Danmarks
Tekniske
Universitet

Stochastic Simulation 02443
Exercises

by
Carlos Parra Marcelo (s192770)

Stochastic Simulation 02443
Technical University of Denmark
Handed in 24. June 2021

Contents

1 Exercise 1: Generation and testing of random numbers 1
1.1 Part 1 1
1.2 Part 2 7

2 Exercise 2: Sampling from discrete distributions 10
2.1 Part 1 10
2.2 Part 2 10
2.3 Part 3 11
2.4 Part 4 12

3 Exercise 3: Sampling from continuous distributions 13
3.1 Part 1 13
3.2 Part 2 15

4 Exercise 4: Discrete event simulation 16
4.1 Part 1 16
4.2 Part 2 16
4.3 Part 3 18
4.4 Part 4 20

5 Exercise 5: Variance reduction methods 22
5.1 Part 1 22
5.2 Part 2 22
5.3 Part 3 22
5.4 Part 4 23
5.5 Part 5 23

6 Exercise 6: Markov chain monte carlo 24
6.1 Part 1 24
6.2 Part 2 25

7 Exercise 7: Simulated annealing 27
7.1 Part 1 27

8 Exercise 8: Bootstrap 31
8.1 Part 1 31
8.2 Part 2 31
8.3 Part 3 31

1 Exercise 1: Generation and testing of random numbers

1.1 Part 1

Write a program implementing a linear congruential generator (LCG). Be sure that the program works correctly using only integer representation.

```python
def seedLCG(initVal):
    global rand
    rand = initVal

def lcg():
    a = 1140671485
    c = 128201163
    m = 2**24
    global rand
    rand = (a*rand + c) % m
    return rand / m

seedLCG(1)
```

(a) Generate 10.000 (pseudo-) random numbers and present these numbers in a histogramme (e.g. 10 classes).

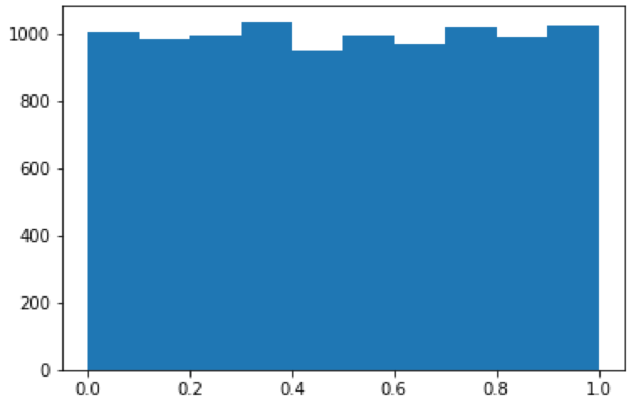

Figure 1: Histogram of pseudo-random numbers

(b) Evaluate the quality of the generator by graphical descriptive statistics (histogrammes, scatter plots) and statistical tests - X^2, Kolmogorov-Smirnov, run-tests, and correlation test.

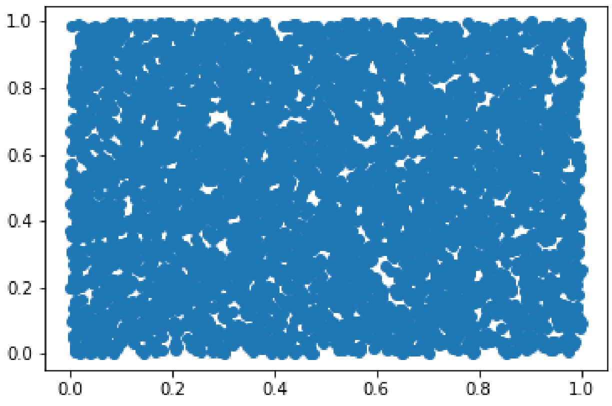

Figure 2: Scater plot of pseudo-random numbers

The generated numbers look quite sparse in the figure above, covering almost all the 2D grid.

If we perform the chi-squared test we get the result below:

```
observed = [1001  991 1000 1038  955  999  973 1023  993 1027]
expected = [1000 1000 1001  999 1000 1001 1000  999 1000 1000]

--- Chi-Squared test results ---
T = 5.718094104094105
critical value = 16.918977604620448
1 - cdf(T, k-1) = 0.7677673492359605
No significant difference detected
```

The hypothesis H_0 is accepted since no significant difference is detected.

I have some soubts about my implementation of the Kolmogorov-Smirkov test. The code is shown below:

```python
# Kolmogorv-Smirnov test
K_plus =[]
K_minus =[]
x = np.array(nums)

# sort the random numbers
x = np.sort(x)
n = len(nums)

# calculate (xj - (j-1)/n)
for j in range(1, n+1):
    term = x[j-1] - ((j-1)/n)
    K_minus.append(term)

# calculate (j/n - xj)
for j in range(1, n+1):
  term = (j/n) - x[j-1]
  K_plus.append(term)

# calculate (K+, K-)
```

```
result = (int(np.sqrt(n))*max(K_minus), int(np.sqrt(n))*max(K_plus))
print("Value of (K-, K+) is:")
print(result)

# compare it with the theoric values
ls = 1.94947 / np.sqrt(n) # value ks[+50,0.01] from table
if result[0] < ls:
    print('H0 is accepted (ks[+50,0.01])')
else:
    print('H0 is rejected (ks[+50,0.01])')
```

The output for the case of the proposed LCG is:

```
Value of (K-, K+) is:
(0.7841258049011257, 0.4272019672393823)
H0 is rejected (ks[+50,0.01])
```

The Hypothesis is being rejected. I have tried to arrive to other implementation from the slides but at this point, just before the second hand in, I can not manage anything better.

Moving forward, no correlation has been detected when plotting $U(i)$ against $U(i+1)$ in the image below:

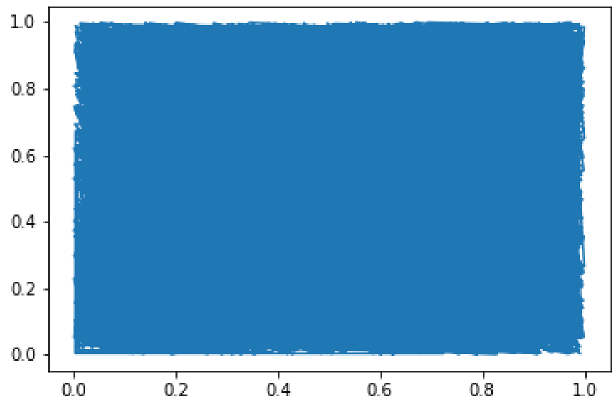

Figure 3: Visual correlation test

If we move to the different visual tests proposed, the figure below shows the normal distribution generated when performing the visual test I: above/below the median.

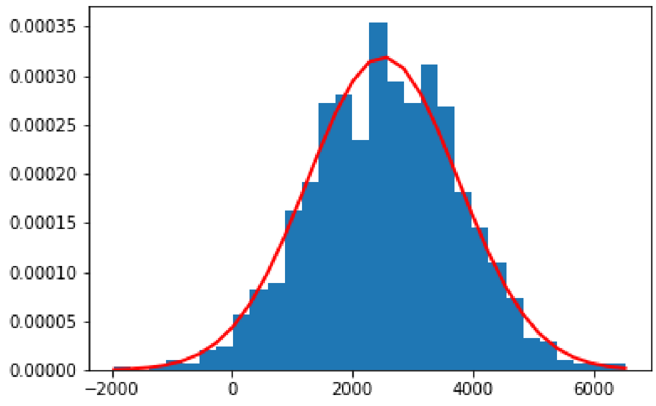

Figure 4: Distribution from Visual Test I: above/below the median

With has been generated from

$$T = n_1 + n_2 = 2502 + 2503 = 5005$$

where n_1 is the number of runs above the median and n_2 is the number of runs below the median.

