02443 Stochastic Simulation, Exercises

Anton Ruby Larsen, s174356
Nicolaj Hans Nielsen, s184335

June 2020

Contents

1 Exercise 1 - Generation and testing of random numbers

1.1 Implementation of a LCG

LCG, Linar Congruential Generator, computes a sequence of independent pseudo random number. A given number x_i in the sequence is computed as:

$$x_i = ax_{i-1} + c \mod M$$

Here a is the multiplier, c is the shift and modulus, M. The LCG has full length if and only if the following conditions hold:

- M and c are relative prime
- The prime factors, p, of M must comply $a = 1 \mod p$
- if 4 is a factor of M, then $a = 1 \mod p$

We notice that M had to be quite great and the magnitude of c is not as important. To simply, we use $c = 1$ and $M = 2^x$ in that way we will always satisfy the first condition, though, we specifically set $M = 2^{33}$. Further, all prime factors of M are then 2 hence to satisfy condition 2, we simply require a to be odd. To satisfy the last condition we simply pick any number, k, that satisfy $k = 0 \mod 4$ and then set $a = k + 1$. We use $a = 562 \cdot 2 + 1$.

If we want the numbers distributed on $]0, 1[$, we use transform each x_i to $u_i = \frac{x_i}{M}$:

1.2 Test of LCG

In exercise 1.2, we are to test our LCG for uniformity by graphical, statistical, run and correlation tests. Theoretically details and test results will be described in the following subsection.

1.2.1 Graphical Tests

The simulated data will be plotted as a histogram and a scatter plot. All bins in the histogram should be as equal as possible while in the scatter plot no patterns should be noticeable.

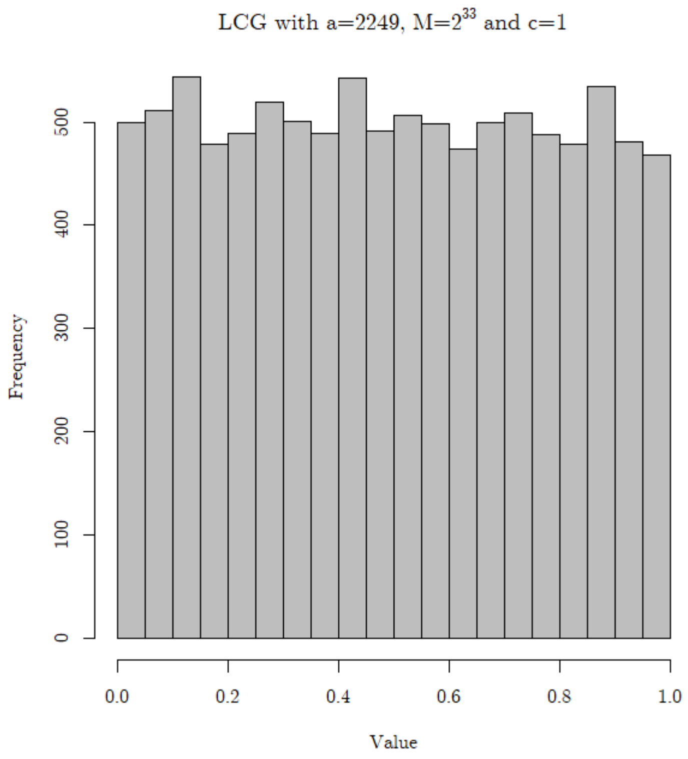


Figure 1: Histogram over 10,000 numbers generated from our LCG

In Figure 1 we see the all the simulated numbers plotted in a histogram. We cannot immediately reject that the data is uniformly distributed from this plot and we will therefore continue testing the LCG.

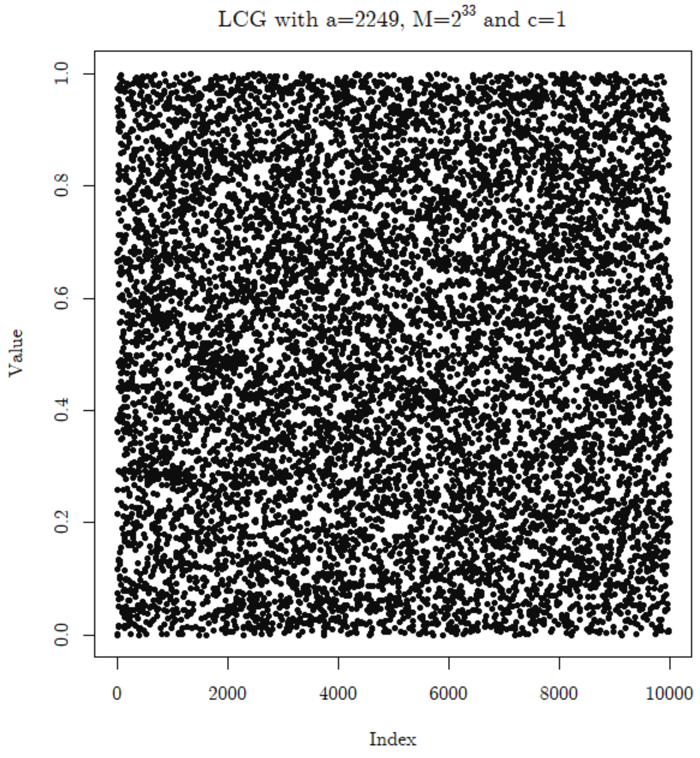


Figure 2: Scatter plot over 10,000 numbers generated from our LCG

In Figure 2 all generated points are plotted as a scatter plot. We do not see any pattern in all the points and therefore we cannot either here reject uniformity of the generator. We, therefore, move on to statistical tests.

1.2.2 Statistical tests

To test if the LCG samples are uniformly distributed, we use χ^2-test Kolmogov Smirnow test. First we will account for theoretical properties and empirical results of the χ^2-test.

$\chi^2 - test$: When we have some known underlying distribution of our data, we can use a χ^2-test to check if what we observe follows that distribution. We split our space up into bins and we can now model it as a multinomial distribution. Each bin will have X_j observed members and from the theoretical distribution we can calculate the expected np_j. We then divide by the standard deviation to normalize. By the central limit theorem we know that the observed mean should deviate from the expected following a normal distribution.

$$\frac{X_j - np_j}{\sqrt{np_j(1-p_j)}} \overset{asymp}{\sim} N(0,1)$$

We can now square the single bins and add them together to test all our observed data. When we square the single normal distribution and we get a χ^2 distribution with one degree of freedom.

$$\frac{(X_j - np_j)^2}{np_j(1-p_j)} \overset{asymp}{\sim} \chi^2(1)$$

When we add the bins and the denominator can be rewritten as np_j and we, therefore, have for the test statistic:

$$T = \sum_{i=1}^{k} \frac{(X_i - np_i)^2}{np_i} \overset{asymp}{\sim} \chi^2(k-1)$$

In our case where the underlying distribution is uniform, all p_j are equal and we therefore get

$$T_{unif} = \sum_{i=1}^{k} \frac{(X_i - n\frac{n}{k})^2}{n\frac{n}{k}}$$

We test with $n = 10000$, $k = 100$ and a significance level of 95%. We get the following theoretical 95% quantile and t-statistic.

$$T_{unif} = 94.52$$
$$\chi^2(99)_{0.95} = 123.225$$

We see that χ^2-test cannot reject that our LCG is uniform either.

Kolmogov Smirnow test: The Kolmogov Smirnow test is a non-parametric test as suppose to the χ^2-test which assumes the normality of the deviations from the groups. The Kolmogov Smirnow test compares the empirical distribution function(ECDF) to the cumulative distribution function(CDF) of the distribution we want to test for. Therefore in our case we test how much the ECDF differs from the CDF of a uniform distribution.

We get a test statistic of $D_n = 0.011464$ and the adjusted is equal to $\sqrt{n} + 0.12 + \frac{0.11}{\sqrt{n}} D_n = 1.147801$. We look up the value of the 95% quantile and find it to be 1.358 and therefore conclude that by this test we cannot either reject uniformity. In Figure 3 the ECDF of our LCG in black and the CDF of an uniform distribution in red is plotted.

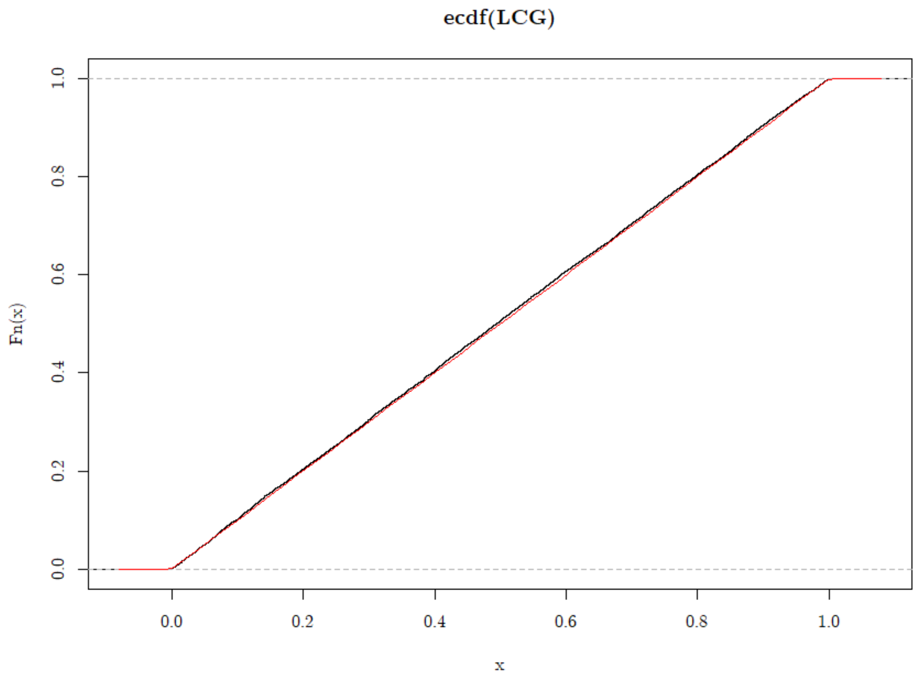


Figure 3: The ECDF of our LCG in black and the CDF of an uniform distribution in red

1.2.3 Run test

To check for independence, we use the run tests; above/below, Up/Down, and the-Up-and-Down.

Above/below We test the number of runs above, R_a and below, R_b the median. These are asymptotically normal distributed with:

$$(R_a + R_b) \sim N(2\frac{n_1 n_2}{n_1 + n_2} + 1,\ 2\frac{n_1 n_2(2n_1 n_2 - n_1 - n_2}{(n_1 + n_2)^2(n_1 + n_2 - 1})$$

Where n_1 is samples above and n_2 is samples below. The test statistics would be the observed number of runs below or above the median, i.e. $T = R_a + R_b$. The median was 0.5083, $n_1 = 5000$, $n_2 = 5000$

The test statistic is found to be 4962 and the 95% confidence interval for the statistic is [4903.01,5098.99]. We, therefore, see that the test statistic lies within the interval and we cannot reject that the data is uniform.

Up/Down Here we use a slightly different approach where we go through the sequence and split into increasing runs. We, therefore, start a new run each time we encounter an x_i where $x_{i-1} > x_i$. We then sort the runs according to their length and set them into the bins 1,2,..,5 and ≥ 6. We now have a vector, R, with all the frequencies which is used to compute the test statistics:

$$Z = \frac{1}{n-6}(R - nB)^T A(R - nB)$$

Here A is a special matrix seen on slide 19, Lecture 1, and likewise B is a special vector that can be seen on the same page.

If $n > 4000$ then Z is assumed to follow a $\chi^2(6)$ distribution.

In the exercise, we got:

$$R = \begin{bmatrix} 1705 & 2065 & 909 & 259 & 63 & 14 \end{bmatrix} \tag{1}$$

$Z = 2.560444$ and a 95% quantile for the $\chi^2(6)$ of 12.591587. Therefore, we cannot reject that the data is uniformly distributed.

The-Up-and-Down In this test, we count runs of either increasing or decreasing values. Therefore, we start a new run every time the "slope" of the run changes sign. The total number of runs is denoted X and the test statistic is calculated as follows:

$$Z = \frac{X - \frac{2n-1}{3}}{\sqrt{\frac{16n-29}{90}}}$$

which is assumed to be asymptotically distributed as $N(0, 1)$.

We get a total number of runs X to be 6621 and the test statistic is found to be $Z = -1.075272$. A 95% confidence interval for a standard normal distribution is ± 1.96 and we therefore observe that our test statistic lies within the range. We, therefore, have that we cannot reject that the data is uniformly distributed.

1.2.4 Correlation test

We can also compute the correlation coefficient as:

$$c_h = \frac{1}{n-h}\sum_{i=1}^{n-h} U_i U_{i+h}$$

which is assumed to be normal distributed with $N(\frac{1}{4}, \frac{7}{144n})$

We used $h = 1$ and computed $c_1 = 0.255370$ and with 95% quantiles [0.236141,0.263859] hence it passes this test.

We see that non of our tests was able to reject uniformity of our LCG and we can therefore with confidence trust that numbers generated by it are pseudo random.

1.3 Test of In-Build Mersenne-Twister PRNG In R

It is tested and passes all tests. Results can be seen in the attached code.

2 Exercise 2 - Sampling from discrete distributions

2.1 Simulation of a Geometric Distribution

In this exercise we are to simulate 10000 samples from a geometric distribution. A geometric distribution can be thought of as n independent Bernoulli trials. An example of a Bernoulli trials is a coin toss. We have some probability p for getting heads and 1-p for tails.

$$P(X = \text{head}) = p \qquad\qquad P(X = \text{tail}) = p - 1$$

For the geometric distribution we can model such n independent trials as:

$$f(n) = P(X = n) = (1-p)^{n-1}p$$

where the CDF for such density function is

$$CDF(n) = F(n) = P(X \leq n) = 1 - (1-p)^n$$

To simulate from random numbers on the interval]0,1[, we want the inverse of the CDF. So we solve for n in:

$$\begin{aligned} CDF(n-1) <&U \leq CDF(n) \longrightarrow \\ n - 1 <& \frac{\log(1-U)}{\log(1-p)} \leq n \end{aligned}$$

Because we want an integer output from the function we floor the function and add 1. Besides that, we also want low probabilities to correspond to less likely outcomes so in the nominator we change $\log(1-U)$ to $\log(U)$. We therefore get.

$$X = \lfloor \frac{\log(U)}{\log(1-p)} \rfloor + 1$$

We choose $p = 0.2$ and our outcome of the simulation can be read in Figure 4. As one can see they look a lot alike but to get a formal proof we run a χ^2-test. We get a test statistic of 0.004256 and the 95% quantile is 59.3. We therefore conclude that the simulated 10000 points follows the geometric distribution.

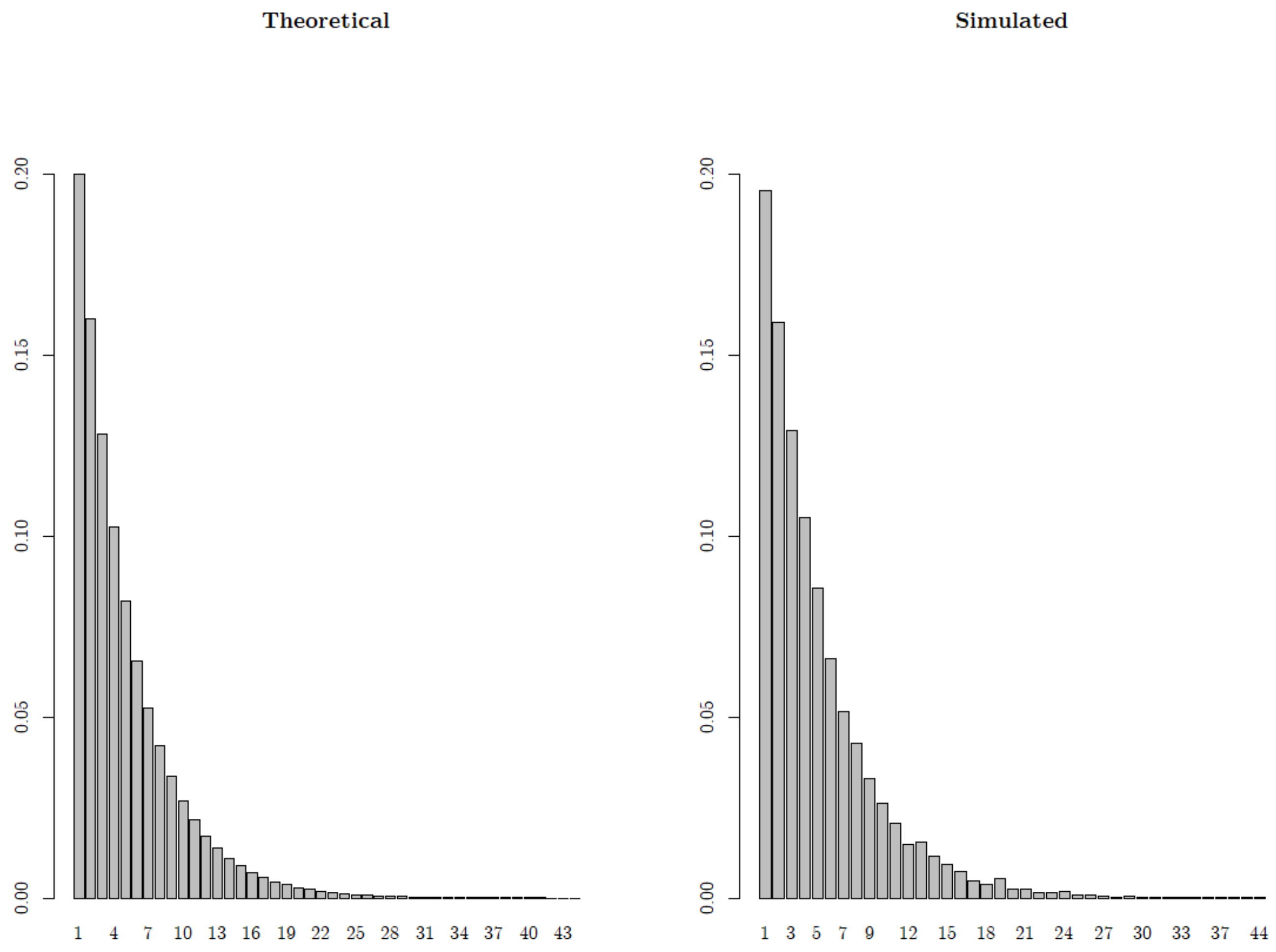


Figure 4: Histogram of the distribution of 10000 simulated geometric runs with p=0.2 and the corresponding theoretical distribution

2.2 Simulation of a 6 point distribution

Given the probabilities:

X	1	2	3	4	5	6
p_i	$\frac{7}{48}$	$\frac{5}{48}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{5}{16}$

we simulate the 6 point distribution with the direct method, rejection method and the Alias method.

Direct method In the direct - crude - method, the value of X is determined using the cumulative density function which we in our discrete case with p_i can formulate as:

$$\sum_{j=1}^{i-1} p_j < U \leq \sum_{j=1}^{i} p_j$$

which means that we will assign $X = x_i$ when we encounter the i interval in which the $U \sim Unif(0,1)$ falls.

Rejection method First we find a c that satisfies $\frac{p_i}{c} < 1$ and $c > p_i$.

For each U we compute:

$$I = 1 + \lfloor kU_1 \rfloor$$

We now only assign $X = x_I$ if:

$$U_2 < \frac{p_I}{c}$$

where U_2 is another sampled independent random number on]0, 1[.

We see that this method does add more computational complexity as we potentially have many rejections.

The Alias method The main idea here is to make a lookup table with F(I) and L(I) values. We can imagine that we first make 6 columns of height $\frac{1}{6}$. We then take each probability p_i and fill them in. If p_I is greater than $\frac{1}{6}$, then we simply write $F(I) = 1$ and $L(1) = 1$, however, the remaining $p_I - \frac{1}{6}$ has to go somewhere. This goes to another column where $p_j < \frac{1}{6}$ and $p_j + p_I - \frac{1}{6} = \frac{1}{6}$. We can then fill out the table as $F(j) = p_j$ and with $L(j) = I$.

Obviously, it takes some time to find the correct fit. However, with the tables generated, we can compute numbers of the distribution using:

$$I = 1 + \lfloor kU_1 \rfloor$$

And then assign $X = x_I$ if $U_2 < F(I)$ otherwise we assign $X = L(I)$

Test of methods Initially we look at the histograms where we have normalized each bin with the number of observations so that they are on the same axis as the theoretical:

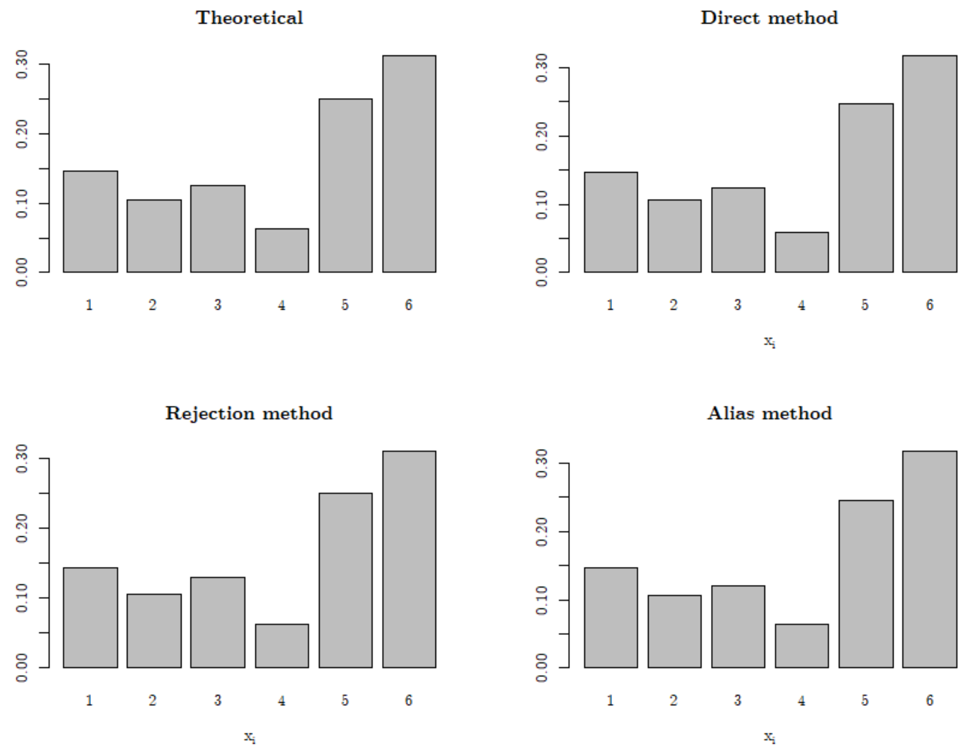


Figure 5: Normalized histograms of introduced methods

In figure 5 we see that all methods visually seems to follow the desired distribution.
The χ^2 test statistics for each of the methods are:

- Direct method: 0.0003805
- Rejection method: 0.0002586
- Alias method: 0.0004769

where the 0.95 quantile of the χ^2-distribution is 1.145, hence we cannot reject the null hypothesis that the simulated and theoretical distributions are alike.
We can also test the distribution using the Kolmogorovs distribution. The test statistics are:

- Direct method: 0.009673
- Rejection method: 0.006623
- Alias method: 0.01359

The 0.95 quantile of the Kolmogorovs distribution is 0.9990, hence we can not reject that the theoretical and simulated distributions are a like.

