trials ]

    details = ['weeks', 'sales', 'lost_sales', 'storage_days', 'max_stock']

    sd = {i: [r[i] for r in results ] for i in details}

    return pd.DataFrame(sd)

## make a list of 10 trials. Each trial has length 2 years
## use the `new_dow_probs`
ten_trials = make_trials(new_dow_probs)
```

We first report the results for the `standing orders`:

```
stand_results = report_trials(stand_order,ten_trials)
stand_results
=>
   weeks  sales  lost_sales  storage_days  max_stock
0  104.0     91          12          3573         14
1  104.0    103          13          1805          7
2  104.0     97          10          1819          8
3  104.0    103           0          2083          6
4  104.0     99           0          3548          9
5  104.0     92           7          6062         16
6  104.0     86           2          6980         19
7  104.0     93           8          2630         12
8  104.0    102           6          3408         11
9  104.0     97           5          4687         14
```

And now we report the results for `on demand ordering`:

```
demand_results = report_trials(order_on_demand, ten_trials)
demand_results
=>
    weeks    sales   lost_sales  storage_days    max_stock
0   104.0       61           42           362            1
1   104.0       69           47           314            1
2   104.0       66           41           332            1
3   104.0       65           38           338            1
4   104.0       63           36           354            1
5   104.0       63           36           354            1
6   104.0       55           33           402            1
7   104.0       63           38           350            1
8   104.0       69           39           316            1
9   104.0       65           37           338            1
```

Since we report the trials as a `pandas DataFrame`, we can easily use a `pandas` method to compute the *means* of the various values:

```
stand_results.mean()
=>
weeks            104.0
sales             96.3
lost_sales         6.3
storage_days    3659.5
max_stock         11.6
dtype: float64

demand_results.mean()
=>
weeks           104.0
sales            63.9
lost_sales       38.7
storage_days    346.0
max_stock         1.0
dtype: float64
```

As was already the case with the version we discussed in the lecture, the standing-order strategy results in a much larger number of days required to store fish-tanks – 3659 days of storage – than the on-demand strategy.

On the other hand, we make more sales using the standing order strategy, and have many fewer lost sales

---

3. In this problem, let's consider again the "constant" customer arrival probability described in the notebook.

   For each strategy `stand_order` and `order_on_demand` , compute the average `storage_days` and the average `sales` for 10 simulations. (So you'll have averages for `stand_order` and averages for `order_on_demand`).

   If the storage costs are $1 per tank per day, use your averages to estimate what the profit per tank needs to be for `JFTE` to have a positive `net_profit` for each of these strategies.

Solution:

We can just use the code we used in problem 2 if we specify the `constant probabilities` as a dictionary.

```
const_probs = { n: 1./7 for n in range(7) }
const_probs
=>
```

```
{0: 0.14285714285714285,
 1: 0.14285714285714285,
 2: 0.14285714285714285,
 3: 0.14285714285714285,
 4: 0.14285714285714285,
 5: 0.14285714285714285,
 6: 0.14285714285714285}
```

Now we make 10 trials and get the

```
## make a list of 10 trials. Each trial has length 2 years
## this time use constant probabilities
const_ten_trials = make_trials(const_probs)

const_stand_results = report_trials(stand_order,const_ten_trials)
const_demand_results = report_trials(demand_order,const_ten_trials)
```

Now we can inspect the results:

```
const_stand_results
=>
    weeks  sales  lost_sales  storage_days  max_stock
0   104.0   102          1          3331         10
1   104.0    94          0          4322         12
2   104.0    98         15          1430          9
3   104.0    92          2          5146         14
4   104.0   100          9          1834          8
5   104.0   102          6          2704         10
6   104.0    98          6          2273          9
7   104.0    91          0          8034         16
8   104.0    95          2          3031         10
9   104.0   101          8          2369          9

const_demand_results
=>
    weeks  sales  lost_sales  storage_days  max_stock
0   104.0     59         44           374          1
1   104.0     55         39           401          1
2   104.0     66         47           332          1
3   104.0     56         38           392          1
4   104.0     60         49           370          1
5   104.0     64         44           344          1
6   104.0     62         42           356          1
7   104.0     51         40           422          1
8   104.0     58         39           380          1
9   104.0     58         51           380          1
```

And even inspect the *means*:

```
const_stand_results.mean()
=>
weeks            104.0
sales             97.3
lost_sales         4.9
storage_days    3447.4
max_stock         10.7
dtype: float64
```

```
const_demand_results.mean()
=>
weeks           104.0
sales            58.9
lost_sales       43.3
storage_days    375.1
max_stock         1.0
dtype: float64
```

We use these means to answer the question

> If the storage costs are \$1 per tank per day, use your averages to estimate what the profit per tank needs to be for `JFTE` to have a positive `net_profit` for each of these strategies.

Using the *standing order* strategy, over a two year period we expect to pay for \$3447 storage-days, so at the indicated rate, we expect to pay \$3447 in storage costs for the period.

On the other hand, we make on average 97 sales. To make a profit, our profit per tank needs to be at least $3447/97 = 35.53$ dollars per tank.

Using the *on demand* ordering strategy, over a two year period we expect to pay \$375 in storage costs.

On the other hand, we make on average 58 sales. To make a profit, the profit-per-tank needs to be at least $375/58 = 6.46$ dollars per tank.

In fact, we can compute these requirements with a function:

```
def required_profit(results):
  # we'll take the pandas DataFrame as argument

  means = results.mean()
  return means["storage_days"]/means["sales"]

required_profit(const_stand_results)
=>
35.430626927029806

required_profit(const_demand_results)
=>
6.36842105263158
```

Note that the two year length of our trial contributes to the storage costs (apparently because tanks build up in our warehouse). We can compute mean for 10 trials of 6 months, instead (still using the constant probabilities).

We get in this case:

```
six_month_ten_trials = make_trials(const_probs,trial_weeks=26,num_trials=20)

six_month_stand_results = report_trials(stand_order,six_month_ten_trials)
six_month_demand_results = report_trials(order_on_demand,six_month_ten_trials)
```

Computing the `required_profits` we find

```
required_profit(six_month_stand_results)
=>
22.224944320712694

required_profit(six_month_demand_results)
=>
6.239202657807309
```

Thus there is a substantial difference in the required profit per tank for the standing order strategy when comparing a six-month time period to a two-year time period.

But the required profit per tank for the on-demand order strategy was not substantially different for the six-month time period and the two-year time period.