Moving to Visual Test II: up/down from Knuth, we get this output from the execution:

```
Z = 1.57996
H0 is accepted (1.57996 < 1.63538)
```

Where the test statistic Z is computed using the formula from the lecture slides and compared with the value from the X^2 distribution for 6 degrees of freedom and 95% confidence.

For the case of the visual test III I am a bit stuck. This is my best try:

```python
# Test III: the-Up-and-Down test
n = len(nums)
R = np.zeros(10000-1, dtype=int)
i = 0
while i<n:
    length = 1
    next_idx = i + length
    if next_idx >= n:
        R[length-1] += 1
        break
    if nums[next_idx] > nums[next_idx-1]:
        while nums[next_idx] > nums[next_idx-1]:
            length += 1
            next_idx = i + length
            if next_idx >= n:
                break
    else:
        while nums[next_idx] < nums[next_idx-1]:
            length += 1
            next_idx = i + length
            if next_idx >= n:
                break
    R[length-1] += 1
    i = i+length

Z = (R-((2*n-1)/3)) / np.sqrt((16*n-29)/90)
print(np.mean(Z))
```

It returns a mean of -158.11 that makes no sense at all, but I followed the description from the slides and this is the implementation I understand. The correct result should be very close to 0.

Finally, the estimated correlation computed using the formula for the correlation coefficients prints the result below:

$$c_h = 0.25025$$

The value from c_h should be distributed as $N(0.25, 7/144n)$. Since the result is 0.24, we consider that the generator is also passing this test.

(c) Repeat (a) and (b) by experimenting with different values of "a", "b" and "M". In the end you should have a decent generator. Report at least one bad and your final choice.

As inspiration for possible values, I have used some of the ones mentioned in the paper *Finding near optimal parameters for Linear Congruentital Pseudorandom Number Generators by means of Evolutionary Computation*[1] by Hernndez J.C., Ribagorda, A., Isasi P., Sierra. J.M.

On one hand, I found a bad performance when trying with the values from the code below:

```
def lcg():
    a = 6364136223846793005
    c = 1
    m = 2**64
    global rand
    rand = (a*rand + c) % m
    return rand / m
```

I have tested the uniform distribution generated by the LCG with these parameters taking a look to its histogram and the X_2 test results.

[1]https://dl.acm.org/doi/pdf/10.5555/2955239.2955463

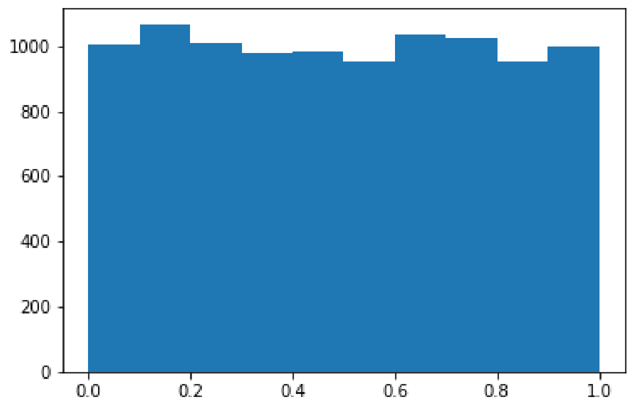

Figure 5: Histogram

```
observed = [1003 1066 1010  975  983  954 1037 1023  952  997]
expected = [1000 1000 1001  999 1000 1001 1000  999 1000 1000]

--- Chi-Squared test results ---
T = 11.776865440865441
critical value = 16.918977604620448
1 - cdf(T, k-1) = 0.22618287880819898
No significant difference detected
```

On the other hand, I found the best performance using the values from the code below:

```python
def lcg():
    a = 26977
    c = 10516
    m = 20817
    global rand
    rand = (a*rand + c) % m
    return rand / m
```

I have tested the uniform distribution generated by the LCG with these parameters taking a look to its histogram and the X_2 test results.

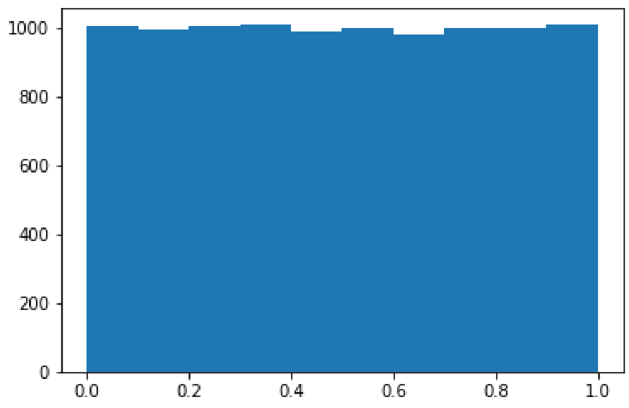

Figure 6: Histogram

```
observed = [ 999  996 1012 1008 1000  993  982  999 1001 1010]
expected = [1000 1000 1001  999 1000 1001 1000  999 1000 1000]

--- Chi-Squared test results ---
T = 0.707896265896266
critical value = 16.918977604620448
1 - cdf(T, k-1) = 0.999866267270547
No significant difference detected
```

Taking a look to the results, the second configuration proposed gets the best performance and this is easy to check with the X_2 test.

1.2 Part 2

Apply a system available generator and perform the various statistical tests you did under Part 1 point (b) for this generator too.

For this part of the exercise I have chosen the NumPy random number generator. The results for the different performed tests are printed below. Since the histogram and the chi-squared test has been already included in the previous part of the exercise, we jump directly to the visual tests (I have skipped Kolmogorov-Smirkov test this time because I think my implementation is wrong, as previously mentioned).

Some correlation has been detected when plotting $U(i)$ against $U(i+1)$ in the image below (take a look to the edges of the figure):

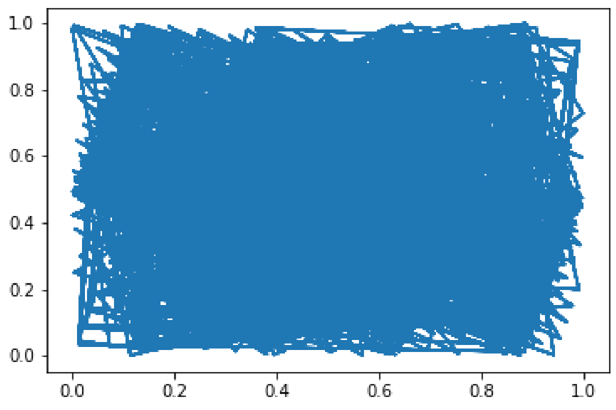

Figure 7: Visual correlation test

If we move to the different visual tests proposed, the figure below shows the normal distribution generated when performing the visual test I: above/below the median.