3 Exercise 3 - Sampling from continuous distributions

3.1 Simulation of Exponential, Normal and Pareto Distributions

We are to simulate samples from the exponential, normal and pareto distributions in this exercises. We start with the exponential distribution.

3.1.1 Exponential Distribution

The exponential distribution describes time between events in e.g. a Poisson distribution. The distribution is described by its density and CDF.

$$f(x;\lambda) = \lambda e^{-\lambda x}, \quad x \leq 0, \lambda > 0$$
$$F(x;\lambda) = 1 - e^{-\lambda x}, \quad x \leq 0, \lambda > 0$$

To go from U to X, we need the inverse of the CDF.

$$F(U;\lambda)^{-1} = -\frac{\log(1-U)}{\lambda}$$

To get low U-values to correspond to less likely outcomes, we swap 1-U with U.

$$X = -\frac{\log(U)}{\lambda}$$

We simulate 10000 samples with $\lambda = 1.1$. The distribution of the simulated samples is plotted in Figure 6 together with the theoretical distribution.

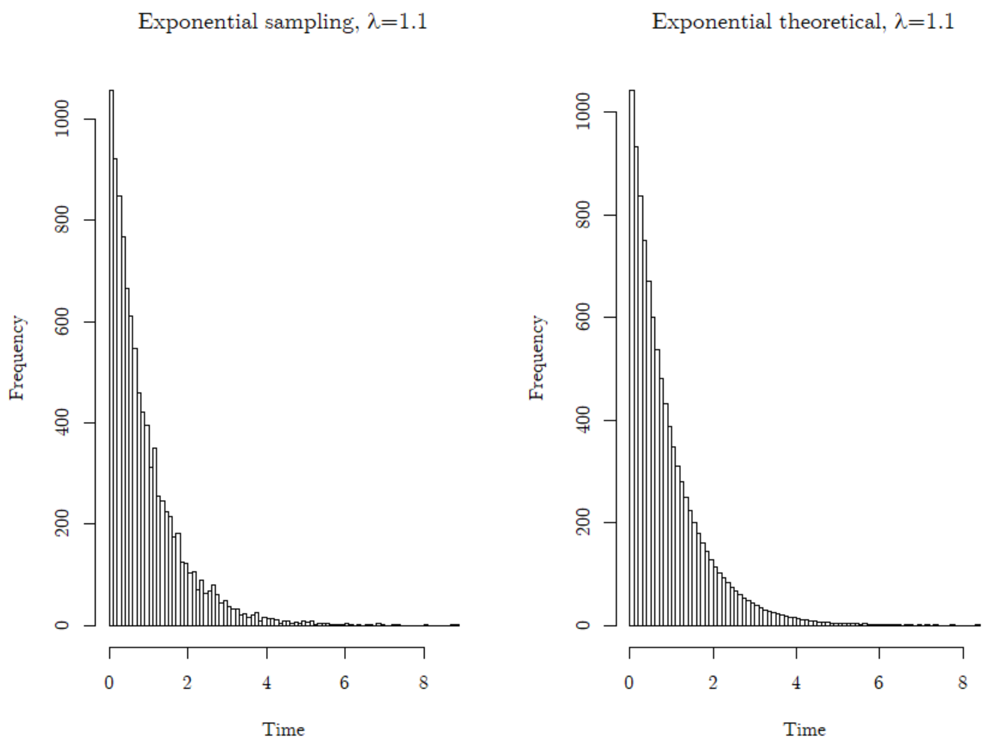


Figure 6: Histogram of the distribution of 10000 simulated samples from an exponential distribution with λ=1.1 and the corresponding theoretical distribution

In Figure 6 we see that the simulated samples look a lot like the theoretical. To build up this argument we will perform a Kolmogov Smirnow test. In Figure 7 the ECDF and the CDF is plotted for the exponential distribution with $\lambda = 1.1$ and we get an adjusted test statistic of 0.9504 and the 95% quantile is 1.358. Therefore, the p value is of 0.3285 and we cannot reject that the simulated samples follows an exponential distribution with $\lambda = 1.1$ as figure 6 also suggests.

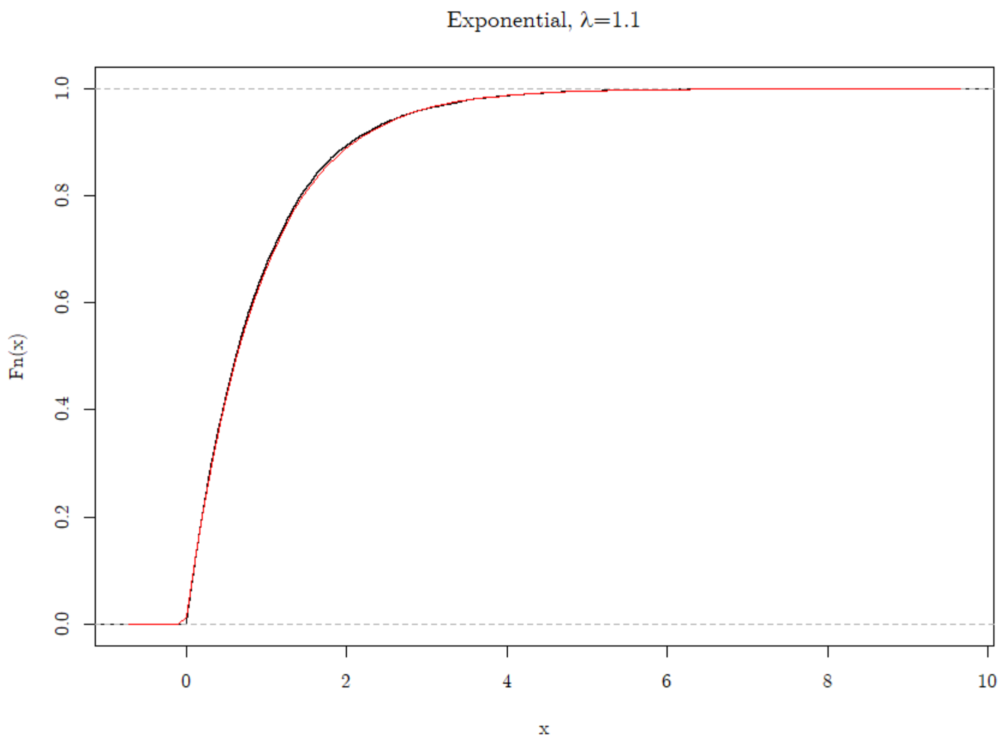


Figure 7: The ECDF for the simulated exponential distribution with $\lambda = 1.1$ plotted together with the CDF

3.1.2 Normal Distribution

The normal distribution is probably the best known continues distribution. To simulate it, we use the Box-Muller transformation due to its computational efficiency. It works by taking two samples from the uniform distribution on the interval]0, 1[and mapping them to two standard, normally distributed samples.

It is done by use of the following formula:

$$\begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} = \sqrt{-2\log(U_1)} \begin{bmatrix} \cos(2\pi U_2) \\ \sin(2\pi U_2) \end{bmatrix}, Z_1, Z_2 \sim N(0,1)$$

Where the cosine and sine are calculated by:

Algorithm 1 Cos and sin simulation

```
Generate $V_1, V_2 \sim U(-1, 1)$
Generate $R^2 = V_1^2 + V_2^2$
**while** $R^2 > 1$ **do**
    Generate $V_1, V_2 \sim U(-1, 1)$
    Generate $R^2 = V_1^2 + V_2^2$
**end while**
Return $\cos(2\pi U_2) = \frac{V_1}{R}, \sin(2\pi U_2) = \frac{V_2}{R}$
```

10000 samples where drawn by the simulation method and the distribution is shown in Figure 8 along side the theoretical distribution.

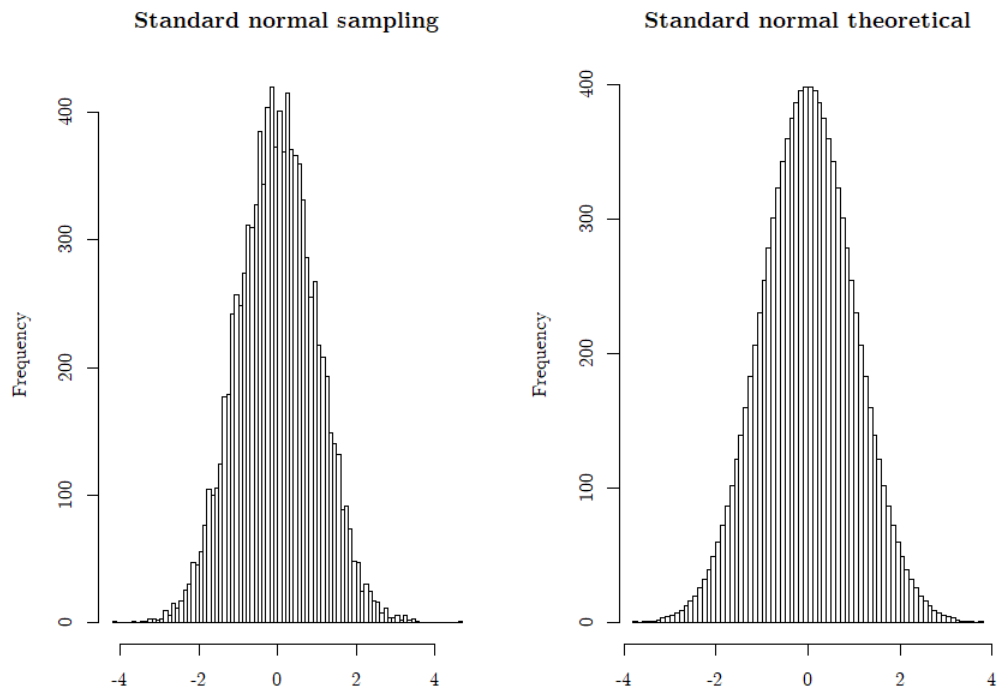


Figure 8: Histogram of the distribution of sampling from the Pareto distribution

In Figure 8 we see that the simulated samples look a lot like the theoretical. To build up this argument, we will perform a Kolmogov Smirnow test. In Figure 9 the ECDF and the CDF is plotted for the standard normal distribution. We get an adjusted test statistic of 0.9029 and the 95% quantile is 1.358. Therefore the p value is of 0.3903 and we cannot reject that the simulated samples are following a standard normal distribution as Figure 8 is also suggesting.

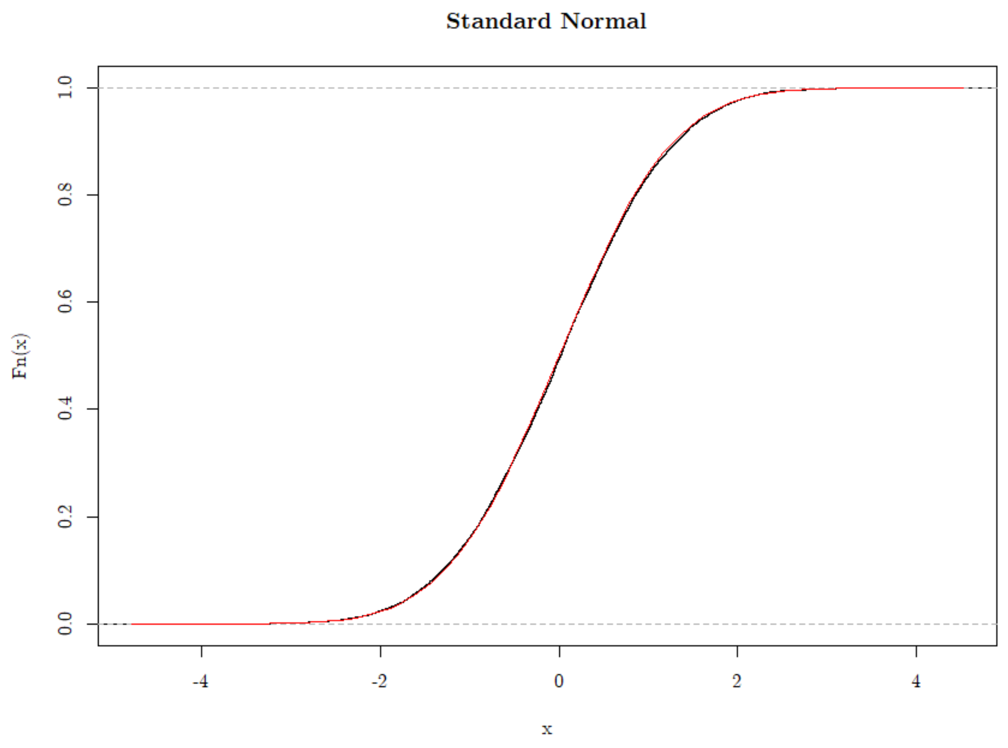


Figure 9: The ECDF for the simulated standard normal distribution plotted together with the CDF

3.1.3 Pareto Distribution

The Pareto distribution is a power-law distribution used in many fields to describe observable phenomena. Originally is was invented by the Italian engineer and economist Vilfredo Pareto to describe income distributions. The distribution is described by its density and CDF.

$$f(x;\beta;k) = \frac{k\beta^k}{x^{k+1}}, x \geq \beta$$
$$F(x;\beta;k) = 1 - (\frac{\beta}{x})^k, x \geq \beta$$

We have simulated 10000 samples with $\beta = 1$ and $k = \{2.05; 2.5; 3; 4\}$.

The distribution of sampling from the different Pareto distribution is shown in Figure 10 and their theoretical counterpart is shown in Figure 11.

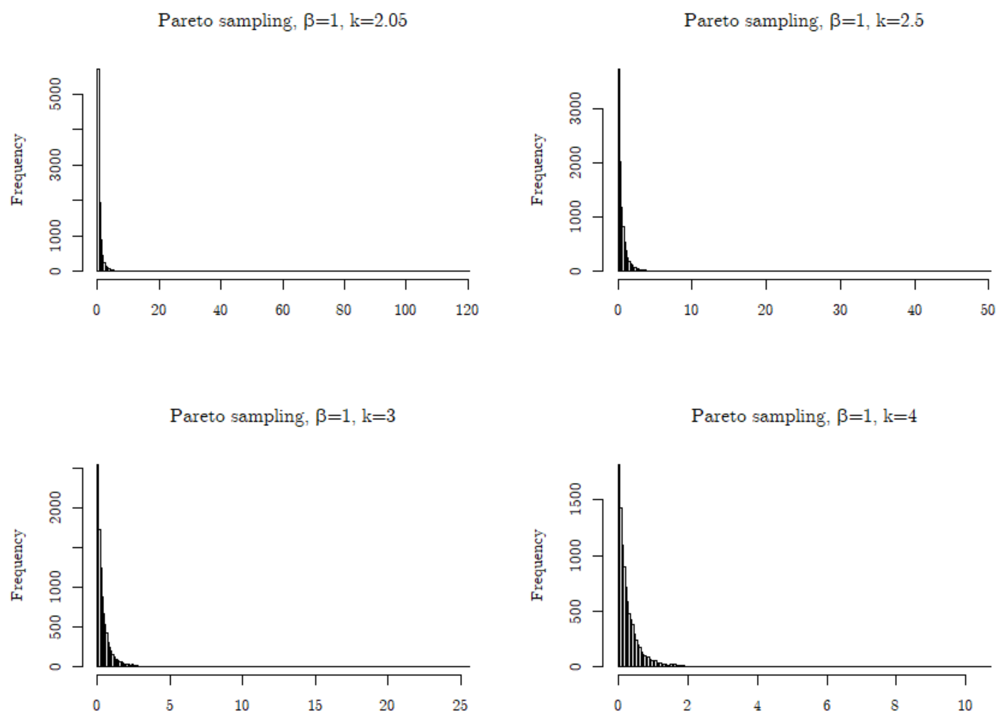


Figure 10: Histogram of the distribution of sampling from the Pareto distribution

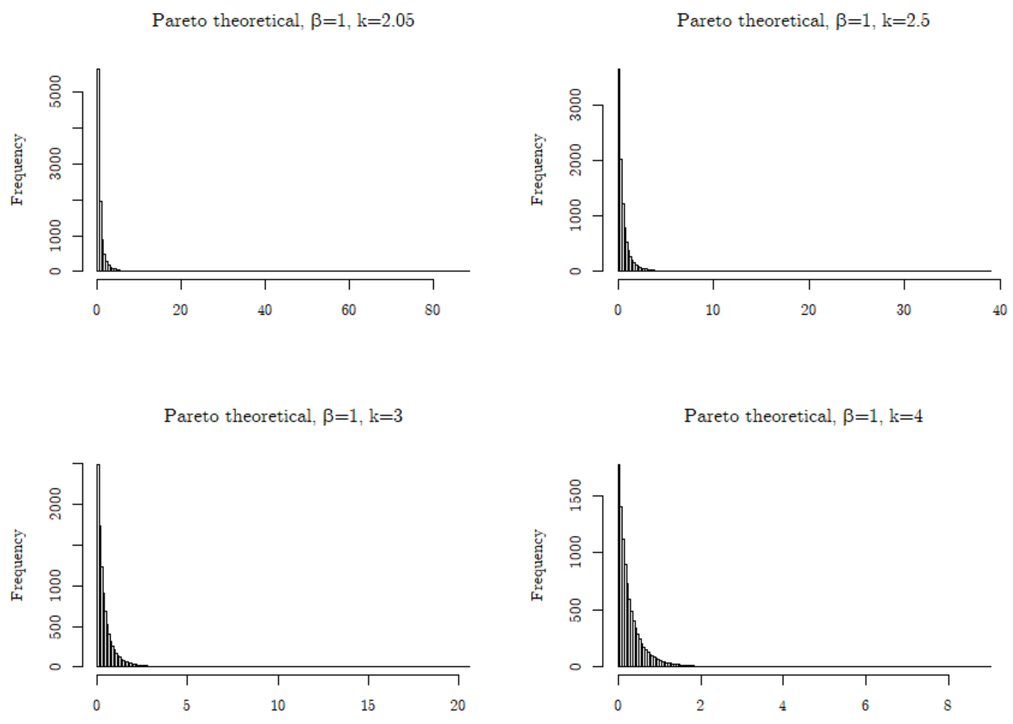


Figure 11: Histogram of the theoretical Pareto distributions

To build up the graphical similarities between the theoretical and the observed we perform a Kolmogov Smirnow test for each. The ECDF for each distribution is plotted together with the CDF in Figure 12. In Table 1 the adjusted test statistics, the 95% quantile and the p-value for each distribution is shown. We see all p values are well above 0.05 so we cannot reject that any of the simulated distribution are not following the proposed Pareto distributions.

Distribution	Adjusted test statistic	95% quantile	p-value
β=1, k=2.05	0.9504	1.358	0.3285
β=1, k=2.5	1.0710	1.358	0.2026
β=1, k=3	0.8240	1.358	0.5072
β=1, k=4	0.6946	1.358	0.7216

Table 1: Kolmogov Smirnow test for the single Pareto distributions

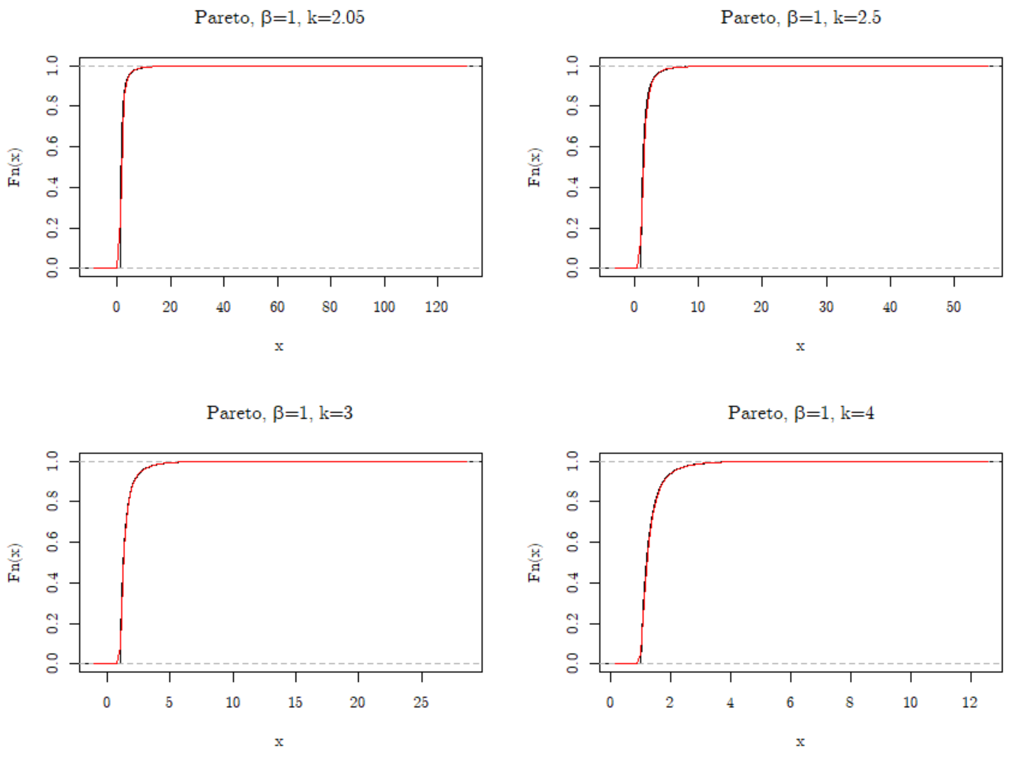


Figure 12: The ECDF for each Pareto distribution plotted together with the CDF

3.2 Comparing mean and variance of the Pareto distribution

The analytical mean and variance for the Pareto distribution is given by

$$E[X] = \beta \frac{k}{k-1}, \;\; k > 1 \qquad\qquad Var[X] = \beta^2 \frac{k}{(k-1)^2(k-2)}, \;\; k > 2$$

To compare the moments of our own simulation with the analytical we simulate 20000 samples and calculate the mean and variance of these. The results are presented in Table 2.