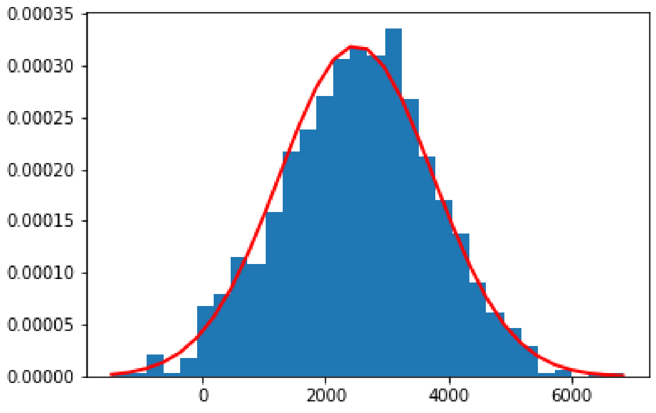

Figure 8: Distribution from Visual Test I: above/below the median

With has been generated from

$$T = n_1 + n_2 = 2500 + 2501 = 5001$$

where n_1 is the number of runs above the median and n_2 is the number of runs below the median.

Moving to Visual Test II: up/down from Knuth, we get this output from the execution:

```
Z = 28.69277
H0 is rejected (28.69277 > 1.63538)
```

Where the test statistic Z is computed using the formula from the lecture slides and compared with the value from the X^2 distribution for 6 degrees of freedom and 95% confidence. As the output shows, the random NumPy generator is not passing this test because the hypothesis is rejected.

Finally, the estimated correlation computed using the formula for the correlation coefficients prints the result below:

$$c_h = 0.24851$$

The value from c_h should be distributed as $N(0.25, 7/144n)$. Since the result is almost 0.25, we consider that the generator is passing this test.

The NumPy generator performs close to the other LCG used in this exercise, but it is important to say that it fails the Visual Test II.

2 Exercise 2: Sampling from discrete distributions

In the exercise you can use a build in procedure for generating random numbers. Compare the results obtained in simulations with expected results. Use histograms (and tests).

2.1 Part 1

Choose a value for the probability parameter p in the geometric distribution and simulate 10,000 outcomes. You can experiment with a small, moderate and large value if you like.

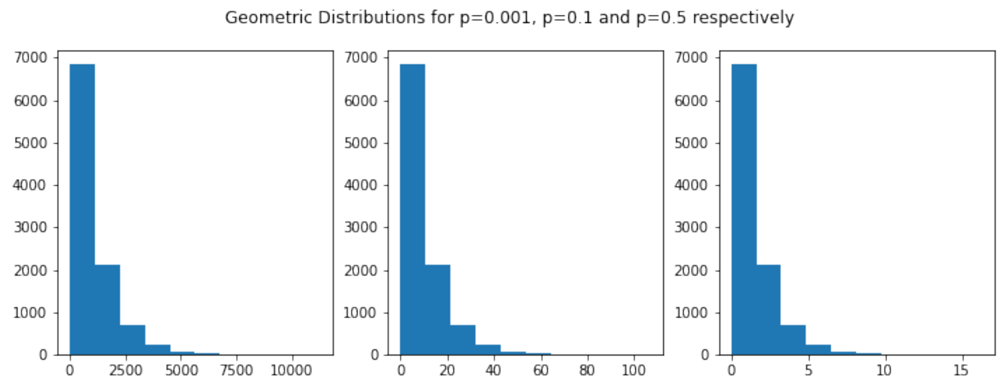

Figure 9: Geometric distributions

2.2 Part 2

Simulate the 6 point distribution with:

X	1	2	3	4	5	6
pi	7/48	5/48	1/8	1/16	1/4	5/16

(a) by applying a direct (crude) method

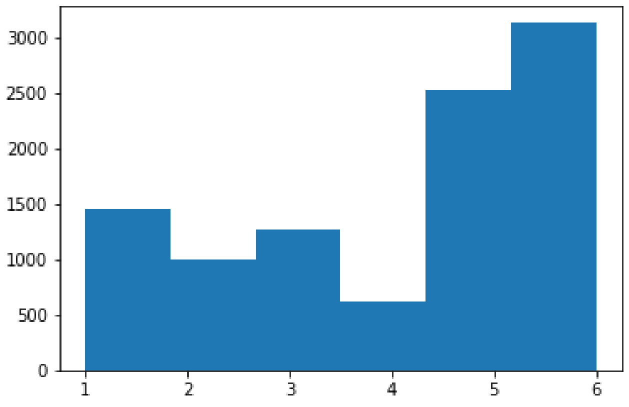

Figure 10: Distribution using crude-direct method

(b) by using the the rejection method

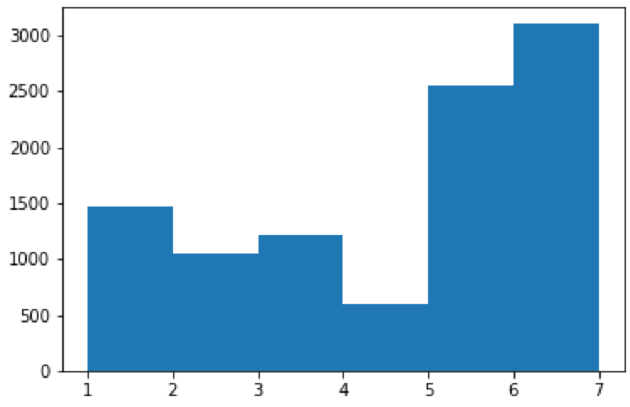

Figure 11: Distribution using rejection method

(c) by using the Alias method

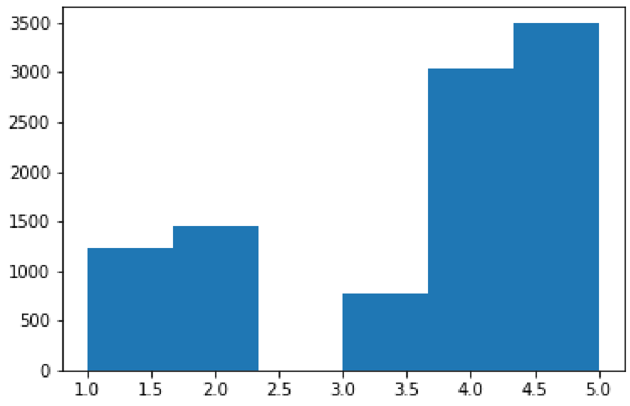

Figure 12: Distribution using Alias method

```
Alias Tables:
L = [5 4 4 5 5 0]
F = [0.875 0.625 0.75  0.375 0.875 1.   ]
```

2.3 Part 3

Compare the three different methods using adequate criteria, then discuss the results.

When performing the X^2 test I got these results:

```
Rejection method vs. crude method using chi-square test:
H0 rejected: Distributions significantly different
```

```
Alias method vs. crude method using chi-square test:
H0 rejected: Distributions significantly different
```

Both rejection and Alias methods are too different from the crude-direct one with 95% of confidence. It is clear in the case of the Alias method since it can be checked directly from the plots that is not working as good as expected by some reason. In the case of the rejection method they look very similar from the plots.

2.4 Part 4

Give recommendations of how to choose the best suited method in different settings, i.e., discuss the advantages and drawbacks of each method. If time permits substantiate by running experiments.

It is easy to check that the direct-crude and the rejection methods are very precise from their respective distribution plots.

The Alias method is much less precise for this concrete case because with only 6 bins for each histogram (6 different values the distribution can take) there are very few possibilities to fill each bin with content from others. This last method has an important computational cost when generating the tables but then is quite fast compared to the other 2 methods (referring only to the loop implementation).

3 Exercise 3: Sampling from continuous distributions

3.1 Part 1

Generate simulated values from the following distributions.

(a) Exponential distribution

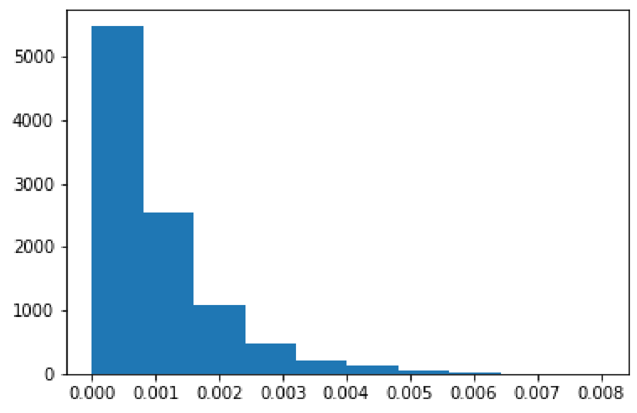

Figure 13: Exponential distribution

(b) Normal distribution (at least with standard Box-Mueller)

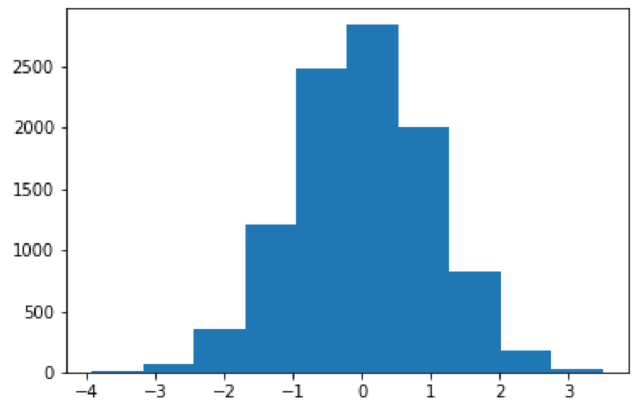

Figure 14: Normal distribution

(c) Pareto distribution, with $\beta = 1$ and experiment with different values of k values: $k = 2.05$, $k = 2.5$, $k = 3$ and $k = 4$.

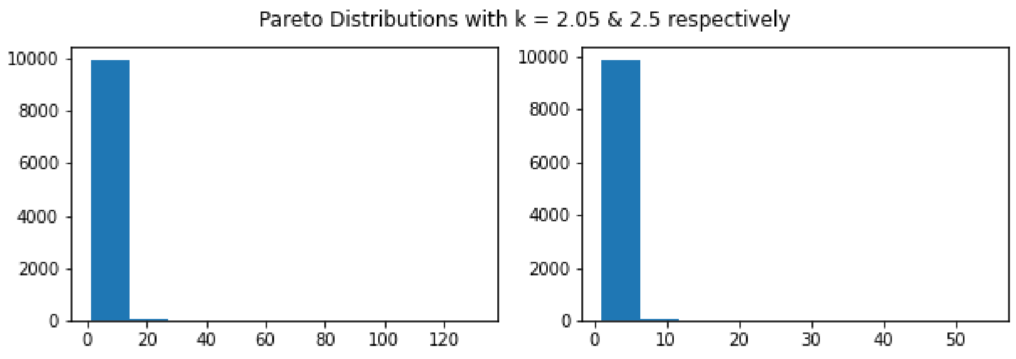

Figure 15: Pareto distributions for k=2.05 and k=2.5

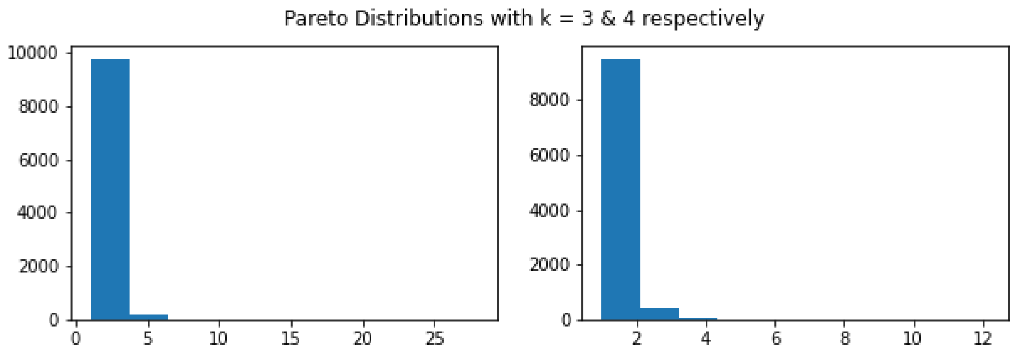

Figure 16: Pareto distributions for k=3 and k=4

Verify the results by comparing histograms with analytical results and perform tests for distribution type.

All the histograms look as they should compared to the original functions, but it is necessary to run the X^2 test for all of them to really check if they are enough similar to the analytical ones.

```
--- Chi-Squared test results ---
T = 0.0
critical value = 16.918977604620448
1 - cdf(T, k-1) = 1.0
No significant difference detected
```

The output from above corresponds to the exponential distribution.

```
--- Chi-Squared test results ---
T = 6.670436399135614
critical value = 16.918977604620448
1 - cdf(T, k-1) = 0.6713881063957532
No significant difference detected
```

The output from above corresponds to the normal distribution.

```
--- Chi-Squared test results ---
T = 8.801433241217302
critical value = 16.918977604620448
```

```
1 - cdf(T, k-1) = 0.45580204963894855
No significant difference detected

--- Chi-Squared test results ---
T = 18.442711553916
critical value = 16.918977604620448
1 - cdf(T, k-1) = 0.03037116867099343
Significant difference detected!

--- Chi-Squared test results ---
T = 24.72467031706121
critical value = 16.918977604620448
1 - cdf(T, k-1) = 0.0032913295652474694
Significant difference detected!

--- Chi-Squared test results ---
T = 16.636805692448814
critical value = 16.918977604620448
1 - cdf(T, k-1) = 0.054716794174227035
No significant difference detected
```

The outputs from above correspond to the 4 pareto distributions. Two of them fail the test (95% confidence), although they are very similar anyway.

3.2 Part 2

For the Pareto distribution with support on $[\beta, \infty]$ compare mean value and variance, with analytical results.

```python
# Pareto
beta = 1 #scale
k = 2.05 #shape
x = np.random.uniform(size=10000)
y = beta / (np.power(1-x, 1/k))
mu = np.mean(y)
var = np.var(y)

t_mu = (beta*k) / (k -1)
t_var = (k*np.power(beta,2)) / (np.power(k-1,2)*(k-2))

print(f'Computed mean = {mu} vs. Analytical mean = {t_mu}')
print(f'Computed variance = {var} vs. Analytical variance = {t_var}')
```

```
Computed mean = 1.919802 vs. Analytical mean = 1.952380
Computed variance = 4.421763 vs. Analytical variance = 37.188208
```

From the code and the output above we can check that the value for the variance is very dissimilar, meanwhile the value for the mean is almost the same in both cases.

4 Exercise 4: Discrete event simulation

Write a discrete event simulation program for a blocking system, i.e. a system with m service units and no waiting room. The offered traffic A is the product of the mean arrival rate and the mean service time.

4.1 Part 1

The arrival process is modelled as a Poisson process. Report the fraction of blocked customers, and a confidence interval for this fraction. Choose the service time distribution as exponential. Parameters: m = 10, mean service time = 8 time units, mean time between customers = 1 time unit (corresponding to an offered traffic of 8 erlang), 10 x 10.000 customers.