	Simulated Mean	Simulated Variance	Analytical Mean	Analytical Variance
Beta = 1, k = 2.05	1.931572	5.3459753	1.952381	37.1882086
Beta = 1, k = 2.5	1.655538	1.5928157	1.666667	2.2222222
Beta = 1, k = 3	1.492674	0.6550193	1.500000	0.7500000
Beta = 1, k = 4	1.328926	0.2110406	1.333333	0.2222222

Table 2: Mean and variance for the single Pareto distributions

We see that the simulated mean of all our different settings for the Pareto distribution are pretty close to the analytical. For the variances the same cannot be said. We see for small values of k the sample variance is very far from the analytical. The reason for this is that k is the ”shape” parameter and the smaller k the heavier a tail our distribution will exhibit. In Figure 13 we have simulated 20000 samples with k=2.05 100 times. We see that some of the runs are far above the analytical variance while most are below. This shows how much weight single very large samples in the far end of the tail can have on the variance of the whole distribution. For this reason the analytical and the simulated variance for low values of k will deviate a lot.

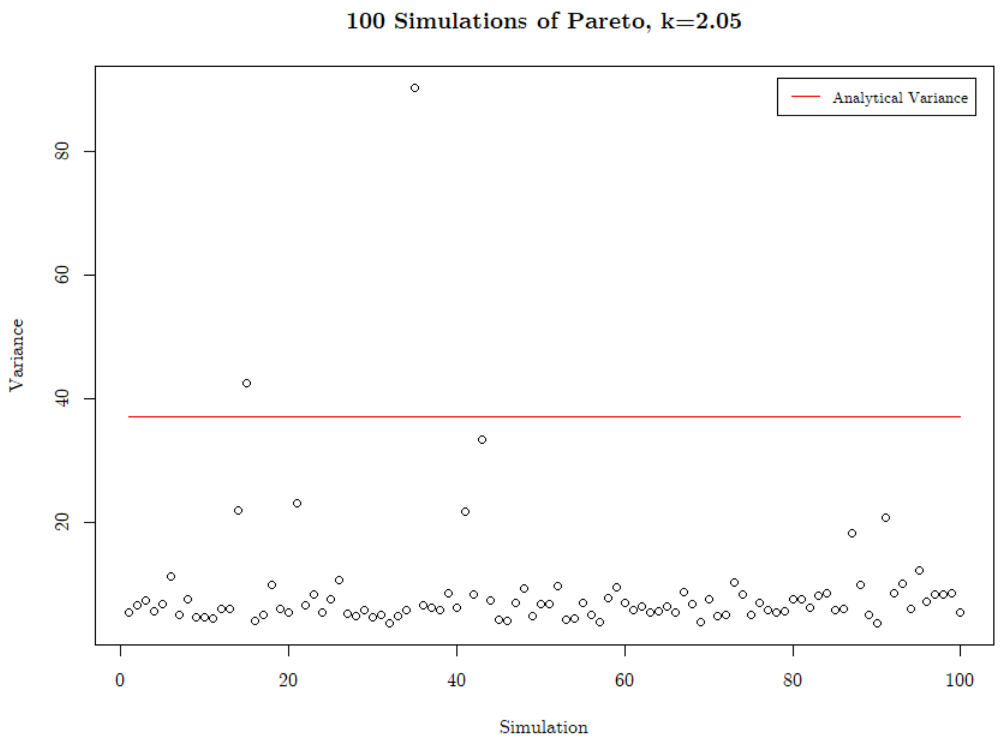


Figure 13: Scatter plot of 100 simulations with 20000 samples from the Pareto distribution with $\beta = 1$ and k=2.05

3.3 Normal Distribution continued

We now simulate 100 normal distribution using the polar Box-Mueller form. We sample 10 values each time and compute the confidence intervals for the mean and variance. For the mean we use the estimated mean and variance, $\hat{\mu}$ and s^2, and the 95% quantile of the t-distribution to compute the CI:

$$CI_{0.95} = \hat{\mu} \pm t_{\frac{\alpha}{2}}(n-1)\sqrt{\frac{s^2}{n}}$$

For the variance, we use a special expression, $\frac{(n-1)s^2}{\sigma^2}$, where σ^2 is the variance of the real distribution and $n = 10$. We know that the expression follows a χ^2-distribution with $n-1$ degrees of freedom. With the χ^2 distribution, we can now calculate the critical values for two-sided 95% significance level and setup the inequality:

$$\chi^2_{0.025} \leq \frac{(n-1)s^2}{\sigma^2} \leq \chi^2_{1-0.025}$$

Since we want a confidence interval for σ^2 we take the inverse which would flib the inequalities and isolate σ^2:

$$\frac{(10-1)s^2}{\chi^2_{1-0.025,9}} \leq \sigma^2 \leq \frac{(10-1)s^2}{\chi^2_{0.025,9}}$$

In other words have now found the 95% confidence interval of the variance:

$$[\frac{(10-1)s^2}{\chi^2_{1-0.025,9}}, \frac{(10-1)s^2}{\chi^2_{0.025,9}}]$$

We now plot the calculated confidence interval of the mean an variance along with the chosen, theoretical values of, $\mu = 0$ and $\sigma^2 = 1$:

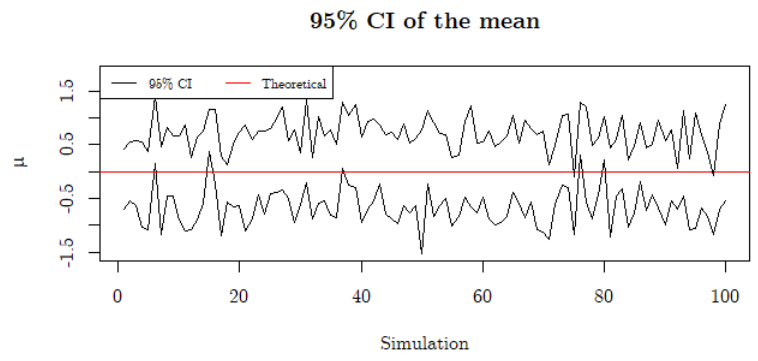


Figure 14: Computed CI of the mean with 100 simulations

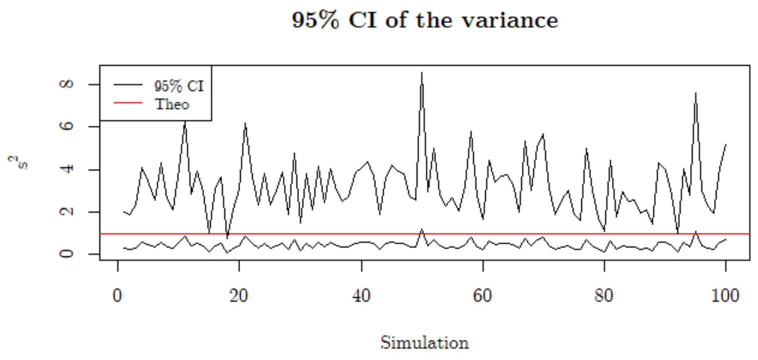


Figure 15: Computed CI of the variance with 100 simulations

We see in figure 14 that 0 is included in most of the CI. More specifically, it is included in 93 of the 100 cases. Likewise in figure 15 we have 1 included in 96 out of 100 simulations. In both cases, we have chosen a 95% interval hence we expect to have 0 and 1 included in the CI in $\frac{95}{100}100 = 95$ of the cases. We see that we deviate slightly from that but if we increase the number of simulations, it follows from the Central Limit Theorem that we would be inside the confidence interval 95% of the times.

4 Exercise 4 - Discrete Event Simulation

In this exercise we are to implement a discrete event simulation program for a blocking system. This system can be denoted as a $A(t)/S(t)/N/K$ queue where $A(t)$ is the arrival process, $S(t)$ is the service process, N is the number of service units in the system and K is the size of the waiting room. The waiting room in our case is 0 and therefore the model is also called a Erlang-B model.

We can picture the model as follows:

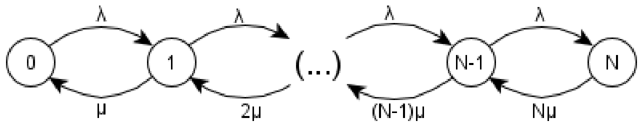


Figure 16: Model of an $A(t)/S(t)/N/0$ queue with no waiting room

In the model λ describes the mean of the arrival process and μ describes the mean of the service process.

For the model we could be interested in the average blocking probability and for some models we are able to derive this analytically. If our arrival process follows a Poisson process we are able to use the Erlang-B formula to derive the average blocking probability.

$$B = \frac{\frac{(\frac{\lambda}{\mu})^N}{N!}}{\sum_{i=0}^{N} \frac{(\frac{\lambda}{\mu})^i}{i!}} \tag{2}$$

If we are in the situation where the arrival process is not a Poisson process we can use numeric approximation. We sample M independent runs of the queue model and from these we can calculate the mean blocking probability as well as the confidence interval for the mean. An extra notation before we list the formulas to calculate the mean and the confidence interval is the total number of arrivals in a simulation which will be denoted *limit*.

$$\begin{aligned}
\hat{p}_{block_i} &= \frac{\sum_{j=1}^{limit_i} \mathbf{1}(x_j = block)}{limit_i} \\
\bar{p}_{block} &= \frac{\sum_{i=1}^{M} \hat{p}_{block_i}}{M} \\
S^2_{\bar{p}_{block}} &= \frac{1}{M-1} \sum_{i=1}^{M} (\hat{p}_{block_i} - \bar{p}_{block})^2 \\
CI &= \bar{p}_{block} \pm \frac{S_{\bar{p}_{block}}}{\sqrt{M}} t_{\alpha/2}(M-1)
\end{aligned}$$

Where we in the above assume normality of the mean of the block probabilities due to the use of the student-t distribution.

4.1 Simulations with different distributions

We now simulate with different arrival and service distributions. In the following we use $N = 10$, simulate $M = 10$ times with $limit = 10.000$ customers.

4.1.1 Poisson and exponential distribution

We start by modeling arrivals as an Poisson process with a mean time between cases of 1 time unit and the service times is assumed to be an exponential process with a mean of 8 time units.

In section 3.1.1 the exponential distribution is described and the mean is given as $E[\text{Exp}(\lambda)] = \frac{1}{\lambda}$. We want the expected value to be 8, hence we have $\lambda = 1/8$.

We will just briefly cover the Poisson distribution which is very often used Queuing theory. It is a discrete probability distribution that gives the probability of k events for non-overlapping intervals. The probability mass function is defined as:

$$P(\lambda, k) = \frac{(\lambda)^n}{k!} \exp^{-\lambda}$$

for $\lambda > 0$ and $k = 0,\ 1,\ 2, .., \ n$. Here λ is a rate parameter which sometimes is written as the product of the expected number of events per time unit and the length of the time interval, i.e., $\lambda = \frac{events}{time} \cdot length\_of\_interval$. We do, however, use the abbreviated version. For the Poisson distribution the expected value is given as, $\mathbf{E}[P(\lambda, k)] = \lambda$, hence $\lambda = 1$.

Our arrival process is in this case a Poisson process allowing us to calculate the mean blocking probability analytically by use of the Erlang-B formula given in equation 2. The average blocking probability is calculated to be 0.1217 which in our case with 10000 arrivals corresponds to 1217 blocked cases.

We compare the analytical result with the 10 simulations:

Simulation	Blocks	Blocking Probability
1	1284	0.1284
2	1296	0.1296
3	1195	0.1195
4	1368	0.1368
5	1215	0.1215
6	1291	0.1291
7	1353	0.1353
8	1361	0.1361
9	1320	0.1320
10	1320	0.1320

Using the blocked no. of individuals, the calculated confidence interval

$$\begin{aligned} \bar{p}_{block} =& 0.12636 \pm 0.005108 \\ =& [0.121252, 0.131468] \end{aligned}$$

We see the analytical result of 0.1217 just lies within the confidence interval which means our simulation follows the theory.

4.1.2 $A(t)$ as an Erlang process

We now repeat the above simulation but the arrival process is modeled as an Erlang process. The Erlang distribution is a special case of the gamma distribution $Gamma(k, \lambda)$ where k is restricted to the natural numbers $\mathbb{N}$. This is in other words the sum of k independent exponential distribution with mean $\frac{1}{\lambda}$.

The density function of the Erlang distribution is given by:

$$f(x; k; \lambda) = \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}, \quad x, \lambda \geq 0, \quad k \in \mathbb{N}$$

The mean of the distribution is given as:

$$E[x] = \frac{k}{\lambda}$$

We are to select the parameters of the distribution so the mean of the process equals 1. Therefore, we see that as long k equals λ the mean will be 1. We went with $k = \lambda = 1$ which also leads to the Erlang distribution reducing to an exponential distribution, $Exp(\lambda)$.

The service process is still the $Exp(\lambda = \frac{1}{8})$ and the 10 simulations resulted in the following.

Simulation	Blocks	Blocking Probability
1	1249	0.1249
2	1299	0.1299
3	1188	0.1188
4	1278	0.1278
5	1181	0.1181
6	1232	0.1232
7	1242	0.1242
8	1336	0.1336
9	1142	0.1142
10	1209	0.1209

The mean blocking probability and the confidence interval was found to be

$$\begin{aligned}\bar{p}_{block} =& 0.123560 \pm 0.004188 \\ =& [0.119372, 0.127748]\end{aligned}$$

4.1.3 $A(t)$ as a hyper-exponential process

The next arrival process we will take a look at is the hyper-exponential process. It consists of K exponential distributions each with p_i probability where $\sum_{i=1}^{K} p_i = 1$. So for each time we sample from the distribution a random number i generated to pick an exponential distribution and then a random number is chosen to generate a sample from the chosen exponential. The density function has the form:

$$f(x; \boldsymbol{p}; \boldsymbol{\lambda}) = \sum_{i=1}^{K} Exp(x, \lambda_i) p_i$$

And the mean of the process is

$$E[X] = \sum_{i=1}^{K} \frac{p_i}{\lambda_i}$$

We where given the set $\{p_1 = 0.8; \lambda_1 = 0.8333; p_2 = 0.2; \lambda_2 = 5\}$ to generate the hyper-exponential process from. We simulated 10 runs of 10000 arrivals as before. The service process was still $Exp(\lambda = \frac{1}{8})$ and we got the following results.

Simulation	Blocks	Blocking Probability
1	1110	0.1110
2	1122	0.1122
3	1194	0.1194
4	1162	0.1162
5	1145	0.1145
6	1234	0.1234
7	1189	0.1189
8	1190	0.1190
9	1123	0.1123
10	1100	0.1100

And the estimated mean blocking probability with its confidence interval was found to be

$$\begin{aligned}\bar{p}_{block} =&0.11569 \pm 0.0031471\\ =&[0.1125429, 0.1188371]\end{aligned}$$

The confidence interval makes okay sense because the mean of the arrival process is 1 with the given parameters which is the same as the previous processes.

4.1.4 $A(t)$ as a Pareto process

The Pareto distribution which is what drives the Pareto process is described in section 3.1.3 so we will not dive into further details here.

We got two sets of parameters to simulate, $\{\beta = 1; k = 1.05\}$ and $\{\beta = 1; k = 2.05\}$. As discussed in section 3.2 such low values of k will lead to very unstable distribution which will probably result in wide confidence intervals.

Pareto(β=1,k=1.05)		
Simulation	Blocks	Blocking Probability
1	0	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0
7	0	0
8	0	0
9	0	0
10	0	0

We see from the above table that no blocks was encountered. This makes sense if we calculate the mean of the process.

$$E[X_{Pareto}] = \beta\frac{k}{k-1} = \frac{1}{0.05} = 20$$

So our service process is over twice as fast as our arrival process resulting in no blocks.

We take a look at the $k = 2.05$ case.

Pareto(β=1,k=2.05)		
Simulation	Blocks	Blocking Probability
1	26	0.0026
2	9	0.0009
3	19	0.0019
4	8	0.0008
5	15	0.0015
6	12	0.0012
7	17	0.0017
8	6	0.0006
9	12	0.0012
10	9	0.0009

From the above table, we see the simulation results where some blocks are present but much less than in the other processes which is weird if we take a look at the mean of the process.

$$E[X_{Pareto}] = \beta\frac{k}{k-1} = \frac{1}{1.05} = \frac{20}{21}$$

All the other processes had a similar mean but the variance of this process is much higher than all the previous encountered. In section 3.2 it was calculated to be 37.188 which means that once in a while there will be over 4 times the mean service time between two arrivals resulting in freeing up service units. If the number of service units had been lower the difference between the $Pareto(\beta = 1, k = 2.05)$ and the previous processes would not be this large.

The mean and confidence interval of the $Pareto(\beta = 1, k = 2.05)$ is:

$$\begin{aligned}\bar{p}_{block} =& 0.001330 \pm 0.000435\\ =& [0.000895, 0.001765]\end{aligned}$$

4.1.5 $A(t)$ as a χ^2 process

Lastly we will try to model the arrival process with a χ^2 process. The theory behind this process can be found in section 1.2.2 so we will skip it here. We chose a process with 1 degree of freedom and the following results where produced with the service process still being the $Exp(\lambda = \frac{1}{8})$:

Simulation	Blocks	Blocking Probability
1	1623	0.1623
2	1733	0.1733
3	1668	0.1668
4	1711	0.1711
5	1705	0.1705
6	1677	0.1677
7	1671	0.1671
8	1640	0.1640
9	1656	0.1656
10	1790	0.1790

And the mean blocking probability with its confidence interval was found to be

$$\begin{aligned}\bar{p}_{block} =& 0.168740 \pm 0.003505 \\ =& [0.165235, 0.172245]\end{aligned}$$

Which a bit higher than all the previous processes. The mean of a $\chi^2(df = 1)$ is 1 and the variance is 2 so there will on average be 4 arrivals per average service time. Therefore the high blocking probability makes sense.

5 Exercise 5 - Variance Reduction Methods

In the following we will estimate the $\int_0^1 e^x dx$ using different variance reduction methods with 100 samples.

5.1 Crude Monte Carlo estimator

We have the generic formula:

$$E[g(X)] = \int_{\mathbb{R}} g(x)f(x)dx \tag{3}$$

If we assume that $X \sim U(0,1)$, we can then find the pdf of this uniform distribution. For the general $U(a,b)$ case we have the pdf:

$$f(x) = \begin{cases} \frac{1}{b-a}, & \text{for } x \in [a,b] \\ 0, & \text{otherwise} \end{cases}$$

hence for $X \sim U(0,1)$ we have:

$$f(x) = \begin{cases} 1, & \text{for } x \in [0,1] \\ 0, & \text{otherwise} \end{cases}$$

From eq. 3 and with $f(x)$ we now have, $E[g(X)] = \int_0^1 g(x)1dx$. We now let $g(x) = e^x$ and we can now rewrite it:

$$E[e^X] = \int_0^1 e^x dx \tag{4}$$

To estimate the integral we can use Monte Carlo simulation, i.e., generate n i.i.d. copies of $Y_i = e^X$ and then use the estimator $\bar{Y} = \frac{\sum_i^n Y_i}{n}$.

Using the estimator Y we, can use eq. 4 for analytical considerations of this estimator as $E[Y] = \int_0^1 e^x dx = e - 1 \approx 1.7182$. The variance of the estimator can be calculated as $Var[Y] = E[Y^2] - E[Y]^2$. We can find $E[Y^2]$ as:

$$E[Y^2] = \int_0^1 e^{2x} dx = \frac{1}{2}(e^2 - 1)$$

We can calculate the variance as $Var[Y] = \frac{1}{2}(e^2 - 1) - (e - 1)^2 = 0.2420$.

Using 100 simulation we obtained the estimator: $\bar{Y} = 1.7190 \pm 0097$ where we have a variance of $\hat{\sigma}^2 = 0.2456$. This is almost equal to the anticipated analytical results, however, we have to increase n to get a tighter confidence interval.

5.2 Antithetic variates

We are again to approximate the integral $\int_0^1 e^x dx$. This time we will apply the technique antithetic variates.

As seen in section 5.1 the integral $\int_0^1 e^x dx$ equals $E[e^U]$. Therefore, we define the process $X = e^U$. We then define n copies where $n = 2m,\ m \geq 1$, though an even number of copies. The process Y is then defined such that:

$$E[X(n)] = \frac{1}{n}\sum_{i=1}^{n} X_i = \frac{1}{m}\sum_{i=1}^{m} Y_i = E[Y(m)]$$

where

$$\begin{aligned}
Y_1 =& \frac{X_1 + X_2}{2} \\
Y_2 =& \frac{X_3 + X_4}{2} \\
&\vdots \\
Y_m =& \frac{X_{n-1} + X_n}{2}
\end{aligned}$$

We will now look at the variance of Y to get inside on how to choose the pairs (X_1, X_2).

$$\begin{aligned}
Var[Y] =& Var[\frac{X_1 + X_2}{2}] \\
=& Var[\frac{X_1}{2}] + Var[\frac{X_2}{2}] + 2Cov[\frac{X_1 + X_2}{2}] \\
=& \frac{1}{4}\sigma_X^2 + \frac{1}{4}\sigma_X^2 + \frac{1}{2}Cov[X_1, X_2] \\
=& \frac{1}{2}(\sigma_X^2 + Cov[X_1, X_2])
\end{aligned}$$

We observe that if (X_1, X_2) is chosen such that they are i.i.d. $Var[Y] = \frac{\sigma_X^2}{2}$, and therefore $Var[E[Y]] = \frac{\sigma_X^2}{n}$ as in the case of the crude monte carlo estimator. But if (X_1, X_2) is chosen such that $Cov[X_1, X_2] < 0$ then $Var[Y] < \frac{\sigma_X^2}{2}$. If the pairs are mutually independent, the central limit theorem still holds and $Var[E[Y]] < \frac{\sigma_X^2}{n}$.

In our case where $X = e^U$, the process follows a monotone function and thus (e^{U_i}, e^{1-U_i}) will be negatively correlated. Therefore we choose

$$Y_i = \frac{e^{U_i} + e^{1-U_i}}{2} = \frac{e^{U_i} + \frac{e^1}{e^{U_i}}}{2}$$

On slide 6, lecture 7 it is worked out analytically that this method will reduce the variance of $E[X]$ with 98% compared with the crude monte carlo estimator.

We implemented it in R and with n=100 we got:

$$E[e^U] = 1.709039 \pm 0.011203$$

And the variance was $\hat{\sigma}^2 = 0.00361124$.

5.3 Control variate method

The overall idea is still to estimate the mean of a random variable using Monte Carlo simulation. However, to reduce the variance, we introduce another random variable, Y_i, with a know mean, $E[Y_i]$, and a constant c:

$$Z_i = X_i + c(Y_i - E[Y_i])$$

A nice feature is that $E[Z_i] = E[X_i] + c([E[Y_i] - E[Y_i]) = E[X]$ and we have the variance:

$$Var[Z_i] = Var[X_i] + c^2 Var[Y_i] + 2cCov[X_i, Y_i] \tag{5}$$

If we saw this variance estimate as a function of c, $h(c)$, then we have $h'(c) = 2cVar[Y_i] + 2Cov[X_i, Y_i]$ which can be used to find the minimum c^*:

$$c^* = \frac{-Cov[X_i, Y_i]}{Var[Y_i]}$$

If we set this into eq. 5, we get the variance:

$$Var[Z_i] = Var[Z_i] - \frac{-Cov[X_i, Y_i]^2}{Var[Y_i]} \tag{6}$$

We, therefore, have to pick Y_i so that it has a high correlation with X_i and thereby high covariance. Further, to hold down the computational complexity, it would also be smart to choose Y_i so that it might be some part of the simulation already.

In practice, we often do not know the covariance of X_i and Y_i nor the variance of the X_i hence we simply determine them empirically.

In this specific case we have $X_i = e^{U_i}$, hence we could choose $Y_i = U_i$ and the control variate would be $(U_i - E[U_i]) = U_i - \frac{1}{2}$. For the variance and covariances, we simply use the simulation estimates. Using 100 simulations we get $\hat{Z}_i = 1.7417 \pm 0.011573$ with $\hat{\sigma}^2_{Z_i} = 0.002944275$ which is way better that the direct, crude estimate.

5.4 Stratified sampling

We have the stratification

$$Y_i = \frac{e^{\frac{U_{i,1}}{m}} + e^{\frac{1}{m} + \frac{U_{i,2}}{m}} + \ldots + e^{\frac{m-1}{m} + \frac{U_{i,m}}{m}}}{m}$$

We want to find the optimal number of stratas m to minimize $Var[E[e^U]]$ so we plot $m = [i \cdot 10], i = 1, 2, ..., 100$ and n = 100.

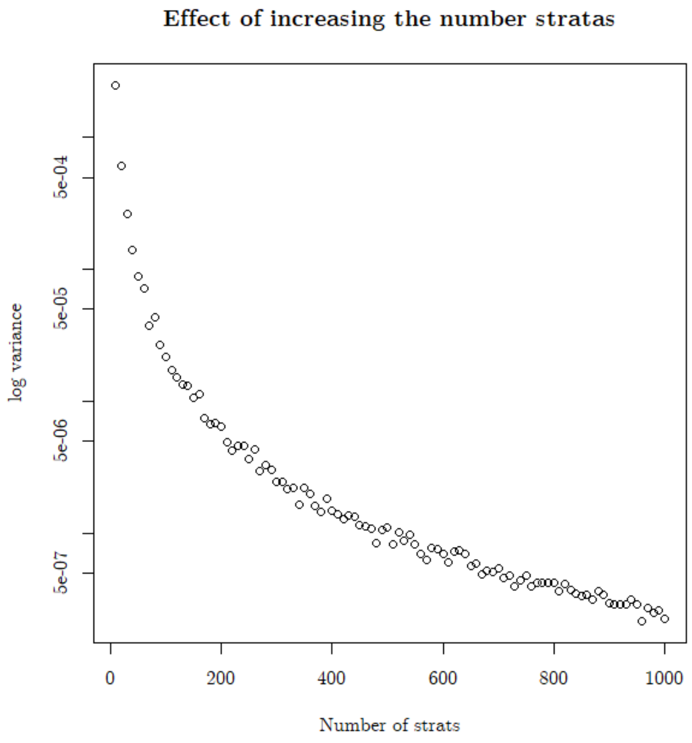


Figure 17: $Var[E[e^U]]$ for different m

We see that the variance keep decreasing which makes sense because it approximates the same way as a Riemann sum. We present a confidence interval m=1000 and n=100.

$$E[e^U] = 1.71821557174777 \pm 9.606151e - 05$$

And the variance is 2.402164e-07.

5.5 Control variates on blocking probability

In this section we will reduce the variance of the blocking estimator from exercise 4. Recall that the Erlang B-formula, eq. 2, is used to determine the theoretical blocking probability. From that we now that the blocking probability must increase if the mean arrival time, λ, is lowered or if we increase the mean service time, μ.

We now simulate and estimate the blocking probability, b_i, and try with several different control variates. Let A_i denote the arrivals and S_i denote the service process, then we could use:

$$Z_{i,1} = b_i + c(A_i - E[A_i])$$
$$Z_{i,2} = b_i + c(S_i - E[S_i])$$
$$Z_{i,3} = b_i + c(S_i - A_i - E[S_i - A_i])$$

The last comes from idea that the blocking probability is negatively correlated with arrivals and positively correlated with service times.

In our case we have $A \sim \text{Pois}(1)$ and $S \sim \text{Exp}(8)$, hence $E[A] = \lambda = 1$ and $E[S] = \frac{1}{\mu} = 8$ because $\mu = 8$. We also used 10 service units and a limit 10,000, 100 simulations. For each of the 100 simulation we take the mean service and arrival time

We can now use the formulas specified in section 5.3 and the 3 ideas specified above.

First we look at the correlation where we have:

Corr	S	A	A-S
B	0.6065	-0.6554	-0.68

hence we we could anticipate that the A-S combination could be the best. Using eq. 6 we can compute the reduced variance:

Reduced Var	S	A	A-S
B	$2.659 \cdot 10^{-5}$	$2.400 \cdot 10^{-5}$	$2.262 \cdot 10^{-5}$

With that we get the following estimates:

With A only:

$$\hat{Z}_1 = 0.12956 \pm 0.0009721 \text{ with } \hat{\sigma}_1 = 0.004899$$

With S only:

$$\hat{Z}_2 = 0.12913 \pm 0.001023 \text{ with } \hat{\sigma}_2 = 0.005157$$

With A & S:

$$\hat{Z}_3 = 0.12911 \pm 0.0009437 \text{ with } \hat{\sigma}_3 = 0.004756$$

We can, therefore, conclude that the best control variate is $Z_{i,3} = b_i + c(S_i - A_i - E[S_i - A_i])$ when we want to minimize our estimate of B.

6 Exercise 6 - Markov Chain Monte Carlo simulation

In this exercise we will be simulating by use of technique Markov Chain Monte Carlo(MCMC). The following is based on [1] and [2].

We will only work in a discrete state space $\{X_1, X_2, ..., X_n\}$ and a Markov chain in such space is define by:

$$p(X_t = x_t | X_{t-1} = x_{t-1}, ..., X_1 = x_1) = p(X_t = x_t | X_{t-1} = x_{t-1})$$

In the set of Markov chains we will only be working with time-homogeneous Markov chains meaning the transition matrix T is constant in time. For such we will define ergodicity.

Ergodicity If a Markov chain in the above settings is irreducible and has a stationary distribution π, then

$$\frac{1}{n}\sum_{i=1}^{n} f(X_i) \xrightarrow[n\to\infty]{} E[f(X)], \text{ where } X \sim \pi$$

for any bounded $f : X \to \mathbb{R}$.
Further if the Markov chain is aperiodic, the $P(X_n = x | X_0 = x_0) \xrightarrow[n\to\infty]{} \pi(x), \forall(x, x_0)$

In the above the terms irreducible, aperiodic, and a stationary distribution was introduced. We will explain them one by one in the following.

Irreducibility When all states belong to one communicating class, the Markov chain is said to irreducible. This means that for all times it is possible to reach all other states from all states with non-zero probability.

Aperiodicity Periodicity is defined by: Let $p_{ii}^{(n)}$ be the probability of returning to state i in n steps and $t = \{2, 3, ...\}$. Then state i is periodic with period t if

$$p_{ii}^{(n)} = 0 \text{ for } n \neq t, 2t, ...$$
$$p_{ii}^{(n)} \neq 0 \text{ for } n = t, 2t, ...$$

Aperiodicity is then the opposite and can be checked by:

$$GCD\{t : P(X_t = a | X_0 = a) > 0\} = 1$$

Stationary Distribution Lastly, a stationary distribution is defined by:

$$\pi = \pi T$$

where T is the transition matrix and π is the stationary distribution. If our Markov chain is finite, aperiodic and irreducible then one unique stationary distribution exists.

6.1 Metropolis Hasting simulation in one dimension

We are to simulate an Erlang system given by the distribution

$$P(i) = \frac{\frac{A^i}{i!}}{\sum_{j=0}^{n} \frac{A^j}{j!}}$$

Where $A = \frac{\lambda}{\mu} = \frac{1}{\frac{1}{8}} = 8$ and $n = 10$. To simulate the system we will be using the Metropolis Hasting algorithm but before we need to check if the system is ergodic.

We use a uniform (11×11) transition matrix to model the system in our Metropolis Hasting simulation:

$$T = \begin{bmatrix} \frac{1}{11} & \frac{1}{11} & \cdots & \frac{1}{11} \\ \frac{1}{11} & \frac{1}{11} & \cdots & \frac{1}{11} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{11} & \frac{1}{11} & \cdots & \frac{1}{11} \end{bmatrix}$$

We see that for all times all of our states are reachable with non-zero probability and therefore the Markov chain is irreducible. Further we see that in all states you are able to reach the state itself in one step. Therefore $GCD\{t : P(X_t = a|X_0 = a) > 0\} = 1$ and the Markov chain is also aperiodic.

Lastly we know a unique stationary distribution exists due the Markov chain being finite, irreducible, and aperiodic. Therefore the Markov chain is ergodic and we can use it to simulate the Erlang systems distribution.

Metropolis Hasting algorithm As mentioned we will use the Metropolis Hasting algorithm to simulate the system. This algorithm consists of two steps:

- Generate $Y \sim h(y, x)$ where h is the proposal distribution.
- Set $X_{t+1} = y$ with probability $\alpha(x, y) = min\{1, \frac{f(y)}{f(x)} \cdot \frac{h(y,x)}{h(x,y)}\}$ where $f(x)$ is the posterior distribution. Otherwise $X_{t+1} = x$

The posterior distribution in our case is the truncated Poisson distribution, $P(i)$, and the proposal distribution is a uniform 11-point distribution as illustrated in the transition matrix. Because the proposal distribution is symmetric $\alpha(x, y)$ reduces to $min\{1, \frac{f(y)}{f(x)}\}$

We simulated 100,000 iterations of the algorithm. In Figure 18 the simulated distribution in red is plotted together with the theoretical distribution in blue.

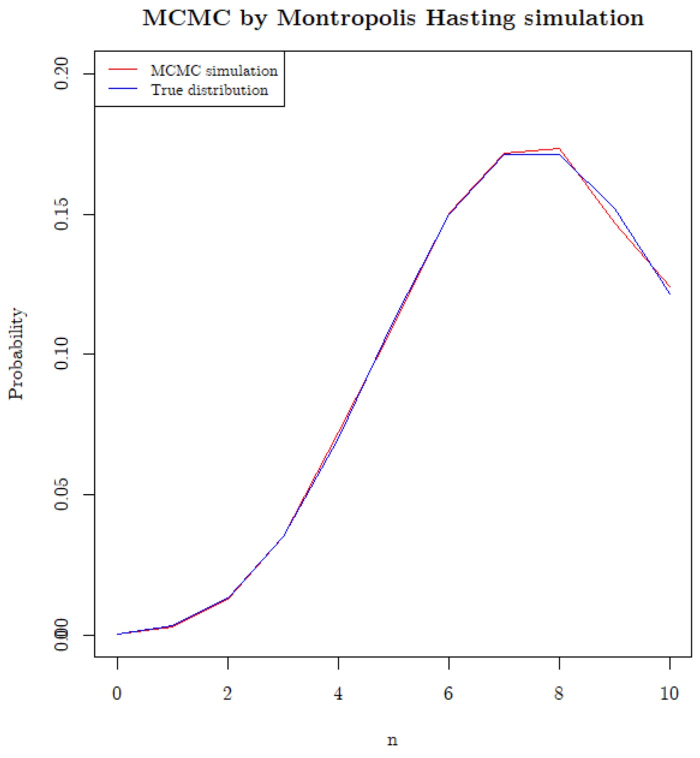


Figure 18: Metropolis Hasting simulation of P(i)

We see that it looks like the simulation fits the distribution quite well but to test it formally we conduct a χ^2-test. As test statistic we get 0.000401 and the theoretical 95% quantile of the $\chi^2(10)$ distribution is 18.307038 and we therefore passes the test with a large margin.

6.2 Metropolis Hasting simulation in two dimensions

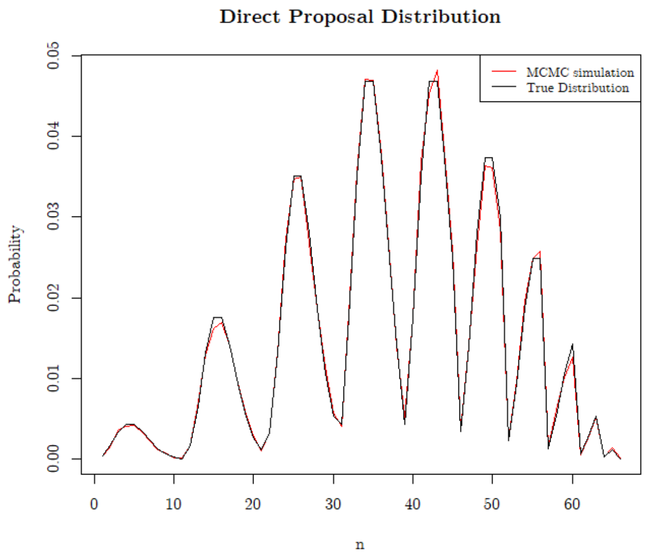
We now introduce a joint posterior distribution that takes two different call types, i and j:

$$P(i,j) = \frac{1}{K}\frac{A_1^i}{i!}\frac{A_2^j}{j!}, \quad 0 \leq i+j \leq n$$

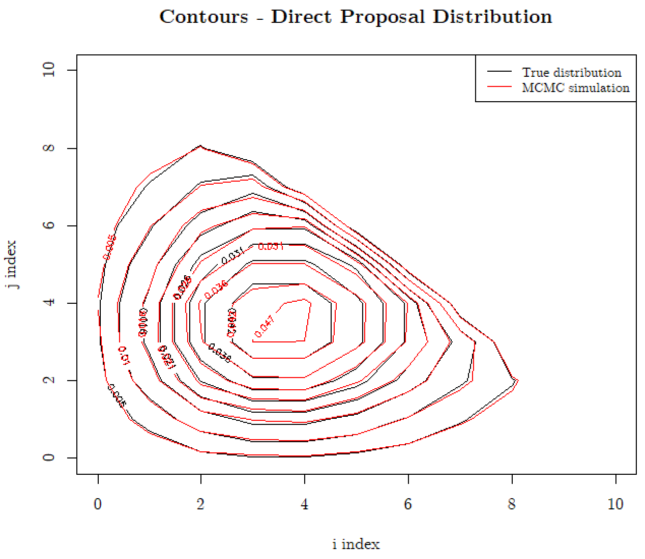
Here K is the normalizing constant, i.e., $K = \sum_{\forall(i,j)(0\leq i+j\leq n)} \frac{A_1^i}{i!}\frac{A_2^j}{j!}$, $i,j \in \mathbb{N}\cup\{0\}$. In our case, we have $A_1 = A_2 = 4$ and $n = 10$ which gives $\frac{(n+2)(n+1)}{2} = 66$ states in our Markov chain. Our proposal distribution is again a uniform but here with 66 points. The arguments for why the chain is still ergodic are the same as in section 6.1. We will again use the Metropolis-Hastings algorithm but approach sampling from the proposal in two different ways. First we will try a direct method where i and j are drawn at once and secondly we will draw them one at a time called the coordinate wise method.

Again we run 100,000 iteration for each proposal

Direct Method With the direct proposal distribution, we obtained the graphical results:



(a) Probabilities in 1D - Direct Proposal



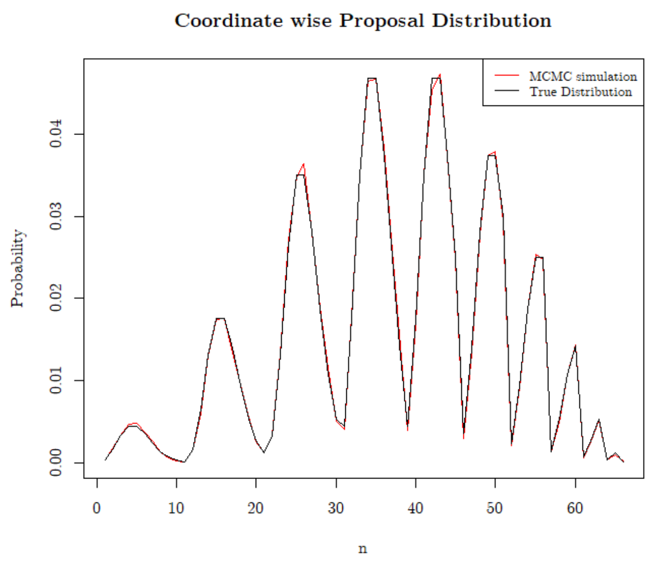
(b) Contours - Direct Proposal

Figure 19: Graphical results w. Direct proposal distribution

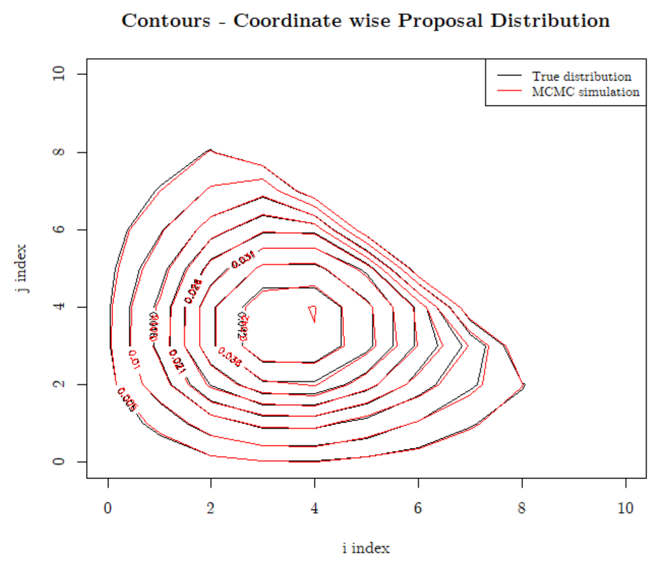
In Figure 19a we see that the simulation is close to the real distribution though the simulations tends to overshoot with areas of high probabilities. Further in Figure 19b we see that the overall picture is good, though the a discrepancy is seen for some indicies in the center of the contours.

The χ^2-test statistics is 0.0018544 and the 95% quantile for $\chi^2(65)$ is 84.820645 hence test is passed with a large margin.

Coordinate-wise method We now use the coordinate wise proposal function and here we obtained the results:



(a) Probabilities in 1D - Coordinate wise



(b) Contours - Coordinate wise Proposal

Figure 20: Graphical results w. Coordinate wise proposal distribution

The left plot in Figure 20 shows that the simulated probabilities seems to fit quite well, though, we again see slight overshoot at indices of high probabilities mass. Likewise, the contour plot seems good with only minor differences.

The obtained χ^2-test statistics is 0.001064 and the 95% quantile for $\chi^2(65)$ is 47.4496 hence the test is passed.

We see that the the coordinate wise proposal function performs slightly better, though, the difference is only marginal at is not seen as a general result.