	Fraction of blocked	Average interarrival time	Total	Blocked
0	0.0005	0.993201	10001.0	5.0
1	0.0001	0.991801	10001.0	1.0
2	0.0002	0.996800	10001.0	2.0
3	0.0000	1.006099	10001.0	0.0
4	0.0000	0.986401	10001.0	0.0
5	0.0000	1.019698	10001.0	0.0
6	0.0002	1.004200	10001.0	2.0
7	0.0001	0.985001	10001.0	1.0
8	0.0004	0.989101	10001.0	4.0
9	0.0001	0.999000	10001.0	1.0

If we focus on some statistics about the fraction of blocked customers, we get:

```
mean                           0.00016
count                             10.0
std                           0.000171
CI low limit     [5.438206253347918e-05]
CI high limit    [0.0002655859406662009]
CI range                      0.000211
```

4.2 Part 2

The arrival process is modelled as a renewal process using the same parameters as in Part 1 when possible. Report the fraction of blocked customers, and a confidence interval for this fraction for at least the following two cases

(a) Experiment with Erlang distributed inter arrival times. The Erlang distribution should have a mean of 1

	Fraction of blocked	Average interarrival time	Total	Blocked
0	0.0	0.989077	10001.0	0.0
1	0.0	0.995992	10001.0	0.0
2	0.0	0.999760	10001.0	0.0
3	0.0	1.003446	10001.0	0.0
4	0.0	1.006480	10001.0	0.0
5	0.0	1.022823	10001.0	0.0
6	0.0	0.997917	10001.0	0.0
7	0.0	0.990758	10001.0	0.0
8	0.0	0.994180	10001.0	0.0
9	0.0	0.993435	10001.0	0.0

Since no customers are being blocked with this setup, it is not necessary to provide some statistics about the fraction. This is weird but I have been looking through the code and I did not find any bug. Anyway, below I have attached the function used to compute new arrival and service times:

```python
def gen_arrival_time(self):          #function to generate arrival times
    if self.arrival_mtd=='erlang':
        return (np.random.gamma(self.param_arrival)) # Erlang distribution (using Gamma with shape=int)
    elif self.arrival_mtd=='hyperexp': # Hyper Exponential distribution p1 = 0.8,  1  = 0.8333, p2 = 0.2,  2  = 5.0
        if np.random.uniform() <= 0.8: #p1
            return (np.random.exponential(scale=1./0.833)) # 1
        else: #p2
            return (np.random.exponential(scale=1./5.)) # 2
    else:
        return (np.random.poisson()) # Poisson distribution


def gen_service_time(self):          #function to generate service time
    if self.service_mtd=='constant':
        return self.param_service
    if self.service_mtd=='pareto': # Pareto distribution
        u = np.random.uniform()
        return (1 / np.power(u,(1/self.param_service)))
    if self.service_mtd=='normal':
        return (np.random.normal(loc=self.param_service)) # Normal Distribution
    else:
        return (np.random.exponential()) # Exponential distribution (lamda=1)
```

(b) hyper exponential inter arrival times. The parameters for the hyper exponential distribution should be $p_1 = 0.8$, $\lambda_1 = 0.8333$, $p_2 = 0.2$, $\lambda_2 = 5.0$.

	Fraction of blocked	Average interarrival time	Total	Blocked
0	0.0	0.988889	10001.0	0.0
1	0.0	0.990259	10001.0	0.0
2	0.0	1.014987	10001.0	0.0
3	0.0	1.006684	10001.0	0.0
4	0.0	1.009461	10001.0	0.0
5	0.0	1.008191	10001.0	0.0
6	0.0	1.002576	10001.0	0.0
7	0.0	0.999442	10001.0	0.0
8	0.0	0.978068	10001.0	0.0
9	0.0	1.012754	10001.0	0.0

Again, since no customers are being blocked with this setup, it is not necessary to provide some statistics about the fraction. This, again, is weird but I did not find any bug while checking.

4.3 Part 3

The arrival process is again a Poisson process like in Part 1. Experiment with different service time distributions with the same mean service time and m as in Part 1 and Part 2.

(a) Constant service time

	Fraction of blocked	Average interarrival time	Total	Blocked
0	0.108189	0.990401	10001.0	1082.0
1	0.108889	1.006199	10001.0	1089.0
2	0.106989	0.992801	10001.0	1070.0
3	0.112589	0.993501	10001.0	1126.0
4	0.106389	1.000600	10001.0	1064.0
5	0.100290	1.020798	10001.0	1003.0
6	0.102890	1.006699	10001.0	1029.0
7	0.104890	0.997900	10001.0	1049.0
8	0.110089	0.985201	10001.0	1101.0
9	0.104190	1.002800	10001.0	1042.0

If we focus on some statistics about the fraction of blocked customers, we get:

```
mean                        0.106539
count                           10.0
std                         0.003625
CI low limit     [0.10430386251083941]
CI high limit    [0.10877482961994751]
CI range                    0.004471
```

(b) Pareto distributed service times with at least k=1.05 and k=2.05.

First, we run the simulation for k=1.05:

	Fraction of blocked	**Average interarrival time**	**Total**	**Blocked**
0	0.191481	0.992401	10001.0	1915.0
1	0.169183	0.994601	10001.0	1692.0
2	0.191381	0.990201	10001.0	1914.0
3	0.214979	1.003600	10001.0	2150.0
4	0.150585	0.983402	10001.0	1506.0
5	0.077692	1.017398	10001.0	777.0
6	0.058194	1.002800	10001.0	582.0
7	0.159484	0.990701	10001.0	1595.0
8	0.097390	0.986601	10001.0	974.0
9	0.132187	1.001800	10001.0	1322.0

If we focus on some statistics about the fraction of blocked customers, we get:

```
mean                          0.144256
count                             10.0
std                           0.052202
CI low limit     [0.11206557168343016]
CI high limit    [0.17644557720168125]
CI range                       0.06438
```

Second, we run the simulation for k=2.05:

	Fraction of blocked	**Average interarrival time**	**Total**	**Blocked**
0	0.0024	0.992701	10001.0	24.0
1	0.0015	0.992201	10001.0	15.0
2	0.0010	0.997400	10001.0	10.0
3	0.0007	1.005999	10001.0	7.0
4	0.0011	0.986601	10001.0	11.0
5	0.0007	1.020198	10001.0	7.0
6	0.0007	1.004200	10001.0	7.0
7	0.0017	0.985201	10001.0	17.0
8	0.0011	0.989201	10001.0	11.0
9	0.0012	0.998700	10001.0	12.0

If we focus on some statistics about the fraction of blocked customers, we get:

```
mean                             0.00121
count                               10.0
std                             0.000536
CI low limit     [0.0008791776721693608]
CI high limit    [0.0015405803520282195]
CI range                        0.000661
```

(c) Choose one or two other distributions.