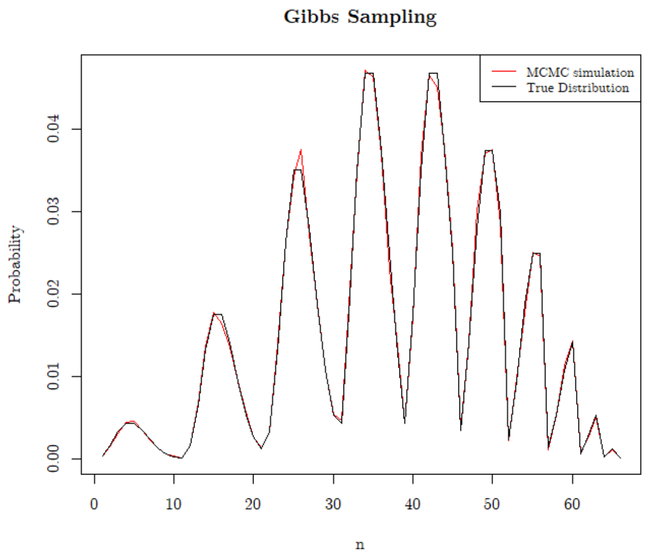
6.3 Gibb's Sampling

The main idea here is to pick an index, $k \in \{1, 2, .., m\}$, randomly and update the state of that single entity based on the state of all the other components from the time step before. Then we go through the rest of the components and update them. Each time we have updated one component, we condition on the new chain based on both the new updated and components we have not yet updated.

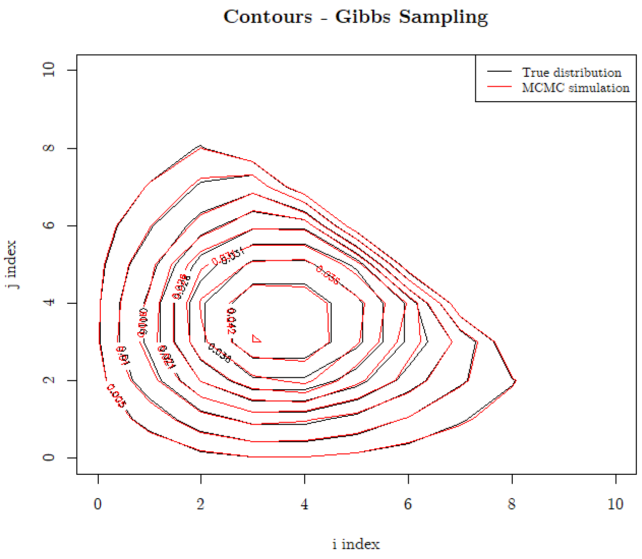
In our case with the state vector x^t at time t, we let x^t_k be the k'th component and x^t_{-k} be the remaining components. Further the proposal distribution is denoted $X \sim h(x^{t+1}|x^t)$.

- choose $k \in \{1, 2\}$ uniformly random
- assign $x^{t+1}_{-k} = x^t_{-k}$
- update $x^{t+1}_k = h(x^{t+1}_k | x^t_{-k})$
- update $x^{t+1}_{k-1} = h(x^{t+1}_{-k} | x^{t+1}_k)$

When we did this we obtained the following result:



(a) Probabilities in 1D - Gibbs



(b) Contours - Gibbs

Figure 21: Graphical results using Gibb's Sampling

In figure 21a, we see that we do not overshoot that often at areas with high probability mass. This is also seen in the right plot where we do not have clear contours for the simulated distribution as in the Metropolis Hasting simulations.

The obtained χ^2-test statistics is 0.001356 and the again 95% quantile for $\chi^2(65)$ is 47.4496 and it passes the test with a large margin.

7 Simulation Annealing

Simulated annealing is a technique that is inspired by the modelling of thermodynamics in cooling metals. Here we want to slowly lower the temperature to make the metal hit a ground state without any defect - the process is called annealing. At a given state, $x \in S$, with the energy $U(x)$ we compute a proposal new state, x_p. In our case we do it as a random walk in a fully connected graph. We accept x_p if $U(x_p) < U(x)$ otherwise we only accept according to a later defined probability function, also called the temperature of the system. We do the latter to lower the probability getting caught in local optima.

The temperature of our system is modelled as:

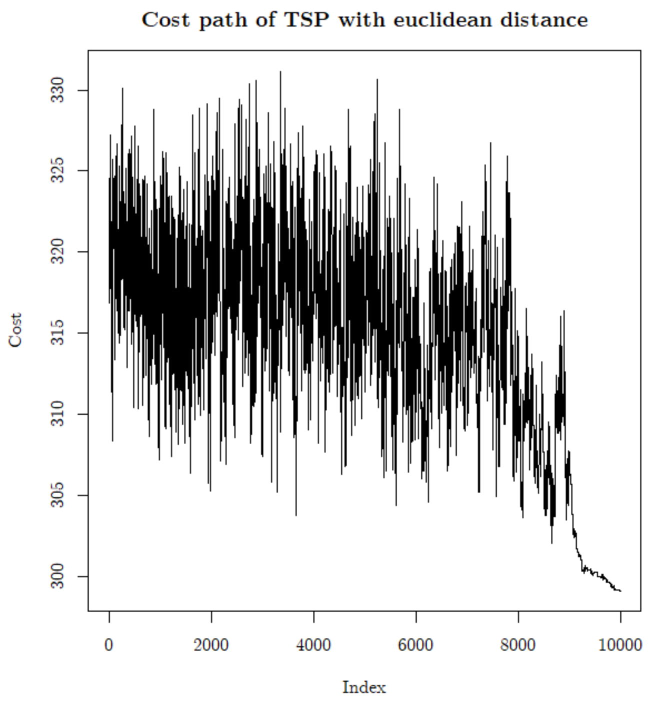
- Generate n equidistant numbers from 1 to c. Assign them to an array "cs".
- Generate an array "scheme" of length equal to the total iterations of the simulation. Assign cs[1] to the first $\frac{iterations}{n}$ slots. Do this for all numbers in cs.
- The temperature is now calculated as $p = i^{\frac{U(x)-U(y)}{scheme[i]}}$, where i is the current iteration of the simulation, U(x) is the cost of the current path, U(y) is the cost of the proposed path and we know $U(y) > U(x)$ if this state of the simulations is entered in iteration i.

In the following we solve two problems formulated as a travelling salesman problem, TSP. Here we are given n cities and a n-by-n matrix, A, where each entry (i, j) denotes the cost for going from i to j. We now want to find the route S that starts and ends at city 1 and has the minimal cost of going through all cities, min $\sum_{i=1}^{n-1} A(S_i,\, S_{i+1})$.

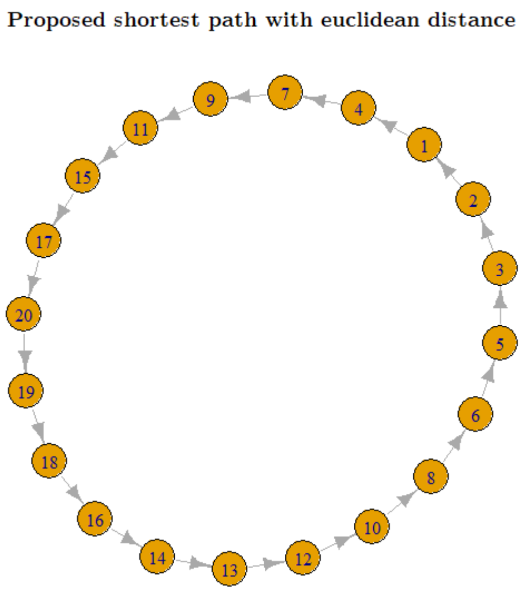
7.1 Simulation Annealing with cost as euclidean distances

Here the cost of going from i to j is defined by the Eucledian distance, e.g., the cost of the entities $A(i, j) = A(j, i) = \sqrt{i^2 + j^2}$.

Here we use the proposed temperature function of accepting a candidate defined above and obtain the following result:



(a) Computed cost of the paths



(b) The found path with minimal cost

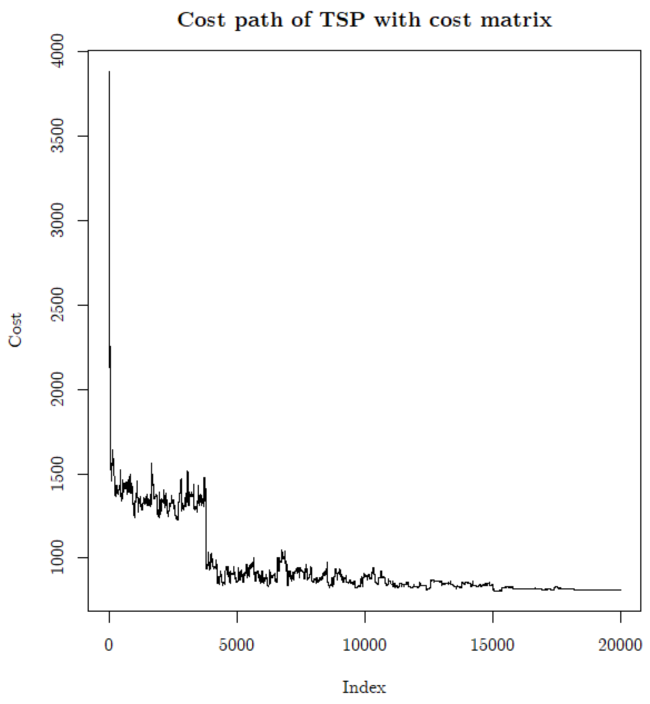
Figure 22: Euclidean cost

In figure 22a we see that computed cost of the path of each iteration. In the beginning we see heavy fluctuations due to a great probability of accepting a state with higher energy, though, in the end it converges towards the a minimum with the path, S^*, seen in the other figure.

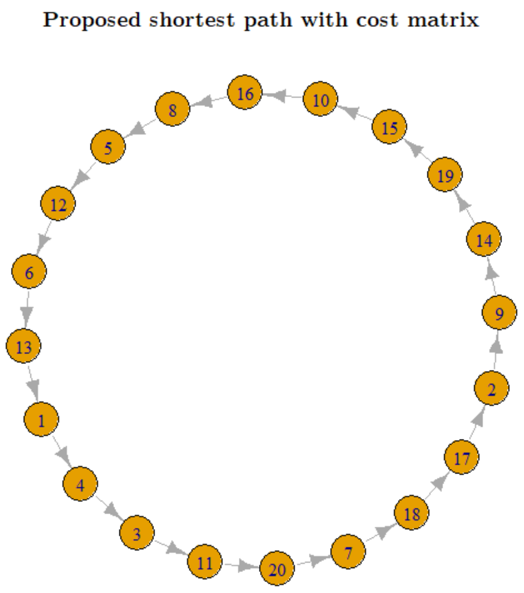
Here one could argue, that the possibility of accepting a state with higher energy is a bit too large because we first see real convergence after around 7000 iterations.

7.2 With given cost matrix

Here we use the same algorithm on a matrix given from [3].



(a) Computed cost of the paths for the cost matrix



(b) The found path with minimal cost

Figure 23: The graphical results for the given cost matrix

We see in figure 23a that initially the cost quickly goes to 1500 and stays here. This is probably a local minimum of an unstable state. We would stay caught here if the probability function did not allow us to shift to a state of greater energy. This happens around iteration 4000. This makes us able to adjust and find a better minimal cost around 800. For iterations above 10,000, we see that the probability of accepting a jump to a state of greater energy is so low that we just stays constant for the last 1000 iterations. In the other figure of 23 we see the computed minimal path.

One could argue that results seen in figure 23 might be a lucky run. Therefore, we simulated the 100 times and saved the cost of the minimal path:

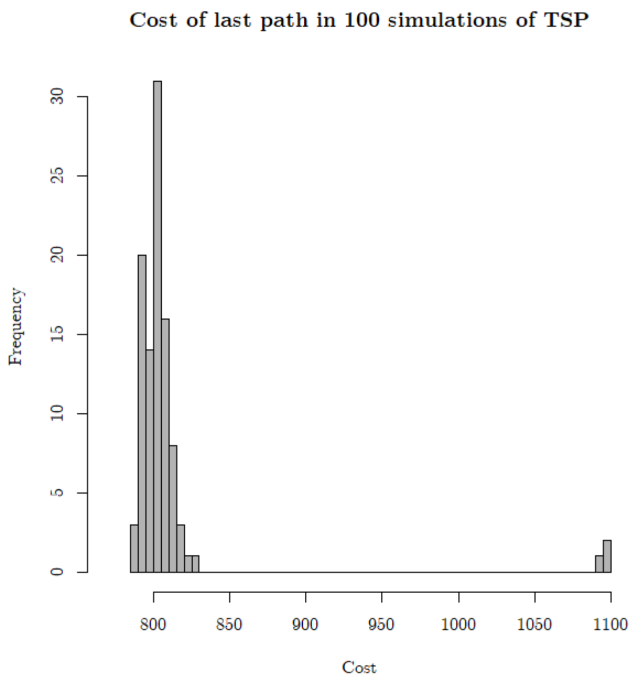


Figure 24: Histogram of the cost of the last path in 100 TSP simulations of 20000 iterations with cost matrix

We see in Figure 24 that in all runs except 3 we get a cost around 800. If we exclude the 3 outliers we have a mean cost of 802.67 and a standard deviation of 7.3933. With the outliers runs we get a mean of 811.53 and a standard deviation of 51.1564.

From the above we can conclude that if one simulate 3 runs of the algorithm one can be almost certain that a very good approximation of the global minimum will have been found.

8 The Bootstrap method

The boostrap method is where we are given a set of n data points from a set $D_n = (x_1, x_2, ..., x_n)$. We then simulate n independent new samples, B_n, by taking n bootstrap samples from the set D_n with replacement.

The probability that x_i is not selected for the boostrap sample B_n is $(1 - \frac{1}{n})^n$ hence

$$\lim_{n\to\infty} (1 - \frac{1}{n})^n = \frac{1}{e} \approx 0.368$$

This shows that we the bootstraped samples will be different each time we mapke a new boostrap sample set.

We will make $r = 100$ boostraped sets, $B_n^1, B_n^2, .., B_n^r$, in the following exercises. We then work with each set as though i.i.d.

8.1 Exercise 13, Chapter 8,[4]

In this exercise we are given a sequence, $X_1, ..., X_n$ of i.i.d. random variables with an unknown mean and two constants, $a < b$, and we want to estimate $p = P\{a < \sum_{i=1}^{n} \frac{X_i}{n} - \mu < b\}$. We can now estimate this probability using bootstrapping applying the following.

1. Make r bootstrapped data sets R_i of size $|X|$ where X is the given data.
2. Make a vector M of all r means of the bootstrapped data sets, $E[R_i]$
3. Make a new vector $Z = M - E[M]$
4. Now $p = \frac{\sum_Z \mathbf{1}(a<z_i<b)}{|Z|}$
5. Repeat 1 to 4 k times so we produce a vector S of k probabilities p_i
6. We now have $P\{a < \sum_{i=1}^{n} \frac{X_i}{n} - \mu < b\} = E[S] \pm u_{\alpha/2}\sqrt{\frac{Var[S]}{|S|}}$

We are given 10 samples $X = \{5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8\}$, $a = -5$, and $b = 5$. With $r = 200$ and $k = 100$ we get

$$P\{-5 < \sum_{i=1}^{10} \frac{X_i}{n} - \mu < 5\} = 0.76695 \pm 0.0062819$$

8.2 Exercise 15, Chapter 8,[4]

In this exercise we are given a sequence $X_1, ..., X_n$ drawn from X. We are to estimate $Var[Var[X]]$. To do this we will follow a similar approach as in section 8.1.

For the sequence $\{5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8\}$ with k=100 and r=200 we get

$$Var[Var[X]] = 57.03714 \pm 1.12989$$

8.3 Examination of a $Pareto(\beta = 1, k = 1.05)$ by use of bootstrapping

In this case we first simulate $N = 200$ Pareto distributed random variables with $\beta = 1$, $k = 1.05$.

With $r = 100$ we now use the bootstrap method to estimate 100 means, $\hat{\mu}$, and find the variance of the means, $Var[\hat{\mu}]$. The same we do for the the median, $\hat{med}$. his is repeated 100 times and we get:

$$Var[\mu] = 0.3662989 \pm 0.0101154$$
$$Var[med] = 0.0161196 \pm 0.00048309$$

We see that the median is a much more stable estimate for the $Pareto(\beta = 1, k = 1.05)$ than the mean. This result is in line with all previous work done with the $Pareto(\beta = 1, k = 1.05)$ in this report. We have seen that for low k the Pareto has some very large outlier values which displace the mean. The median is much more robust so we get a much lower variance as this exercise shows.

9 Appendix

9.1 s174356 - Anton Ruby Larsen

9.1.1 Exercise 1

```
library(schoolmath)
#readline() is used to convert from win source format to R. Use console.
setwd('C:\\Users\\anton\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\Dokumenter\\Uni\\Uni\\6.␣semeste
source("tests␣exercise␣1.R")
library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")

th = theme(text=element_text(family = "Latin␣Modern␣Roman"),plot.title = element_text(hjust = 0.5)


###### 1.1 - Implementation of a LCG #####
LCG = function(simulations = n, a = a, M = M, c = c, x0 = x0){
  out = rep(NA, simulations)
  out[1] = x0
  for (i in 2:(simulations)){
    out[i]= (a*out[i-1]+c) %% M
  }

  return(out/M)
}

# numbers from slide

df = data.frame(x =  LCG(10000, 5, 16, 1, 0))
ggplot(df, aes(x=x)) +
  geom_histogram(bins = 20, fill = "red", color="black",alpha = 0.5)+
  xlim(0,1)+
  labs(title = "LCG␣with␣a=5,␣M=16␣and␣c=1", x="Value", y="Frequency")+
  th
# Normal plot
hist(LCG(10000, 5, 16, 1, 0), breaks = 20, main = "LCG␣with␣a=5,␣M=16␣and␣c=1", xlab = "Value",
     ylab = "Frequency", col = "grey", xlim = c(0,1))


# We check the conditions for a full cycle in
# Thm 1, slide 14, lecture 2.

# Thm 1.1 - c and M should be relative primes
# c = 1 because then it is always true. We then set M to a large number.
c = 1
M = 2^33

# Thm 1.2 - For each prime factor p of M, mod(a, p) = 1
p = unique(prime.factor(M))
p

#because p = 2(because M is 2^x), a mod p = 1 is always true if a is uneven, i.e. 1 mod 2.

a = 1 + 562*2
```

```
# Thm 1.3 - if 4 is a factor of M, then mod(a,4) = 1.
#      Notice, If M is a prime, full period is attained only if a= 1.

# This constrained is solved if a = 1 mod 4, because (a mod uneven) is either 1 or 3 so we just
# skip the 3.

a = 1 + 562*4
a %% 4 == 1
a
# We now have found an a, M and c. We try them with our LCG
df = data.frame(x =  LCG(simulations = 10000, a , M, c, x=0))
ggplot(df, aes(x=x)) +
  geom_histogram(bins = 20, fill = "red", color="black",alpha = 0.5)+
  xlim(0,1)+
  labs(title = "LCG␣with␣a=2249,␣M="~2^33~"␣and␣c=1", x="Value", y="Frequency")+
  th




# We see if a is even the histogram is RIP
hist( LCG(simulations = 10000, a=678 , M, c, x=0), col="red", xlim = c(0,1), breaks = 10)

a
##### 1.2 - Evaluate the quality of the LCG #####
lcg_seq =  LCG(simulations = 10000, a , M, c, x=3)

# Scatterplot
plot(lcg_seq, main =expression(paste(
  "LCG␣with␣a=2249,␣M=2"^"33","␣and␣c=1"
)), xlab="Index", ylab="Value", pch=20, col = "gray5")


# Histograms
hist( lcg_seq, col="grey", xlim = c(0,1), breaks = 20, main =expression(paste(
  "LCG␣with␣a=2249,␣M=2"^"33","␣and␣c=1"
)), xlab="Value", ylab="Frequency" )

# Chi^2 test
chi2.test(lcg_seq,0.95)

# We check if the inbuild version gives the same
chisq.test(table(cut_width(lcg_seq, 0.01, boundary = 0)))

# Kolmogorov-Smirnov test
ks = ks.test(lcg_seq, "punif", 0, 1)
ks

ks$statistic*(sqrt(10000)+0.12+0.11/sqrt(10000))

LCG = lcg_seq
plot(ecdf(LCG))
curve(punif(x, 0, 1), add=TRUE, col="red")
```

```
# Run test 1
above.below(lcg_seq,0.95, plot=FALSE)

# Run test 2
up.down(lcg_seq,0.95)

# Run test 3
the.up.down(lcg_seq,0.95)

# Correlation test
correlation.coef(lcg_seq,0.95,2)


##### 1.3 - Evaluate the quality of the inbuild PRNG #####
inbuild_seq = runif(10000)

# Scatterplot
plot(inbuild_seq)

# Histograms
hist(inbuild_seq, breaks = 20)

# Chi^2 test
chi2.test(inbuild_seq,0.95)

# We check if the inbuild version gives the same
chisq.test(table(cut_width(inbuild_seq, 0.01, boundary = 0)))

# Kolmogorov-Smirnov test
ks.test(inbuild_seq, "punif", 0, 1)

plot(ecdf(inbuild_seq))
curve(punif(x, 0, 1), add=TRUE, col="red")

# Run test 1
above.below(inbuild_seq,0.95, plot=FALSE)

# Run test 2
up.down(inbuild_seq,0.95)

# Run test 3
the.up.down(inbuild_seq,0.95)

# Correlation test
correlation.coef(inbuild_seq,0.95,2)


#################### FUNCTION IN EX1 ######################
library(numbers)
library(schoolmath)
library(ggplot2)
library(BSDA)

chi2.test = function(data = X, quantile = quantile, breaks=100){
  t_value = 0
  bins = table(cut_width(data, 1/breaks, boundary = 0))
  expected = length(data) / length(bins)

  for (i in 1:length(bins)){
    t_value = t_value + (as.numeric(bins[i]) - expected)^2 / expected
  }

  theoretical = qchisq(quantile, df=100-1)
  print(sprintf("The␣theoretical␣%i␣percent␣qunatile:␣%f",quantile*100, theoretical))
  print(sprintf("The␣test␣statistic:␣%f",t_value))

  if(t_value<theoretical){
    print("We␣accept␣the␣null␣hyp.␣and␣therefore␣we␣cannot␣reject␣that␣the␣data␣is␣uniform")
  }else{print("Data␣is␣rejected␣as␣uniform")}
}

above.below = function(data = X, quantile = quantile, plot=FALSE){
  mid = median(data)
  print(mid)
  n1 = sum(data>=mid)
  n2 = sum(data<mid)

  above = 0
  below = 0
  i = 1
  while(i<length(data)+1){
    if(i < length(data)+1 && data[i]<mid){
      while(i < length(data)+1 && data[i]<mid){
        i = i +1
      }
      below = below +1
    }
    else{
      while(i < length(data)+1 && data[i]>mid){
        i = i+1
      }
      above = above +1
    }
  }

  mu = ((2*n1*n2)/(n1+n2))+1;
  sigma2 = (2*n1*n2*(2*n1*n2-n1-n2))/((n1+n2)*(n1+n2)*(n1+n2-1));

  t_val = above+below

  low = qnorm((1-quantile)/2,mu, sqrt(sigma2))
  high = qnorm(quantile+(1-quantile)/2, mu, sqrt(sigma2))

  if(plot){hist((rnorm(length(data), mu, sqrt(sigma2))))}


  print(sprintf("The␣test␣statistic␣is:␣%f",t_val))
  print(sprintf("The␣confidence␣interval␣for␣the␣distribution␣is:␣[%f,%f]",low, high))
  if(t_val>=low && t_val<=high){mes = "cannot␣reject"}else{mes = "reject"}
  print(sprintf("Therefore␣we␣%s␣that␣the␣data␣is␣uniformly␣distributed",mes))
}
```

```
up.down = function(data = X, quantile = quantile){
  count = 1
  bins = rep(0,6)
  for (i in 2:length(data)){
    if(data[i-1]<data[i]){
      count = count +1
    }
    else{
      if (count >6){
        bins[6] = bins[6]+1
      } else{bins[count] = bins[count]+1}
      count = 1
    }
  }

  R = bins
  print(R)
  B = c(1/6, 5/24, 11/120, 19/720, 29/5040, 1/840)

  A = matrix(c(4529.4,  9044.9,  13568,  18091,  22615,  27892,
               9044.9,  18097,   27139,  36187,  45234,  55789,
               13568,   27139,   40721,  54281,  67852,  83685,
               18091,   36187,   54281,  72414,  90470,  111580,
               22615,   45234,   67852,  90470,  113262, 139476,
               27892,   55789,   83685,  111580, 139476, 172860),byrow = T,nrow = 6)

  n = length(data)

  Z = (1/(n-6)*t(R-n*B)%*%A%*%(R-n*B))[1,1]

  theoretical = qchisq(quantile, 6)

  print(sprintf("The␣test␣statistic␣is:␣%f",Z))
  print(sprintf("The␣theoretical␣%i␣percent␣qunatile␣is:␣%f",quantile*100, theoretical))
  if(Z<=theoretical){mes = "cannot␣reject"}else{mes = "reject"}
  print(sprintf("Therefore␣we␣%s␣that␣the␣data␣is␣uniformly␣distributed",mes))

}

the.up.down = function(data = X, quantile = quantile){
  bin = rep(0,length(data))
  n = length(data)

  i=1
  k = 0
  j = 0
  while(i<length(data)){
    while(data[i]<data[i+1] && i<length(data)){
      k = k+1
      i = i+1
    }
    bin[k] = bin[k]+1
    k = 0
    while(data[i]>=data[i+1] && i<length(data)){
      j = j+1
      i = i+1
    }
    bin[j] = bin[j]+1
    j = 0
  }

  X = sum(bin)
  print(X)
  Z = (X-(2*n-1)/3)/sqrt((16*n-29)/90)

  high = qnorm(quantile+(1-quantile)/2)
  low =qnorm((1-quantile)/2)

  print(sprintf("The␣test␣statistic␣is:␣%f",Z))
  print(sprintf("The␣confidence␣interval␣for␣the␣distribution␣is:␣[%f,%f]",low, high))
  if(Z>=low && Z<=high){mes = "cannot␣reject"}else{mes = "reject"}
  print(sprintf("Therefore␣we␣%s␣that␣the␣data␣is␣uniformly␣distributed",mes))
}

correlation.coef = function(data = X, quantile = quantile, step.size = h){
  n = length(data)
  ch = function(h){ lcg_seq[1:(length(lcg_seq)-h)]%*% lcg_seq[(1+h):length(lcg_seq)]/(n-h)}
  high = qnorm(quantile+(1-quantile)/2,0.25,sqrt(7/(14*length(lcg_seq))))
  low  = qnorm((1-quantile)/2,0.25,sqrt(7/(14*length(lcg_seq))))
  test.stat = ch(step.size)

  print(sprintf("The␣test␣statistic␣is:␣%f",test.stat))
  print(sprintf("The␣confidence␣interval␣for␣the␣distribution␣is:␣[%f,%f]",low, high))
  if(test.stat>=low && test.stat<=high){mes = "cannot␣reject"}else{mes = "reject"}
  print(sprintf("Therefore␣we␣%s␣that␣the␣data␣is␣uniformly␣distributed",mes))
}
```