I have chosen a normal distribution with $\mu = 8$ and $\sigma = 1$:

	Fraction of blocked	Average interarrival time	Total	Blocked
0	0.129587	0.993301	10001.0	1296.0
1	0.124788	1.002800	10001.0	1248.0
2	0.130487	0.994601	10001.0	1305.0
3	0.131087	0.997700	10001.0	1311.0
4	0.126887	0.994201	10001.0	1269.0
5	0.122488	1.006599	10001.0	1225.0
6	0.128587	1.006699	10001.0	1286.0
7	0.133687	0.984902	10001.0	1337.0
8	0.130687	0.995500	10001.0	1307.0
9	0.125187	1.013199	10001.0	1252.0

If we focus on some statistics about the fraction of blocked customers, we get:

```
mean                          0.128347
count                             10.0
std                           0.003445
CI low limit         [0.12622312040599]
CI high limit      [0.13047121016095328]
CI range                      0.004248
```

4.4 Part 4

Compare confidence intervals for Parts 1, 2, and 3 and try explain differences if any.

run	mean	CI low limit	CI high limit	CI range
Q1 Poisson	0.000160	[5.438206e-05]	[0.000265]	0.000211
Q2 Erlang for arrival	0.000000	[0.0]	[0.0]	0.000000
Q2 Erlang for arrival	0.000000	[0.0]	[0.0]	0.000000
Q2 Hyperexponential for arrival	0.000000	[0.0]	[0.0]	0.000000
Q3 Constant for service	0.106539	[0.104303]	[0.108774]	0.004471
Q3 Pareto k=1.05 for service	0.144256	[0.112065]	[0.176445]	0.064380
Q3 Pareto k=2.05 for service	0.001210	[0.000879]	[0.001540]	0.000661
Q3 Normal for service	0.128347	[0.126223]	[0.130471]	0.004248

The last 3 columns from the table above show the CI limits for each of the performed runs.

For the first 3 simulations the arrival process is being modified, where only with the Poisson function some customers are being blocked (something weird as previously mentioned). This is due to the shape of this distribution, having its higher probabilities for lower values of x (closer to x=0). Erlang and exponential functions generate higher values, which in this case it would mean more time between arrivals (although is weird not to have customers being blocked).

For the last 4 runs, corresponding to Part 3, the service time is being modified. The function making a bigger impact on the performance is Pareto with k=1.05. Compared to Pareto with k=2.05, with k=1.05 the distribution is more flat and tends to give higher values (less close to 0) than with k=1.05. Higher values from these distributions are translated as higher service times for the customers, increasing the blocking probability and its CI interval. In addition to this, it is logic to have a very similar CI for the constant value and Normal distribution for a high number of simulated customers (since the normal distribution will be represented almost by its mean, which in this scenario is equal to the constant value).

5 Exercise 5: Variance reduction methods

5.1 Part 1

Estimate the integral by simulation (the crude Monte Carlo estimator). Use eg. an estimator based on 100 samples and present the result as the point estimator and a confidence interval.

```
u = np.random.uniform(size=100)
y = np.exp(u)
print(f'The crude Monte Carlo estimator: {sum(y)/len(y)}')
print(f'Variance: {np.var(y)}')
print(f'Confidence Interval: {st.t.interval(0.95, len(y)-1, loc=np.mean(y),
    scale=st.sem(y))}')
```

```
The crude Monte Carlo estimator: 1.6855923521453624
Variance: 0.21286099221965205
Confidence Interval: (1.5935856213811088, 1.7775990829096173)
```

5.2 Part 2

Estimate the integral using antithetic variables, with comparable computer resources.

```
u = np.random.uniform(size=100)
y = (np.exp(u) + np.exp(1-u)) / 2
print(f'Estimator with Antithetic Variables: {sum(y)/len(y)}')
print(f'Variance: {np.var(y)}')
print(f'Confidence Interval: {st.t.interval(0.95, len(y)-1, loc=np.mean(y),
    scale=st.sem(y))}')
```

```
Estimator with Antithetic Variables: 1.7130601352043109
Variance: 0.0038336112935857107
Confidence Interval: (1.7007127329695086, 1.7254075374391131)
```

5.3 Part 3

Estimate the integral using a control variable, with comparable computer resources.

```
u = np.random.uniform(size=100)
x = np.exp(u)
y = u
cov = np.mean(x*y) - (np.mean(x)*np.mean(y))
c = -cov / np.var(y)
mu_y = np.mean(y)
z = x + c*(y-mu_y)
print(f'Estimator with Control Variable: {sum(z)/len(z)}')
print(f'Variance: {np.var(z)}')
print(f'Confidence Interval: {st.t.interval(0.95, len(z)-1, loc=np.mean(z),
    scale=st.sem(z))}')
```

```
Estimator with Control Variable: 1.6968809461356462
Variance: 0.003966635601629877
Confidence Interval: (1.684321146506549, 1.7094407457647411)
```

5.4 Part 4

Estimate the integral using stratified sampling, with comparable computer resources.

```python
def rd():
    return np.random.uniform()

y = []
for i in range(100):
    num = np.exp(rd()/5) + np.exp(1/5+rd()/5) + np.exp(2/5+rd()/5) + np.exp(3/5+
   rd()/5) + np.exp(4/5+rd()/5)
    y.append(num / 5)
y = np.asarray(y)
print(f'Estimator with Stratified Sampling: {sum(y)/len(y)}')
print(f'Variance: {np.var(y)}')
print(f'Confidence Interval: {st.t.interval(0.95, len(y)-1, loc=np.mean(y),
   scale=st.sem(y))}')
```

```
Estimator with Stratified Sampling: 1.724305330012399
Variance: 0.0024568404092535235
Confidence Interval: (1.7144207087962686, 1.7341899512285293)
```

5.5 Part 5

Use control variates to reduce the variance of the estimator in exercise 4 (Poisson arrivals).

run	std	CI low limit	CI high limit	CI range
Standard Poisson	0.000171	[5.438206e-05]	[0.000265]	0.000211
Poisson with Control Variates	0.000107	[-6.286625e-06]	[0.000126]	0.000133

As it can be checked from the table above, the standard deviation has been reduced from 0.000171 to only 0.000107, which means that the variance has been reduced from 2.9241e-08 to 1.1449e-08.

6 Exercise 6: Markov chain monte carlo

6.1 Part 1

The number of busy lines in a trunk group (Erlang system) is given by a truncated Poisson distribution. Generate values from this distribution by applying the Metropolis-Hastings algorithm, verify with a X^2-test. You can use the parameter values from exercise 4.

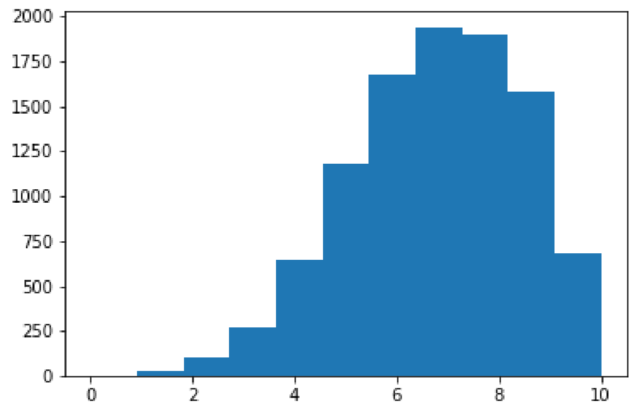

Figure 17: Erlang distribution using Metropolis-Hastings algorithm

The figure above shows the Erlang distribution generated from the Metropolis-Hastings algorithm. If we generate the original Erlang distribution using its analytical formula and we compare it with the previous one using a chi-squared test (95% confidence), we get:

```
x_observed=[   4   28   99  275  649 1182 1673 1936 1894 1577  683]
x_expected=[   6   33  128  337  685 1127 1510 1765 1710 1463 1236]

--- Chi-Squared test results ---
T = 71.1065780679218
critical value = 18.307038053275146
1 - cdf(T, k-1) = 2.7096658250513883e-11
Significant difference detected!
```