9.1.2 Exercise 2

```
library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")

set.seed(102)

# Exercise 2.1
# Choice a value for p in the geometric distribution and simulate 10,000 outcomes
p=0.2
simulations = runif(10000)
geometric = as.numeric(simulations<=p)

# Found distribution
inv_geo_cdf = function(U, p){floor(log(U)/log(1-p))+1}
geo_est = inv_geo_cdf(simulations, p)


#Theoretical distribution
geo_cdf = function(n,p){1-(1-p)^n}
geo_func = function(n,p){(1-p)^(n-1)*p}
```

```
geo_theo = geo_func(seq(1,max(geo_est), length.out = max(geo_est)),p)
names(geo_theo) = labels(geo_theo)

par(mfrow = c(1,2))
barplot(geo_theo, main = "Theoretical", ylim = c(0,0.23))
barplot(table(geo_est)/10000, main = "Simulated", ylim = c(0,0.23))

# Chi^2 test for distribution
chi2.test = function(obs, exp, quan){
  t_value = 0
  for (i in 1:length(obs)){
    t_value = t_value + (obs[i] - exp[i])^2 / exp[i]
  }
  theoretical = qchisq(quan, df=length(obs)-1)
  print(sprintf("The␣theoretical␣%i␣percent␣qunatile:␣%f",quan*100, theoretical))
  print(sprintf("The␣test␣statistic:␣%f",t_value))

  if(t_value<theoretical){
    print("We␣cannot␣reject␣the␣null␣hyp.")
  }else{print("We␣rejected␣the␣null␣hyp.")}
}

bins = rep(0,max(geo_est))
for(i in 1:max(geo_est)){
  bins[i]=length(which(geo_est==i))
}
bins = bins/length(geo_est)
expected = geo_theo

chi2.test(bins,expected,quan = 0.95)




# Exercise 2.2
# Simulate the 6 point distribution with
p1 = 7/48
p2 = 5/48
p3 = 1/8
p4 = 1/16
p5 = 1/4
p6 = 5/16
probs = c(p1,p2,p3,p4,p5,p6)
names(probs)=labels(probs)

k = 6

#Direct method:
simulation2 = runif(10000)

simD = simulation2

simD[simD<p1] = 1
simD[simD<(p2+p1)] = 2
simD[simD<(p3+p2+p1)] = 3
simD[simD<(p4+p3+p2+p1)] = 4
```

```
simD[simD<(p5+p4+p3+p2+p1)] = 5
simD[simD<(p6+p5+p4+p3+p2+p1)] = 6

#Chi^2 tet
chi2.test(table(simD)/10000, probs,0.95)

# Rejection Method

reject = function(p, c){
  while (TRUE) {
    I = 1 + floor(length(p)*runif(1))
    if (runif(1)<=p[I]/c){return(I)}
  }
}

simR = rep(0,10000)
for (i in 1:10000){
  simR[i] = reject(p = probs, c = 1)
}


chi2.test(table(simR)/10000, probs,0.95)

# Alias tables
Alias_tabels = function(probs){
  L = 1:length(probs)
  F_tab = length(probs)*probs
  G = which(F_tab>=1)
  S = which(F_tab<=1)
  while(!(length(S)==0)){
    i = G[1]
    j = S[1]
    L[j]=i
    F_tab[i]=F_tab[i]-(1-F_tab[j])
    if(F_tab[i]<1-10^(-8)){G = G[-1]; S = append(S, i)}
    S = S[-1]
  }
  return(data.frame(F_tab=F_tab, L = L))
}

D = Alias_tabels(probs)
Alias_probs = D$F_tab
Alias = D$L

Alias_samp = function(F_tab, L_tab){
  I = floor(length(F_tab)*runif(1))+1
  if(runif(1)<=F_tab[I]){return(I)}
  else{return(L_tab[I])}
}

simA = rep(0,10000)
for (i in 1:10000){
  simA[i] = Alias_samp(Alias_probs, Alias)
}
```

```
chi2.test(table(simA)/10000, probs,0.95)

# Histograms for all sampling methods
par(mfrow = c(2,2))
barplot(probs, main = "Theoretical")
barplot(table(simD)/10000, main = "Direct␣sampling")
barplot(table(simR)/10000, main = "Reject␣sampling")
barplot(table(simA)/10000, main = "Alias␣sampling")
```

9.1.3 Exercise 3

```
library(knitr)
library(kableExtra)
library(extrafont)
library(EnvStats)
loadfonts()
par(family = "Latin␣Modern␣Roman")

set.seed(666)

##### Ex. 3.1 #####
# Exponential distribution
r.exp.dist = function(no, lambda){
  rand = c()
  for (i in 1:no){
    rand = c(rand,-log(runif(1))/lambda)
  }
  return(rand)
}
q.exp.dist = function(quan, lambda){
  return(-log(quan)/lambda)
}
par(mfrow = c(1,2))
seq_exp = r.exp.dist(10000, 1.1)
hist(seq_exp,breaks = 101,xlab = "Time", main = expression(paste("Exponential␣sampling,␣", lambda,
hist(q.exp.dist(seq(0,1,length.out = 10000), 1.1),breaks = 101,xlab = "Time", main = expression(pa

ks_exp = ks.test(seq_exp, "pexp", 1.1)

ks_exp$statistic*(sqrt(10000)+0.12+0.11/sqrt(10000))
ks_exp$p.value

plot(ecdf(seq_exp), main = expression(paste("Exponential,␣", lambda, "=1.1")) )
curve(pexp(x, 1.1), add=TRUE, col="red")

# Normal distribution
r.normal.dist = function(no){
  rand = c()
  j = no%%2

  for (i in 1:((no-j)/2)){
    V1 = 2*runif(1)-1
    V2 = 2*runif(1)-1
    R2 = V1^2+V2^2
    while(R2>1){
      V1 = 2*runif(1)-1
      V2 = 2*runif(1)-1
      R2 = V1^2+V2^2
    }
    Z = sqrt(-2*log(runif(1)))*c(V1/sqrt(R2),V2/sqrt(R2))
    rand = c(rand,Z)
  }
  if(j==1){
    V1 = 2*runif(1)-1
    V2 = 2*runif(1)-1
    R2 = V1^2+V2^2
    while(R2>1){
      V1 = 2*runif(1)-1
      V2 = 2*runif(1)-1
      R2 = V1^2+V2^2
    }
    Z = sqrt(-2*log(runif(1)))*c(V1/sqrt(R2),V2/sqrt(R2))
    rand = c(rand,Z[1])
  }
  return(rand)
}
par(mfrow = c(1,2))
seq_norm = r.normal.dist(10000)
hist(seq_norm,breaks = 101,xlab = "", main = "Standard␣normal␣sampling")
hist(qnorm(seq(0,1,length.out = 10000)),xlab = "",breaks = 101, main = "Standard␣normal␣theoretical

ks_norm = ks.test(seq_norm, "pnorm")

ks_norm$statistic*(sqrt(10000)+0.12+0.11/sqrt(10000))
ks_norm$p.value

plot(ecdf(seq_norm), main = "Standard␣Normal")
curve(pnorm(x), add=TRUE, col="red")


# Pareto distribution, X>=0
r.pareto.dist = function(no, beta, k){
  rand = c()
  for (i in 1:no){
    rand = c(rand,beta*(runif(1)^(-1/k)))
  }
  return(rand)
}
q.pareto.dist = function(quan, beta, k){
  return(beta*(quan^(-1/k)))
}
par(mfrow = c(2,2))
set.seed(666)
seq_pareto_205 = r.pareto.dist(10000, 1, 2.05)
hist(seq_pareto_205,breaks = 200,xlab = "", main = expression(paste("Pareto␣sampling,␣", beta, "=1
                                                                     "=2.05")))

#set.seed(666)
seq_pareto_25 = r.pareto.dist(10000, 1, 2.5)
hist(seq_pareto_25,breaks = 200,xlab = "", main = expression(paste("Pareto␣sampling,␣", beta, "=1,␣
                                                                          "=2.5")))


#set.seed(666)
seq_pareto_3 = r.pareto.dist(10000, 1, 3)
hist(seq_pareto_3, breaks = 200,xlab = "", main = expression(paste("Pareto␣sampling,␣", beta, "=1,␣
                                                                          "=3")))



#set.seed(666)
seq_pareto_4 = r.pareto.dist(10000, 1, 4)
hist(seq_pareto_4,breaks = 200,xlab = "", main = expression(paste("Pareto␣sampling,␣", beta, "=1,␣
                                                                         "=4")))



par(mfrow = c(2,2))
hist(q.pareto.dist(seq(0,1,length.out = 10000), 1, 2.05),breaks = 200,xlab = "", main = expression

hist(q.pareto.dist(seq(0,1,length.out = 10000), 1, 2.5),breaks = 200,xlab = "", main = expression(

hist(q.pareto.dist(seq(0,1,length.out = 10000), 1, 3),breaks = 200,xlab = "", main = expression(pa

hist(q.pareto.dist(seq(0,1,length.out = 10000), 1, 4),breaks = 200,xlab = "", main = expression(pa


# Vodka test

# k=2.05
ks_pareto205 = ks.test(seq_pareto_205, "ppareto", 1, 2.05)

ks_pareto205$statistic*(sqrt(10000)+0.12+0.11/sqrt(10000))
ks_pareto205$p.value
plot(ecdf(seq_pareto_205),main = expression(paste("Pareto,␣", beta, "=1,␣", k,
                                                          "=2.05")))
curve(ppareto(x,1,2.05), add=TRUE, col="red")


# k=2.5
ks_pareto25 = ks.test(seq_pareto_25, "ppareto", 1, 2.5)

ks_pareto25$statistic*(sqrt(10000)+0.12+0.11/sqrt(10000))
ks_pareto25$p.value
plot(ecdf(seq_pareto_25),main = expression(paste("Pareto,␣", beta, "=1,␣", k,
                                                         "=2.5")))
curve(ppareto(x,1,2.5), add=TRUE, col="red")

# k=3
ks_pareto3 = ks.test(seq_pareto_3, "ppareto", 1, 3)

ks_pareto3$statistic*(sqrt(10000)+0.12+0.11/sqrt(10000))
ks_pareto3$p.value
plot(ecdf(seq_pareto_3),main = expression(paste("Pareto,␣", beta, "=1,␣", k,
                                                        "=3")))
curve(ppareto(x,1,3), add=TRUE, col="red")
```

```
# k=4
ks_pareto4 = ks.test(seq_pareto_4, "ppareto", 1, 4)

ks_pareto4$statistic*(sqrt(10000)+0.12+0.11/sqrt(10000))
ks_pareto4$p.value
plot(ecdf(seq_pareto_4),main = expression(paste("Pareto,␣", beta, "=1,␣", k,
                                                "=4")))
curve(ppareto(x,1,4), add=TRUE, col="red")

##### Ex 3.2 #####
# Pareto, X>=beta


k = c(2.05,2.5,3,4)
beta = 1
set.seed(666)
simulation1 = r.pareto.dist(20000, beta, k[1])
set.seed(666)
simulation2 = r.pareto.dist(20000, beta, k[2])
set.seed(666)
simulation3 = r.pareto.dist(20000, beta, k[3])
set.seed(666)
simulation4 = r.pareto.dist(20000, beta, k[4])

analytical.mean = k/(k-1)*beta
analytical.variance = k/((k-1)^2*(k-2))*beta^2

simulated.mean = c(mean(simulation1),mean(simulation2),mean(simulation3),mean(simulation4))
simulated.variance = c(var(simulation1),var(simulation2),var(simulation3),var(simulation4))

df = data.frame(SMean = simulated.mean,SVariance = simulated.variance, AMean = analytical.mean, AVa
rownames(df) = c("Beta␣=␣1,␣k␣=␣2.05","Beta␣=␣1,␣k␣=␣2.5","Beta␣=␣1,␣k␣=␣3","Beta␣=␣1,␣k␣=␣4")
colnames(df) = c("Simulated␣Mean", "Simulated␣Variance", "Analytical␣Mean", "Analytical␣Variance")

kable(df,"latex",booktabs =T) %>%
  kable_styling(latex_options ="striped")

vars_own = c()
vars_R = c()

for (i in 1:100){
  vars_own = append(vars_own,var(r.pareto.dist(20000, beta, k[1])))
  vars_R   = append(vars_R, var(rpareto(20000, 1, 2.05)))
}
ana = rep(analytical.variance[1],100)

par(mfrow = c(1,1))

hist(vars_own)
hist(vars_R)
plot(vars_own, xlab = "Simulation", ylab = "Variance",
     main = "100␣Simulations␣of␣Pareto,␣k=2.05")
lines(ana, col = "red")
legend("topright", legend=c("Analytical␣Variance"),
       col=c("red"), lty=1, cex=0.8, inset=.02)
```

```
##### Ex. 3.3 #####

mean.high = c()
mean.low = c()

for (i in 1:100){
  dummy.mean = c()
  for (i in 1:1000){
    sim = r.normal.dist(10)
    dummy.mean = c(dummy.mean ,mean(sim))
  }
  mean.high = c(mean.high, mean(dummy.mean)+1.96*sd(dummy.mean))
  mean.low = c(mean.low, mean(dummy.mean)-1.96*sd(dummy.mean))
}
```

9.1.4 Exercise 4

```
library(R6)
library(distr)
library(liqueueR)
library(knitr)
library(kableExtra)
library(extrafont)
library(EnvStats)
source("C:\\Users\\anton\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\Dokumenter\\Uni\\Uni\\6.␣semes
loadfonts()
par(family = "Latin␣Modern␣Roman")



Queue_sys <- R6Class("Queue_sys", list(

  Arrival_process = NULL,
  Job_process = NULL,
  Arrivals = c(),
  Jobs = c(),

  Arrival_type = "Arrival",
  Job_type = "Job",

  Event_list = PriorityQueue$new(),
  Avalible_units = NULL,

  Blocked = 0,
  Served = 0,
  Tmax = NULL,

  initialize = function(Arrival_process = Pois(1), Job_process = Exp(8),
                        Event_list = PriorityQueue$new(), Avalible_units = 10,
                        Blocked = 0, Served = 0, Tmax = 10) {
    stopifnot(typeof(Arrival_process)=="S4")
    stopifnot(typeof(Job_process)=="S4")
    stopifnot(class(Event_list)[1] == "PriorityQueue")

    stopifnot(as.integer(Avalible_units) == Avalible_units)
    stopifnot(as.integer(Blocked) == Blocked)
    stopifnot(as.integer(Served) == Served)
    stopifnot(as.integer(Tmax) == Tmax)

    self$Arrival_process <- Arrival_process
    self$Job_process <- Job_process

    self$Event_list <- Event_list
    self$Avalible_units <- Avalible_units

    self$Blocked <- Blocked
    self$Served <- Served
    self$Tmax <- Tmax


  },
  decrease_units = function(){
    self$Avalible_units = self$Avalible_units-1
  },
  increase_units = function(){
    self$Avalible_units = self$Avalible_units+1
  },
  increase_blocked = function(){
    self$Blocked = self$Blocked + 1
  },
  increase_served = function(){
    self$Served = self$Served +1
  },
  add_job = function(job){
    self$Jobs = c(self$Jobs, job)
  },
  add_arrival = function(arrival){
    self$Arrivals = c(self$Arrivals, arrival)
  }
))


Event <- R6Class("Event", list(
  Event_type = NULL,
  Time = NULL,

  initialize = function(Event_type, Queue_sys,time_offset){
    stopifnot(class(time_offset)=="numeric")
    stopifnot(class(Queue_sys)[1]=="Queue_sys")
    stopifnot(Event_type==Queue_sys$Arrival_type || Event_type==Queue_sys$Job_type)


    if (Event_type == Queue_sys$Arrival_type){
      time_set = r(Queue_sys$Arrival_process)(1)
      target_time = time_set
      Queue_sys$add_arrival(time_set)
    }
    else if (Event_type == Queue_sys$Job_type){
      time_set = r(Queue_sys$Job_process)(1)
      target_time = time_set
      Queue_sys$add_job(time_set)
    }

    self$Event_type <- Event_type
    self$Time <- time_offset+target_time
  }
))


Create_arrival = function(Queue_sys, event){
  if(Queue_sys$Served+Queue_sys$Blocked>=Queue_sys$Tmax){return()}
  new_event = Event$new("Arrival",Queue_sys, event$Time)
  Queue_sys$Event_list$push(new_event, priority = -new_event$Time)


  if(Queue_sys$Avalible_units>0){
    Job_event = Event$new("Job",Queue_sys, event$Time)
    Queue_sys$Event_list$push(Job_event, priority = -Job_event$Time)
    Queue_sys$decrease_units()
  }else{
    Queue_sys$increase_blocked()
  }


}

Create_Job = function(Queue_sys){
  if(Queue_sys$Served+Queue_sys$Blocked>=Queue_sys$Tmax){return()}
  Queue_sys$increase_units()
  Queue_sys$increase_served()
}

Simulation = function(Queue_sys){
  stopifnot(class(Queue_sys)[1]=="Queue_sys")
  initial_event = Event$new("Arrival", Queue_sys, 0)
  Queue_sys$Event_list$push(initial_event, priority = -initial_event$Time)

  while(Queue_sys$Event_list$size()>0){
    event = Queue_sys$Event_list$pop()

    if(event$Event_type == Queue_sys$Arrival_type){
      Create_arrival(Queue_sys, event)
    }
    else if(event$Event_type == Queue_sys$Job_type){
      Create_Job(Queue_sys)
    }
  }
  return(Queue_sys)
}

conf_int = function(Vec, quant = 0.05){
  mean = mean(Vec)
  sd = sd(Vec)
  q = 1-quant/2
  conf_vec = c(mean-sd/sqrt(length(Vec))*qt(q,length(Vec)-1), mean, mean+sd/sqrt(length(Vec))*qt(q

  return(conf_vec)
}


latex_table = function(que){
  blocks = c()
  block_prob = c()
  index = seq(1,10,length.out = 10)
  for (i in 1:10){
    blocks = c(blocks, que$Queue_sys[[i]]$Blocked)
    block_prob = c(block_prob, que$Queue_sys[[i]]$Blocked/10000)
  }

  df = data.frame(Simulation = index, Blocks =blocks , Block_prob = block_prob)

  colnames(df)[3] = "Blocking␣Probability"

  table = kable(df,"latex",booktabs =T) %>%
    kable_styling(latex_options ="striped")

  return(table)
}

run_processes = function(Ap, Sp, Tmax, sims, units){
  sim_vec = c()
  block_vec = c()
  arrive_vec = c()
  serve_vec = c()
  for (i in 1:sims){
    sim_vec = c(sim_vec, Simulation(Queue_sys$new(Arrival_process = Ap, Job_process = Sp,
                                                  Tmax = Tmax, Avalible_units = units)))
    block_vec = c(block_vec, sim_vec[[i]]$Blocked/(sim_vec[[i]]$Blocked+sim_vec[[i]]$Served))
    arrive_vec= c(arrive_vec, mean(sim_vec[[i]]$Arrivals))
    serve_vec= c(serve_vec, mean(sim_vec[[i]]$Jobs))
    print(i)
  }

  conf_block = conf_int(block_vec, 0.05)

  return(list(Queue_sys = sim_vec,blocks = block_vec, Arrival_mean = arrive_vec, Served_mean = serv
}


pois_exp = run_processes(Ap = Pois(1), Sp = Exp(1/8), Tmax = 10000, units = 10, sims = 100)

pois_exp$conf

conf_int_control(pois_exp, 0.05)

Erlang_exp = run_processes(Ap = Gammad(shape = 1, scale = 1), Sp = Exp(1/8), Tmax = 10000, units =

Erlang_exp$conf
```

```
Hyperexp = function(p1, p2, lam1, lam2){p1*Exp(lam1)+p2*Exp(lam2)}

Hyp_exp = run_processes(Ap = Hyperexp(0.8, 0.2, 0.8333, 5 ), Sp = Exp(1/8), Tmax = 10000, units = 

Hyp_exp$conf

latex_table(Hyp_exp)

Pareto = function(beta, k){ beta*(Unif()^(-1/k))}

Pareto_205_exp = run_processes(Ap = Pareto(1,2.05), Sp = Exp(1/8), Tmax = 10000, units = 10, sims 

Pareto_205_exp$conf

Pareto_105_exp = run_processes(Ap = Pareto(1,1.05), Sp = Exp(1/8), Tmax = 10000, units = 10, sims 

Pareto_105_exp$conf

Chi_exp = run_processes(Ap = Chisq(df = 1), Sp = Exp(1/8), Tmax = 10000, units = 10, sims = 10)

Chi_exp$conf
```

9.1.5 Exercise 5

```
library(magrittr)
setwd("C:\\Users\\anton\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\Dokumenter\\Uni\\Uni\\6.␣semeste
source("exercise␣4.R")
source("C:\\Users\\anton\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\Dokumenter\\Uni\\Uni\\6.␣semest

Conf = function(vec){
  sd_vec = sd(vec)
  mean_vec = mean(vec)
  CI = sd_vec/sqrt(length(vec))*qnorm(0.975)
  low = mean_vec-CI
  high = mean_vec+CI
  print(paste("The␣mean␣is:␣",mean_vec))
  print(paste("The␣standard␣deviation␣is:␣",sd_vec))
  print(paste("The␣95%␣confidence␣interval␣is:␣[",low,";",high,"]"))
  values = list(mean = mean_vec, sd = sd_vec, conf = c(low,high))
  return(values)
}

## Ex5.1 - Crude variable
n = 100
Crude = exp(runif(n))
Crude_values = Conf(Crude)

# Ex5.2 - Antithetic variable
n = 100
Antithetic = runif(n) %>% (function(U){(exp(U)+exp(1)/exp(U))/2})
Antithetic_values = Conf(Antithetic)

# Ex 5.3 - Control variable
n = 100
Control = list(U = runif(n)) %$% {exp(U)-(3/2-exp(1)/2)/(1/12)*(U-1/2)}
```

```
Control_values = Conf(Control)

# Ex 5.4 - Stratified sampling,
n = 100
m = 1000
Stratified = list(U = runif(n), m=m) %$% {out = 0;for (i in 1:m){out = out+exp(U/m+(i-1)/m)/m} ; r
Stratified_values = Conf(Stratified)

vars = rep(0,100)
index = rep(0,100)
for (i in 1:100){
  t = Stratified = list(U = runif(n), m=m*i) %$% {out = 0;for (i in 1:m){out = out+exp(U/m+(i-1)/m)
  vars[i] = var(t)
  index[i] = i*m
}
plot(index, vars, log = "y", ylab = "log␣variance", xlab = "Number␣of␣strats", main = "Effect␣of␣i

# Ex 5.5 - Control variate in queues

pois_exp = run_processes(Ap = Pois(1), Sp = Exp(1/8), Tmax = 10000, units = 10, sims = 100)

A = pois_exp$Arrival_mean
S = pois_exp$Served_mean
B = pois_exp$blocks
control = S-A
c = -cov(B, control)/var(control)

Z = B+c*((control)-7)
(conf_vec_control = conf_int(Z, quant))
(conf_vec = pois_exp$conf)
(improvment = sd(B)-sd(Z))

# Ex 5.6 - Random sampling

rexp_own = function(lambda){-log(Unif())/lambda}
Hyperexp = function(lambdas, probs){
  stopifnot(length(lambdas)==length(probs))

  D = Alias_tabels(probs)
  Alias_probs = D$F_tab
  Alias = D$L
  simA = Alias_samp(Alias_probs, Alias)

  output = rexp_own(lambdas[simA])
  return(output)
}
# test
probs = c(0.8,0.2)
lambdas = c(0.8333, 5)

set.seed(1)
test1 = rhyperexp_own(1000, lambdas, probs)
set.seed(1)
test2 = rhyperexp_own(1000, lambdas, probs)

unique(test1 == test2)
```

```
set.seed(1)
numbers = runif(10)
set.seed(1)
rpois(1,1)
numbers2 = runif(10)
set.seed(1)
rpois(1,1)
numbers3 = runif(10)
set.seed(1)
rpois(1,1)
numbers4 = runif(10)
```

9.1.6 Exercise 6

```
library(R6)
source("C:\\Users\\anton\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\Dokumenter\\Uni\\Uni\\6.␣semest
setwd('C:\\Users\\anton\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\Dokumenter\\Uni\\Uni\\6.␣semeste
source("tests␣exercise␣1.R")

# Ex6.1

# generate uniform 11 point distribution
Point_dist <- R6Class("Queue_sys", list(
  Alias_probs = NULL,
  Alias = NULL,
  probs = NA,
  initialize = function(probs){
    self$probs = probs
    D = Alias_tabels(probs)
    self$Alias_probs = D$F_tab
    self$Alias = D$L
  },
  r = function(curr_point){
    out = (Alias_samp(self$Alias_probs, self$Alias))-1
    return(out)
  },
  q = function(curr_point, future_point){
    return(self$probs[future_point+1])
  }
))

probs = rep(1/11,11)
proposal=Point_dist$new(probs)


# generate posterior
trunc_pois <- R6Class("Queue_sys", list(
  A = NULL,
  initialize = function(lambda, mu, N){
    A = rep(0,N+1)
    for (i in 1:(N+1)){
      A[i] = (lambda/mu)^(i-1)/(factorial(i-1))
    }
    self$A = A/sum(A)
  },
  q = function(j){
    out = self$A[j+1]
    return(out)
  }
))

posterior = trunc_pois$new(1,1/8,10)


# Hasting ratio
Hasting = function(px, py, qx, qy){
  h = log(py)-log(px)+log(qx)-log(qy)
  h = min(1, exp(h))
  return(h)
}

# Montropolis Hasting simulation

MH_MCMC = function(x0, proposal, posterior, iter){

  #Init
  sim_states = rep(NA,iter+1)
  sim_states[1] = x0
  accepted_states = 0

  #start simulation
  for (i in 2:(iter+1)){
    x = sim_states[i-1]
    y = proposal$r(x)

    px = posterior$q(x)
    py = posterior$q(y)

    qx = proposal$q(y,x)
    qy = proposal$q(x,y)
    H = Hasting(px, py, qx, qy)

    u = runif(1)

    if(H>u){
      sim_states[i]=y
      accepted_states = accepted_states + 1
    }else{sim_states[i]=x}
  }
  return_obj = list(Simulated_States = sim_states, accepted_states = accepted_states)
  return(return_obj)
}


sim = MH_MCMC(0, proposal, posterior, 100000)

par(mfrow = c(1,1))
xaxis = seq(0,10,length.out = 11)
plot(xaxis,as.vector(table(sim$Simulated_States)/100000), type = "l", col = "red", main = "MCMC␣by␣
axis(2, seq(0,0.2,10))
lines(xaxis,posterior$A, type = "l", col = "blue")
legend("topleft", legend=c("MCMC␣simulation", "True␣distribution"),
       col=c("red", "blue"), lty=1, cex=0.8)

chi2.test = function(obs, exp, quan){
  t_value = 0
  for (i in 1:length(obs)){
    t_value = t_value + (obs[i] - exp[i])^2 / exp[i]
  }
  theoretical = qchisq(quan, df=length(obs)-1)
  print(sprintf("The␣theoretical␣%i␣percent␣qunatile:␣%f",quan*100, theoretical))
  print(sprintf("The␣test␣statistic:␣%f",t_value))

  if(t_value<theoretical){
    print("We␣cannot␣reject␣the␣null␣hyp.")
  }else{print("We␣rejected␣the␣null␣hyp.")}
}

chi2.test(table(sim$Simulated_States)/100000, posterior$A, 0.95)

# Ex 6.2

N = 10
A1 = 4
A2 = 4
A = matrix(0, N+1, N+1)
for (i in (1:(N+1))){
  for (j in (1:(N+1))){
    if(i+j<13){
      A[i,j]=(A1^(i-1)*A2^(j-1))/(factorial(i-1)*factorial(j-1))
    }
  }
}
K = sum(A)

A = 1/K*A

state = matrix(0,11, 11)
x = 0
y = 0

for (i in 1:100000){
  x = x
  y = y
  x1 = ceiling(runif(1,0,11))
  y1 = ceiling(runif(1,0,11))
  while(x1+y1>12){
    x1 = ceiling(runif(1,0,11))
    y1 = ceiling(runif(1,0,11))
  }

  H = min(1, A[x1,y1] / A[x,y])

  if (runif(1) < H){
    state[x1,y1] = state[x1,y1] + 1
    x = x1
    y = y1
  }
  else{
    state[x,y] = state[x,y] + 1
  }
}
state_prob = state/100000

plot_prob = as.vector(state_prob)[as.vector(state_prob)!=0]
plot_A = as.vector(A)[as.vector(A)!=0]

# normal plot
plot(plot_prob,lty=1, type = "l", col = "red", main = "2D␣MCMC␣by␣Montropolis␣Hasting␣simulation",
lines(plot_A, type = "l", col = "black", lty = 1)
legend("topright", legend=c("MCMC␣simulation", "True␣distribution"),
       col=c("red", "black"), lty=1, cex=0.8)

# Contour plot
levels = round(seq(0,max(A), length.out = 10),3)
xaxis = seq(0,10,length.out = 11)
par(mfrow= c(1,1))
contour(xaxis, xaxis, A, levels = levels, main = "2D␣MCMC␣by␣Montropolis␣Hasting␣simulation", xlab
contour(xaxis, xaxis,state_prob, levels = levels, col = "red", add = T)
legend("topright", legend=c("MCMC␣simulation", "True␣distribution"),
       col=c("red", "black"), lty=1, cex=0.8)


chi2.test(state_prob[state_prob!=0], A[A!=0], 0.95)


# Ex 5.3, MH coordinate wise
state = matrix(0,11, 11)
dims = c(1,2)

x = 0
y = 0
for (i in 2:100000){
  x = x
  y = y
  # 1st coordinate
  x1 = ceiling(runif(1,0,11))
  # 2nd
  y1 = ceiling(runif(1,0,(12-x1)))

  H = min(1, A[x1,y1] / A[x,y])

  if (runif(1) < H){
    state[x1,y1] = state[x1,y1] + 1
    x = x1
    y = y1
  }
  else{
    state[x,y] = state[x,y] + 1
  }
}
```

```
state_prob = state/100000

plot_prob = as.vector(state_prob)[as.vector(state_prob)!=0]

# normal plot
xaxis = seq(1,66, length.out = 66)
plot(xaxis,plot_prob,lty=1, type = "l", col = "red", main = "2D␣MCMC␣by␣Montropolis␣Hasting␣simulat
lines(xaxis,plot_A, type = "l", col = "black", lty = 1)
legend("topright", legend=c("MCMC␣simulation", "True␣distribution"),
       col=c("red", "black"), lty=1, cex=0.8)

# Contour plot
levels = round(seq(0,max(A), length.out = 10),3)
xaxis = seq(0,10,length.out = 11)
par(mfrow= c(1,1))
contour(xaxis, xaxis, A, levels = levels, main = "2D␣MCMC␣by␣Montropolis␣Hasting␣simulation,␣C-wise
contour(xaxis, xaxis,state_prob, levels = levels, col = "red", add = T)
legend("topright", legend=c("MCMC␣simulation", "True␣distribution"),
       col=c("red", "black"), lty=1, cex=0.8)



chi2.test(state_prob[state_prob!=0], A[A!=0], 0.95)

# Gibb's sampling

CondX = function(A){
  condX = matrix(0,nrow(A),nrow(A))

  for (i in 1:length(A)){
    for (j in 1:length(A)){
      if(i+j<13){
        condX[i,j]=A[i,j]/sum(A[i,])
      }
    }
  }
  return(condX)
}

CondY = function(A){
  condY = matrix(0,nrow(A),nrow(A))

  for (i in 1:length(A)){
    for (j in 1:length(A)){
      if(i+j<13){
        condY[i,j]=A[i,j]/sum(A[,j])
      }
    }
  }
  return(condY)
}

SampleX = function(x, condx){
  x = x
  u = runif(1)
  i = 1
  runsum = condx[x,i]
  while(runsum<=u){
    i = i+1
    runsum = runsum+condx[x,i]
  }
  return(i)
}

SampleY = function(y, condy){
  y = y
  u = runif(1)
  i = 1
  runsum = condy[i,y]
  while(runsum<=u){
    i = i+1
    runsum = runsum+condy[i,y]
  }
  return(i)
}

state = matrix(0,11, 11)
dims = c(1,2)

condx = CondX(A)
condy = CondY(A)

y1 = 1
x1 = 1

for (i in 1:100000){
  x1 = SampleY(y1, condy)
  y1 = SampleX(x1, condx)

  state[x1,y1] = state[x1,y1] +1
}

state_prob = state/100000

plot_prob = as.vector(state_prob)[as.vector(state_prob)!=0]

#normal plot
xaxis = seq(1,66, length.out = 66)
plot(xaxis,plot_prob,lty=1, type = "l", col = "red", main = "2D␣MCMC␣by␣Gibb's␣sampling", xlab = "
lines(xaxis,plot_A, type = "l", col = "black", lty = 1)
legend("topright", legend=c("MCMC␣simulation", "True␣distribution"),
       col=c("red", "black"), lty=1, cex=0.8)

# Contour plot
levels = round(seq(0,max(A), length.out = 10),3)
xaxis = seq(0,10,length.out = 11)
par(mfrow= c(1,1))
contour(xaxis, xaxis, A, levels = levels, main = "2D␣MCMC␣by␣Gibb's␣sampling", xlab = "i␣index", y
contour(xaxis, xaxis,state_prob, levels = levels, col = "red", add = T)
legend("topright", legend=c("MCMC␣simulation", "True␣distribution"),
       col=c("red", "black"), lty=1, cex=0.8)
```

```
chi2.test(state_prob[state_prob!=0], A[A!=0], 0.95)
```

9.1.7 Exercise 7

```
library(igraph)
library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")

cost_mat = as.matrix(read.csv("C:\\Users\\anton\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\Dokument

cost_func = function(cost_mat, path , euclid = FALSE){
  if(euclid){
    cost = 0
    previous_state = path[1]
    for (i in 2:length(path)){
      cost = cost + sqrt(previous_state^2+ path[i]^2)
      previous_state = path[i]
    }
    return(cost)
  }
  cost = 0
  previous_state = path[1]
  for (i in 2:length(path)){
    cost = cost + cost_mat[previous_state, path[i]]
    previous_state = path[i]
  }
  cost = cost[[1]]
  return(cost)
}

tsp = function(x, x1, iter, c, scheme, euclid = FALSE){
  c_s = seq(1,c,length.out=scheme)
  c_scheme = c()
  for (i in scheme:1){
    c_scheme = c(c_scheme, rep(c_s[i], iter/scheme))
  }
  costs = rep(0,iter)
  for (i in 1:iter){
    u1 = ceiling(runif(1,1,20))
    u2 = ceiling(runif(1,1,20))
    while(u1 == u2){
      u1 = ceiling(runif(1,1,20))
      u2 = ceiling(runif(1,1,20))
    }

    x1[u1] = x[u2]
    x1[u2] = x[u1]
    cx = cost_func(cost_mat, x, euclid)
    cx1 = cost_func(cost_mat, x1, euclid)

    if (cx > cx1){
      x = x1
    }
    else{
      p = (i)^((cx-cx1)/c_scheme[i])

      r = runif(1)
      if(r <p){
        x = x1
      }else{
        x1 = x
      }

    }
    costs[i] = cost_func(cost_mat, x, euclid)
  }
  output = list(last_cost = min(costs[iter-200:iter]), all_costs = costs, last_path = x)
  return(output)
}

path_matrix = function(path){
  path_matrix = matrix(0, length(path)-1, length(path)-1)
  previous_state = path[1]
  for (i in 2:length(path)){
    path_matrix[previous_state, path[i]] = 1
    previous_state = path[i]
  }
  return(path_matrix)
}

path_vec = function(path){
  path_vec = c()
  previous_state = path[1]
  for (i in 2:length(path)){
    path_vec = c(path_vec, previous_state, path[i])
    previous_state = path[i]
  }
  return(path_vec)
}

# Ex 1
x =sample(seq(2,20,length.out = 19))
x = c(1, x, 1)
x1 = x

sim = tsp(x,x1, 10000, 100, 10, TRUE)
print(sim$last_cost)
print(sim$last_path)
plot(sim$all_costs, type = "l", main = "Cost␣path␣of␣TSP␣with␣euclidean␣distance", xlab = "Index",


g1 = graph(edges = path_vec(sim$last_path), n = 20)
plot(g1, main = "Proposed␣shortest␣path␣with␣euclidean␣distance")

# Ex 2
x =sample(seq(2,20,length.out = 19))
x = c(1, x, 1)
x1 = x
```

```
sim2 = tsp(x,x1, 40000, 220, 10)
print(sim2$last_cost)

t = rep(0,100)

for (i in 1:100){
  t[i] = tsp(x,x1, 20000, 220, 10)$last_cost
  print(t[i])
}

t2 = rep(0,100)

for (i in 1:100){
  t2[i] = tsp(x,x1, 40000, 220, 10)$last_cost
  print(t[i])
}

plot(sim2$all_costs, type = "l", main = "Cost␣path␣of␣TSP␣with␣cost␣matrix", xlab = "Index", ylab


g2 = graph(edges = path_vec(sim2$last_path), n = 20)
plot(g2, main = "Proposed␣shortest␣path␣with␣cost␣matrix")
```

9.1.8 Exercise 8

```
library(EnvStats)

# Ex 8.1(Exercise 13 in Chapter 8 of Ross (P.152)).
data = c(56, 101, 78, 67, 93, 87,64, 72, 80, 69)
# p=P{a<  Xi  /  n    <b}

# bootstrap method
r = 200
boot = matrix(NA, r, 10)
boot_mean = rep(NA,r)
boot_probs = rep(NA,100)

for (k in 1:100){
  for (i in 1:r){
    for (j in 1:10){
      index = ceiling(runif(1,0,10))
      boot[i,j] = data[index]
    }
    boot_mean[i] = mean(boot[i,])
  }
  mu = mean(boot_mean)
  esti = (boot_mean-mu)
  boot_probs[k] = sum(((-5<esti)+(esti<5))==2)/r
}

prob_est = c(mean(boot_probs)-sd(boot_probs)/sqrt(100)*1.96,mean(boot_probs),mean(boot_probs)+sd(b
prob_est

# Ex 8.2(Exercise 15 in Chapter 8 of Ross (P.152)).
data = c(5,4,9,6,21,17,11,20,7,10,21,15,13,16,8)
```

```
r = 200
boot = matrix(NA, r, 15)
boot_var = rep(NA,r)
boot_var_var = rep(NA,100)
for (k in 1:100){
  for (i in 1:r){
    for (j in 1:15){
      index = ceiling(runif(1,0,15))
      boot[i,j] = data[index]
    }
    boot_var[i] = var(boot[i,])
  }
  boot_var_var[k] = var(boot_var)
}

var_var_est = c(mean(boot_var_var)-sd(boot_var_var)/sqrt(100)*1.96,mean(boot_var_var),mean(boot_va
var_var_est

# Ex 8.3
data = rpareto(200, 1, 1.05)
r = 100
k = 100
boot = matrix(NA, r, 200)
boot_median = rep(NA,r)
boot_mean = rep(NA,r)
vars_med = rep(NA,k)
vars_mean = rep(NA,k)
for (l in 1:k){
  for (i in 1:r){
    for (j in 1:200){
      index = ceiling(runif(1,0,200))
      boot[i,j] = data[index]
    }
    boot_median[i] = median(boot[i,])
    boot_mean[i]   = mean(boot[i,])
  }
  vars_med[l]=var(boot_median)
  vars_mean[l]=var(boot_mean)
}


var_median_est = c(mean(vars_med)-sd(vars_med)/sqrt(100)*1.96,mean(vars_med),mean(vars_med)+sd(vars


var_mean_est = c(mean(vars_mean)-sd(vars_mean)/sqrt(100)*1.96,mean(vars_mean),mean(vars_mean)+sd(va

var_median_est
var_mean_est
```