As we can check above, the distribution generated with the Metropolis-Hastings algorithm is not passing the test, although from the values of their histogram bins it is clear that they have the same shape. The bigger difference can be found in the last bin, were the observed value is almost the half than the expected one. This could be due to the jump between the state 0 and 10 when using the Markov chain. To demonstrate this hypothesis I have removed the jump in between states 0 and 10, forcing the algorithm to stay in the same state if it tries to perform that step. The resulting distribution is plotted below:

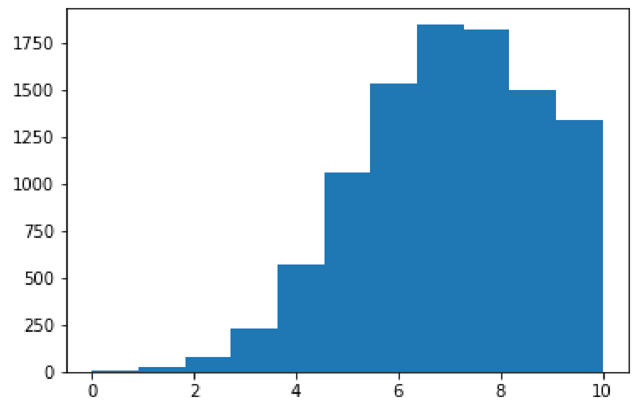

Figure 18: Erlang distribution using the new version of Metropolis-Hastings algorithm

Now, if we run the chi-squared test (95% confidence) and we check the bin values we get:

```
x_observed=[   7   23   81  231  567 1061 1535 1845 1817 1495 1338]
x_expected=[   6   33  128  337  685 1127 1510 1765 1710 1463 1236]

--- Chi-Squared test results ---
T = 26.19010645772049
critical value = 18.307038053275146
1 - cdf(T, k-1) = 0.0034927433701420485
Significant difference detected!
```

As shown above, the new distribution is closer to the original one (T=26.19 when before it was T=71.1) although it keeps failing the test.

6.2 Part 2

For two different call types the joint number of occupied lines is given by the formula from the slides. You can use A1, A2 = 4 and m = 10. Test the distribution fit with a X^2-test

(a) Use Metropolis-Hastings, directly to generate variates from this distribution.

The figure below shows the original 2-dimensional Erlang distribution generated using the direct-crude method:

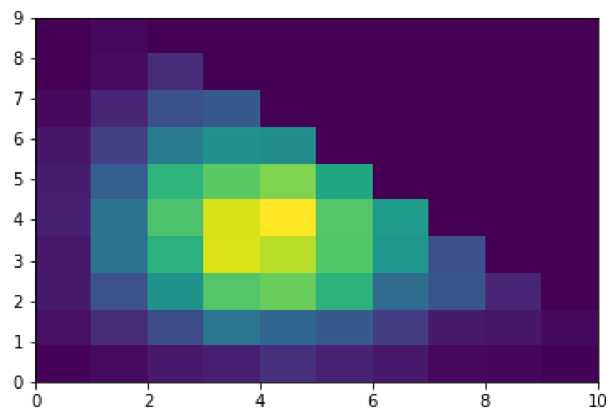

Figure 19: 2-dimensional Erlang distribution using the crude-direct method

Now, the figure below shows the 2-dimensional Erlang distribution generated using the Metropolis-Hastings algorithm:

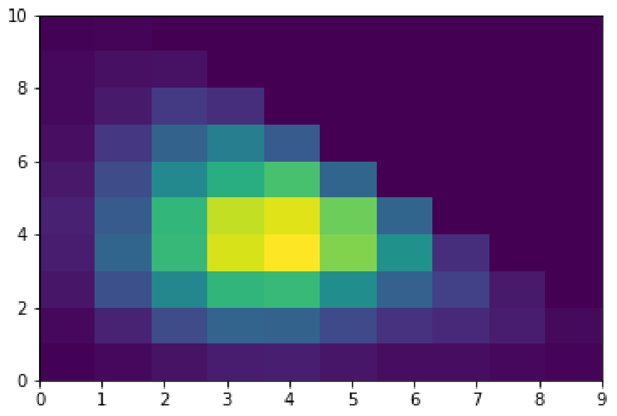

Figure 20: 2-dimensional Erlang distribution using the Metropolis-Hastings algorithm

They look very similar, but it is necessary to test their difference with the X^2-test:

```
--- Chi-Squared test results ---
T = 713.058935083138
critical value = 18.307038053275146
1 - cdf(T, k-1) = 0.0
Significant difference detected!
```

The test fails using 11 bins per dimension. I have tried different combinations for the bins and also other 2 different implementations of the Metropolis-Hastings algorithm but this is the best result I can achieve.

7 Exercise 7: Simulated annealing

7.1 Part 1

Implement simulated annealing for the travelling salesman. As proposal, permute two random stations on the route. As cooling scheme, you can use e.g. $T_k = 1/\sqrt{1+k}$ or $T_k = -log(k+1)$, but feel free to experiment with different choices. The route must end where it started. Initialise with a random permutation of stations.

For my implementation I have chosen $T_k = 1/\sqrt{1+k}$ to be my cooling scheme because I was not getting good results using the other proposal. In addition, I have used the proposed permutation for 2 stations to build each new route.

(a) Have input be positions in the plane of the n stations. Let the cost of going i to j be the Euclidean distance between station i and j. Plot the resulting route in the plane. Debug with stations on a circle.

My code generates random coordinates, in a map of size [0,100] for both dimensions, to locate 20 stations for the travelling salesman problem. I have configured the algorithm to run 2500 iterations to try to find the best solution. Some outputs from the run are attached below:

```
+------------------------ RESULTS -------------------------+
  initial temp: 100
    final temp: 0.020000
     max steps: 2500
    final step: 2500

  best energy: 401.670203
+-------------------------- END ---------------------------+
```

The two figures below show the initial generated route and the optimal (last) one respectively:

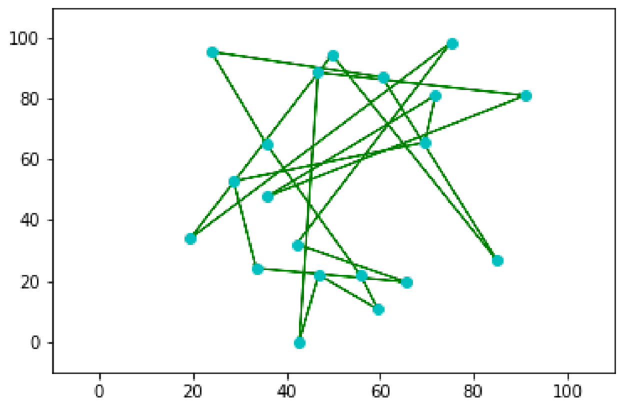

Figure 21: Initial random route

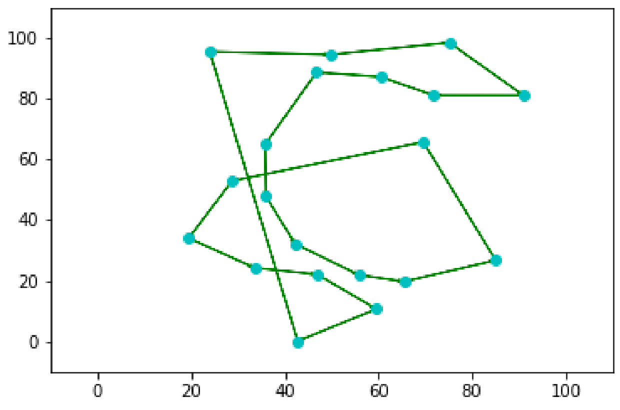

Figure 22: Optimal (last) route

To provide more information about how much the cost of the route has been reduced, the plot below shows evolution of the cost during the different iterations. This is very representative to show how the algorithm is able to solve the optimization problem:

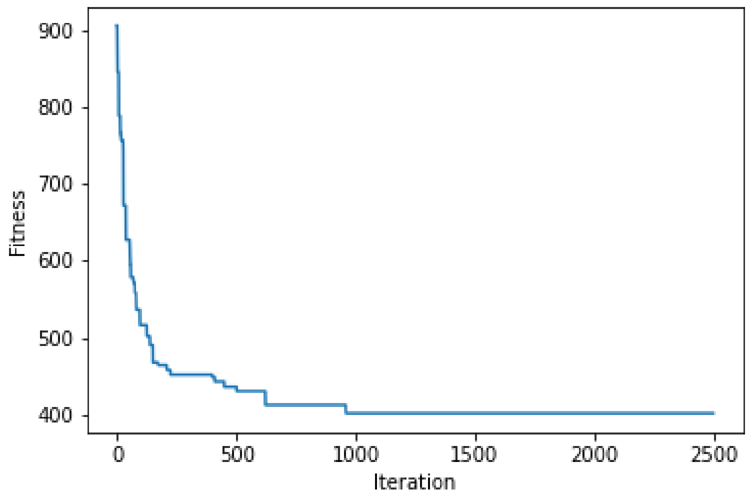

Figure 23: Learning curve

Where the initial cost of the route was approx. 900 and at the end of the run it has been reduced to 401.67. It important to notice that, checking the statistics from the algorithm, the best energy value found is the same as the one of the last route. This means that the algorithm has converged to an absolute minimum with high probability.

(b) Then modify your code to work with costs directly and apply it to the cost matrix from the course homepage.

Keeping the same configuration but using the proposed cost matrix for the cost function instead of the Euclidean distance, the output of the algorithm is:

```
+------------------------ RESULTS -------------------------+
  initial temp: 100
    final temp: 0.020000
     max steps: 2500
    final step: 2500

   best energy: 1475.000000
+-------------------------- END --------------------------+
```

Moreover, the plots for the first and last route are provided below respectively:

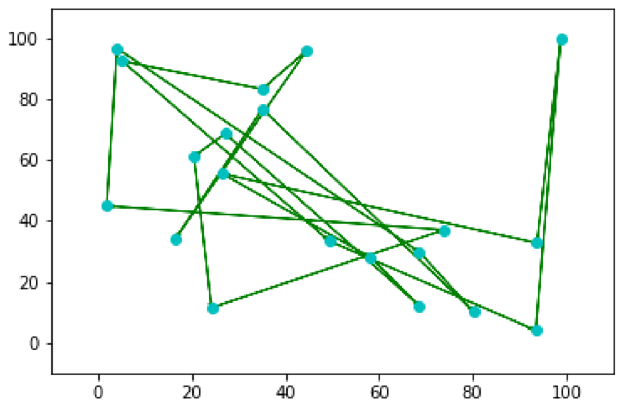

Figure 24: Initial random route

`Figures/ex7b_map_end.png`

Figure 25: Optimal (last) route

Of course, after the change applied to the cost function both plots are less visual because the Euclidean distance would be something similar to the cost function used for a real-life map. Anyway, by the learning curve plotted below we can say the algorithm is optimizing:

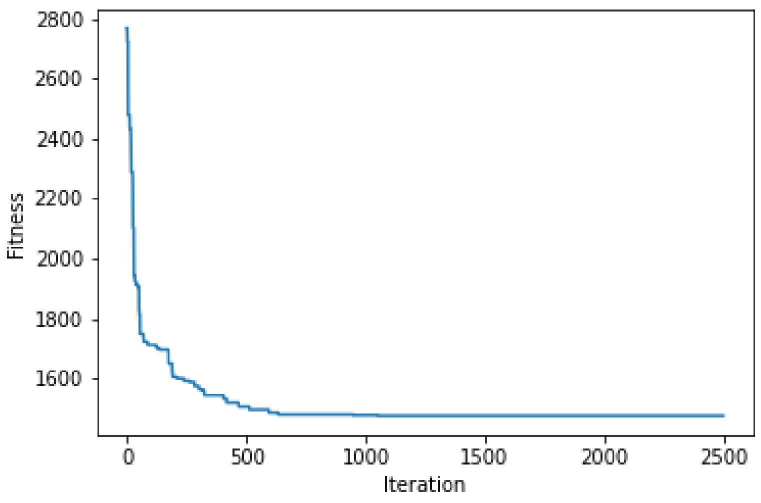

Figure 26: Learning curve

The cost of the initial route has been reduced from approx. less than 2800 to 1475. Once more, since the best energy is the same one computed for the last route, we can say that the algorithm has found the absolute minimum with high probability. However, since I have been performing several runs, I have notice that sometimes the algorithm discards the best energy state during the first iterations arriving to a local minimum instead. This is perfectly possible and is the main goal of every optimization problem: find the true absolute minimum but with some randomness to skip local minimums.

8 Exercise 8: Bootstrap

8.1 Part 1

Let X1,..., Xn be independent and identically distributed random variables having unknown mean μ. For given constants a < b, we are interested in estimating $p = P(a < sum(Xi)/n - \mu < b)$.

(a) Explain how we can use the bootstrap approach to estimate p.

The bootstrap method is a technique to estimate the variance of an estimator based on sampling from the empirical solution. For a concrete number of iterations, it takes N samples from the original dataset with replacement. For a relatively high number of iterations, we can use this algorithm to estimate the probability given by the estimator p. This is can done by checking if the condition $(a < sum(Xi)/N - \mu < b)$ for each of the generated samples and then simply divide the number of times the condition is matched by the total amount of iterations.

(b) Estimate p if n = 10 and the values of the Xi are 56, 101, 78, 67, 93, 87, 64, 72, 80, and 69. Take a = -5, b = 5.

```
p = 0.777
Variance using bootstrap algorithm: 0.17327
Confidence Interval = (0.75115,0.80284)
```

The result of the bootstrap method is printed above.

8.2 Part 2

If n = 15 and the data are 5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8

```
p = 1.0
Variance using bootstrap algorithm: 0.0
Confidence Interval = (1.0,1.0)
```

From the results above we can say that the condition is true in all the iterations, which makes sense in some way because in the previous case the variance of the data was 174.01 and now, with the new data (n=15), it is only 32.03.

8.3 Part 3

Write a subroutine that takes as input a "data" vector of observed values, and which outputs the median as well as the bootstrap estimate of the variance of the median, based on r = 100 bootstrap replicates. Simulate N = 200 Pareto distributed random variates with $\beta = 1$ and k = 1.05.

(a) Compute the mean and the median (of the sample)

```
Mean: 5.00271
Median: 2.02841
```

(b) Make the bootstrap estimate of the variance of the sample mean.

```
Bootstrap estimate of the variance of the sample mean: 0.15177
```

(c) Make the bootstrap estimate of the variance of the sample median.

```
Boostrap estimate of the variance of the sample median: 0.78527
```

(d) Compare the precision of the estimated median with the precision of the estimated mean.

By the definition of mean and median, it is logic to find that the estimate variance of the mean is lower than the estimated variance for the median. Since the different samples of length N=200 are being generated from a set of (also) N values, the median will change its value per iteration more than the average of an N-sequence.