9.2 s184335 - Nicolaj Hans Nielsen

9.2.1 Exercise 1

```
# import relevant libraries
```

```
library('numbers')
library('schoolmath')
library('spgs')
library('xtable')
library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")
loadfonts(device = "win")

# import functions from other sheet:
source("C:\\Users\\Nicolaj\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\DTU␣mapper\\4.␣semester\\Sto


#### LCG ####

# implementation of the LCG

LCG = function(a,M,c,N,x0){

  # intialize data structure

  out = rep(NA,N)
  out[1] = x0

  # compute
  for(i in 2:N){
    out[i] = (a*out[i-1]+c)%%M
  }

  return(out/M)
}


# define the parameters
c = 1
M = 2^(33)

# we choose c=1 because 1 is the only devisible in 1
# hence we know they must be relative prim

# The next contraint we can overcome by utilizing that
# all prime factors of 2^x are 2 verify this by:
primefac = prime.factor(M)

# we then just have to choose an odd a. Furher if x is odd then
# it would not be a factor of 4


# make simulation and initial plot
sim = LCG(562*4+1,M,c,10000,3)

# histogram
hist(sim,col='SteelBlue',breaks=seq(0,1,0.05),xlab=expression(U[i]),main="Distribution␣of␣random␣nu

# simple scatterplot
plot(sim, main="Scatterplot")
```

```
# plot emperical density function
plot(ecdf(sim), main="ecdf␣theory␣and␣simulation")
abline(0,1,col='red')

#test on implementation:

# make q-chi-square for the uniform distribution
chisq.uni(sim,100,0.95 )

chisq.unif.test(sim,bins=100)

# test if the distribution is corect:
ks.test(sim, "punif", min(sim), max(sim))


## run test 1
runI(sim)


# n1 and n2 is inside the confidence interval hence
# we the run test is OK

### Run test II

runII(sim,0.95)

## Run test III

runIII(sim)


## Correlation coefficents

ch(sim,1)



#### Test the test on In-Build Mersenne-Twister PRNG I ####

sim = runif(10000)

hist(sim,col='SteelBlue',breaks=seq(0,1,0.05),xlab=expression(U[i]),main="Distribution␣of␣random␣n

# simple scatterplot
plot(sim, main="Scatterplot")

# plot emperical density function
plot(ecdf(sim), main="ecdf␣theory␣and␣simulation")
abline(0,1,col='red')

# make q-chi-square for the uniform distribution
chisq.uni(sim,100,0.95)

chisq.unif.test(sim,bins=100)
```

```r
# test if the distribution is corect:
ks.test(sim, "punif", min(sim), max(sim))


## run test 1
runI(sim)


# n1 and n2 is inside the confidence interval hence
# we the run test is OK

### Run test II

runII(sim,0.95)

## Run test III

runIII(sim)


## Correlation coefficents

ch(sim,1)


## Sheet with functions for Ex 1 & 2

#### LCG ####

LCG = function(a,M,c,N,x0, Ui=FALSE){

  # intialize data structure

  out = rep(NA,N)
  out[1] = x0

  # compute
  for(i in 2:N){
    out[i] = (a*out[i-1]+c)%%M
  }
  if (Ui==TRUE){
    return(out/M)
  } else{
    return(out)
  }
}

# Compute the chisq statistics for comparison with uniform distribution

chisq.uni = function(x,nclass,quantile){
  if(missing(nclass)){
    nclass=100
  }
  h <- hist(x,nclass,plot = F)
  n.expected = length(x)/nclass
  t.stat = 0
  for(i in 1:nclass){
    t.stat = t.stat + (h$counts[i]-n.expected)^2/n.expected
  }
  quantDist = qchisq(quantile, df=nclass-1)
  print(paste("Test␣statistics:␣", t.stat, "␣␣Here␣the␣", quantile, "%␣quantile␣is␣", quantDist ))
}


# the general chisq function version 1 vere x is simple a vector of random variables

chisq.func = function(x,dist, quantile){
  nclass = length(dist)
  h <- table(x)/length(x)
  t.stat = 0
  for(i in 1:nclass){
    t.stat = t.stat + (h[i]-dist[i])^2/dist[i]
  }
  quantDist = qchisq(quantile, df=nclass-1,lower.tail = F)
  print(paste("Test␣statistics:␣", t.stat, "␣␣Here␣the␣", quantile, "%␣quantile␣is␣", quantDist ))
}


#### Run function I ####
runI = function(sim){
  med = median(sim)
  split = sim>med
  i=1
  run.vec = rep(0,length(sim)-1)
  while(i<=length(sim)){
    run = 0
    while(split[i]==1 && i <= length(sim)){
      run = run + 1
      i = i + 1
    }
    if(run){
      run.vec[run] = run.vec[run] + 1
    }
    run = 0
    while(split[i]==0 && i <= length(sim)){
      run = run + 1
      i = i + 1
    }
    if(run){
      run.vec[run] = run.vec[run] + 1
    }
  }
  n1 = sum(split)
  n2 = length(sim) - n1
  mu = (2*n1*n2)/(n1+n2)+1
  var = (2*n1*n2*(2*n1*n2-n1-n2))/((n1+n2)^2*(n1+n2-1))
  confint = mu + c(-1,1) * qnorm(1-0.05/2)*sqrt(var)


  print(paste("The␣test␣statistics:␣", sum(run.vec), "␣Lower␣CI:␣", confint[1], "Upper␣CI", confin
}
```

```
#### Run function II ####

runII = function(sim,quantile){
  bin = rep(0,6)
  count = 1

  for(i in 2:length(sim)){
    if(sim[i] > sim[i-1]){
      count = count +1
    } else {
      if(count>6){
        count = 6
      }
      bin[count] = bin[count] + 1
      count = 1
    }
  }
  A = matrix(c(4529.4, 9044.9, 13568, 18091, 22615, 27892,
               9044.9, 18097, 27139, 36187, 45234, 55789,
               13568, 27139, 40721, 54281, 67852, 83685,
               18091, 36187, 54281, 72414, 90470, 111580,
               22615, 45234, 67852, 90470, 113262, 139476,
               27892, 55789, 83685, 111580, 139476, 172860),byrow = T, nrow=6)
  b = c(1/6, 5/24, 11/120, 19/720, 29/5040, 1/840)

  test.stat = 1/(length(sim)-6)* t((bin-length(sim)*b)) %*%A%*% (bin-length(sim)*b)
  quantDist = qchisq(quantile, df=6)
  print(paste("Test␣statistics:␣", test.stat, "␣␣Here␣the␣", quantile, "%␣quantile␣is␣", quantDist
}



runIII = function(x){
  n = length(x)
  totalRuns = 0
  # flag = 0 if before is <, 1 if before is >
  flag = 0
  # variable to store the current length
  currentLen = 1
  if(x[1] > x[2]){
    flag =1
  }
  for(i in 3:n){
    if(flag){
      if(x[i-1] >= x[i]){
        currentLen = currentLen + 1
      } else {
        totalRuns = totalRuns + 1
        currentLen = 1
        flag = 0
      }
    } else {
      if(x[i-1] < x[i]){
        currentLen = currentLen + 1
      } else {
        totalRuns = totalRuns + 1
        currentLen = 1
        flag = 1
      }
    }
  }

  # compute the test statistics
  Z = (totalRuns - (2*n-1)/3)/(sqrt((16*n-29)/90))

  confint = Z + c(-1,1) * qnorm(1-0.05/2)

  print(paste("The␣test␣statistics:␣", Z, "␣Quantiles␣[", -qnorm(1-0.05/2),",",qnorm(1-0.05/2), "]"
}


#### test for correlation  ####

ch = function(sim,h){
  n = length(sim)
  tstat = 1/(n-h)*t(sim[-c(1:h)]) %*% sim[-((n-h+1):n)]
  qq = c(qnorm(0.05/2,mean =1/4 ,sd=sqrt(7/(144*n))),qnorm(1-0.05/2,mean =1/4 ,sd=sqrt(7/(144*n)))

  print(paste("The␣test␣statistics:␣", tstat, "␣Quantiles␣[", qq[1],",",qq[2], "]"))
}


#### Kolo Smir for all parameters ####

KolmoSmirAllPar = function(Dn,N){
  test.stat = (sqrt(N)+0.12+0.11/sqrt(N))*Dn
  print(paste("Test␣statistics:␣", test.stat, "␣95%␣confint␣0.9990"))
}
```

9.2.2 Exercise 2

```
# import libraries and fonts

library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")
loadfonts(device = "win")

# import functions:
source("C:\\Users\\Nicolaj\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\DTU␣mapper\\4.␣semester\\Sto


######## Exercise 2 #######


#### Geometric distribution: ####

# Sampling random, independent numbers
```

```
N = 10000
sim = runif(N)

# Initializef(N) p to 0.5
p = 0.2;
geo_dist = floor( log(sim)/log(1-p)+1 )

# visual results

par(mfrow=c(1,2))
barplot(dgeom(0:35,prob=0.2),main="Theoretical␣geometric␣distribution",xlab=expression(x[i]),ylab="
barplot(table(geo_dist)/N,main="Simulated␣geometric␣distribution",xlab=expression(x[i]),ylab="",yl



#### Simulate 6 point distribution ####

# Probabilities
pvec = c(7/48, 5/48, 1/8, 1/16, 1/4, 5/16)


#### Crude, direct method vectorized version ####


crude = sim
for(i in 1:6){
  crude[crude < sum(pvec[1:i])] = i
}


#### rejection method ####

rejectVec = rep(NA,N)
c = 1

for(i in 1:N){
  I = 1 + floor(length(pvec)*runif(1))

  while(!(runif(1) <= pvec[I]/c)) {
    I = 1 + floor(length(pvec)*runif(1))
  }
  rejectVec[i] = I
}


#### Alias method  ####

# genereate aliastables

aliastable = function(p){
  k = length(p)
  L = 1:k
  Ff = k*p
  G = which(Ff >= 1)
  S = which(Ff<=1)
  eps = 10^(-10)
  while(!(length(S)==0)){
    i = G[1]
    j = S[1]
    L[j] = i
    Ff[i] = Ff[i] - (1-Ff[j])
    if(Ff[i]<(1-eps)){
      G  = G[-1]
      S = c(S, i)
    }
    S = S[-1]
  }
  return(data.frame(Ff = Ff, L = L))
}

# Evaluate the function

aliasmethod = function(U,p){
  out = rep(NA,length(U))
  k = length(p)
  FL = aliastable(p)
  for(i in 1:length(out)){
    I = floor(k*U[i])+1
    if(runif(1)<=FL$Ff[I]){
      out[i] = I
    } else {
      out[i] = FL$L[I]
    }
  }
  return(out)
}

alias.vec = aliasmethod(sim,pvec)


# Visual results of the methods used:

par(mfrow=c(2,2))
barplot(pvec, names.arg=1:6, main="Theoretical")
barplot(table(crude)/N,main="Direct␣method", xlab=expression(x[i]))
barplot(table(rejectVec)/N,main="Rejection␣method", xlab=expression(x[i]))
barplot(table(alias.vec)/N,main="Alias␣method", xlab=expression(x[i]))

# other test
chisq.uni(crude,6)
chisq.func(crude,pvec,1-0.05)
chisq.func(rejectVec,pvec,1-0.05)
chisq.func(alias.vec,pvec,1-0.05)

# Kolmogorov Smirnov test
KolmoSmirAllPar(max(pvec-table(crude)/N),6)
KolmoSmirAllPar(max(pvec-table(rejectVec)/N),6)
KolmoSmirAllPar(max(pvec-table(alias.vec)/N),6)
```

9.2.3 Exercise 3

```
#### Exercises 3 ####

###### Exp dist ####

dist.exp = function(N,lambda){
  U = runif(N)
  return(-log(U)/lambda)
}

#### Normal distribution w Box-Muller method ####

cos.sin = function(N){
  cos = rep(NA, N)
  sin = rep(NA, N)
  for(i in 1:N){
    R = 2
    while(R^2 > 1){
      V1 = runif(1,-1,1)
      V2 = runif(1,-1,1)
      R = sqrt(V1^2 + V2^2)
    }
    cos[i] = V1/R
    sin[i] = V2/R
  }
  return(data.frame(cos=cos,sin=sin))
}

dist.normal = function(N){
  NN = ceiling(N/2)
  U1 = runif(NN)
  sqrtu1 = sqrt(-2*log(U1))
  dcossin = cos.sin(NN)
  Z1= sqrt(-2*log(U1)) * dcossin$cos
  Z2= sqrt(-2*log(U1)) * dcossin$sin

  if(N%%2){
    return(c(Z1,Z2[-1]))
  } else {c(Z1,Z2)}
}




#### Pareto distribution ####

# for X >= 0

dist.pareto = function(N,beta,k){
  return(beta*((runif(N))^(-1/k)-1))
}

# for X >= Beta

dist.pareto.beta = function(N,beta,k){
  return(beta*((runif(N))^(-1/k)))
```

```
}

# define parameters
b = 1
k = c(2.05, 2.5, 3, 4)

# initialize data structures to save histrograms and second order moments
histogram = {}
mean.pareto = rep(NA,4)
var.pareto = rep(NA,4)
mean.ana.pareto = rep(NA,4)
var.ana.pareto = rep(NA,4)


# we do not require the distribution to attain 0
for(i in 1:4){
  dist.temp = dist.pareto.beta(100000,b,k[i])
  histogram[[i]] = hist(dist.temp,200)
  mean.pareto[i] = mean(dist.temp)
  var.pareto[i] = var(dist.temp)

  # theoretical. All k are greater than 2 hence:
  mean.ana.pareto[i] = b*k[i]/(k[i]-1)
  var.ana.pareto[i] = b^2*k[i]/((k[i]-1)^2*(k[i]-2))
}

tempdf = data.frame(mean.pareto,mean.ana.pareto,var.pareto,var.ana.pareto)
names(tempdf) = c("Experimental␣mean","Analytical␣mean", "Experimental␣Variance", "Theoratical␣Var
tempdf
xtab = xtable(tempdf,type='latex')



#### Generation of 100 normal distribution ####

mean.norm = rep(NA,100)
mean.confint = matrix(c(rep(NA,100*2)),nrow=2)
var.norm = rep(NA,100)
var.confint = matrix(c(rep(NA,100*2)),nrow=2)

for(i in 1:100){
  temp.dist = dist.normal(10)
  mean.norm[i] = mean(temp.dist)
  var.norm[i] = var(temp.dist)

  mean.confint[,i] = mean.norm[i]+c(-1,1)*pnorm(1-0.05/2)*sqrt(var.norm[i])
  var.confint[1,i] = var.norm[i]*(9)/qchisq(1-0.05/2,9)
  var.confint[2,i] = var.norm[i]*(9)/qchisq(0.05/2,9)
}

# with means from t-dist

for(i in 1:100){
  temp.dist = dist.normal(10)
  mean.norm[i] = mean(temp.dist)
  var.norm[i] = var(temp.dist)
  mean.confint[,i] = mean.norm[i]+c(-1,1)*qt(1-0.05/2,9)*sqrt(var.norm[i]/10)
  var.confint[1,i] = var.norm[i]*(9)/qchisq(1-0.05/2,9)
  var.confint[2,i] = var.norm[i]*(9)/qchisq(0.05/2,9)
}

# plot the means

par(mfrow=c(1,1))
plot(mean.confint[1,],type='l',ylim=c(min(mean.confint),max(mean.confint)*1.3),xlab="Simulation",
abline(h=0,col="red")
lines(mean.confint[2,])
legend("topleft", legend=c("95%␣CI","Theoretical"),
       col=c("black", "red"), lty=1, cex=0.7,
       box.lty=1,ncol=2)


# plot of the variances
par(mfrow=c(1,1))
plot(var.confint[1,],type='l',ylim=c(min(var.confint[1,]),max(var.confint[2,])),xlab="Simulation",
abline(h=1,col="red")
lines(var.confint[2,])
legend( "topleft", legend=c("95%␣CI","Theo"),
        col=c("black", "red"), lty=1, cex=0.8,
        box.lty=1)


sum(mean.confint[1,] < 0 & 0 < mean.confint[2,] )
sum(var.confint[1,] < 1 & 1 < var.confint[2,] )
```

9.2.4 Exercise 4

```
# import libraries
library(R6)
library(distr)
library(xtable)
library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")
loadfonts(device = "win")

#### Exercise 4 ####

# in the following an object orientated implementation is used, hence a no of objects are defined

# an event object - either

# an event structure where

Event <- R6Class("Event",list(
  # A - arriaval, S - service
  type = NULL,
  time = NULL,
  initialize = function(type,world,currentTime){
    # when we initialize an event we want to handle differntly depending on the type:

    if(type == 'A'){
      # sample of the arrival distribution for the world - see below
      duration = r(world$arriveD)(1)
      self$type = 'A'

      # if a service event
    } else if(type== 'S'){
      # sample from the service distribution
      duration = r(world$serviceD)(1)
      self$type = 'S'
    } else{
      print("invalied␣type␣specified")
    }

    # add the current time with the computed duration or arrival time of the event
    self$time = currentTime + duration
  })
)


# implementatin of a priority queue
# made with inspiration from:

# https://www.r-bloggers.com/deriving-a-priority-queue-from-a-plain-vanilla-queue/

PriorityQueueFun <- setRefClass("PriorityQueueFUN",
                                contains = "Queue",
                                fields = list(
                                  priorities = "numeric"
                                ),
                                methods = list(
                                  push = function(item, priority) {
                                    'Inserts␣element␣into␣the␣queue,␣reordering␣according␣to␣prior
                                    callSuper(item)
                                    priorities <<- c(priorities, priority)
                                    #
                                    order = order(priorities, partial = size():1)
                                    #
                                    data <<- data[order]
                                    priorities <<- priorities[order]
                                  },
                                  #
                                  pop = function() {
                                    'Removes␣and␣returns␣head␣of␣queue␣(or␣raises␣error␣if␣queue␣i
                                    if (size() == 0) stop("queue␣is␣empty!")
                                    priorities <<- priorities[-1]
                                    callSuper()
                                  })
)


# make the world object that contains the main properties of the which we model
world = R6Class("World", list(
  arriveD = NULL,
  serviceD = NULL,

  eventList = NULL,
  unitsFree = NULL,

  blocked = NULL,
  served  = NULL,
  limit = NULL,
  initialize= function(){
    self$arriveD <- Pois(lambda=1)
    self$serviceD <- Exp(rate=8)
    self$unitsFree <- 10
    self$eventList <- PriorityQueueFun()
    self$blocked <-0
    self$served <- 0
    self$limit = 10000
  }
))


# compute the functino that handle arrivals when encountered in simulation
arriveFunction = function(world, event){

  # if there are units free to handle the encounter
  if(world$unitsFree > 0){
    serviceEvent = Event$new('S',world,event$time)
    world$eventList$push(serviceEvent, priority=serviceEvent$time)

    world$unitsFree = world$unitsFree - 1

  # if there are no units to handle:
  } else {
    world$blocked = world$blocked + 1
  }
}

# function to handle a service events when encountered
serviceFunction = function(World, event){
  World$unitsFree = World$unitsFree + 1
  World$served = World$served + 1
}


#### Simulation ####

simulation = function(arrivalDist, serviceDist,serviceUnits, limit){
  # initialization
  newWorld = world$new()

  newWorld$arriveD = arrivalDist
  newWorld$serviceD = serviceDist
  newWorld$unitsFree = serviceUnits
  newWorld$limit = limit
  initEvent = Event$new('A',newWorld,0.0)
  newWorld$eventList$push(initEvent,priority = initEvent$time)
  # run the simulation:
  while(newWorld$eventList$size() > 0 && newWorld$served + newWorld$blocked < newWorld$limit){
    # pop the next event in the priority quoeue

    tempEvent = newWorld$eventList$pop()

    # if it is an arrival event

    if(tempEvent$type == 'A'){

      # handle the arrive event
      arriveFunction(newWorld,tempEvent)

      # make the next arrive event
      nextEvent = Event$new('A',newWorld,tempEvent$time)
      newWorld$eventList$push(nextEvent,priority = nextEvent$time)

    # if the poped event is a service event:

    } else if(tempEvent$type == 'S') {

      # handle the service event
      serviceFunction(newWorld, tempEvent)
    } else {
      print("Invalid␣event␣type␣in␣simulation")
    }
  }

  return(newWorld)
}



#### Eksperiments ####

# functions to evaluate performance:

# Blocking probability:

# theoretical
Bform = function(n,lam,s){
  A=lam*s
  denom=1
  for(i in 1:n){
    denom = denom + A^i/(factorial(i))
  }
  return((A^n/factorial(n))/denom)
}

# emperical

Bconfint2 = function(Bvec){
  n = length(Bvec)
  mean = mean(Bvec)
  s2 = var(Bvec)
  CI = c()
  CI[1] = mean + qt(0.05/2,n-1)*sqrt(s2)/sqrt(n)
  CI[2] = mean + qt(1-0.05/2,n-1)*sqrt(s2)/sqrt(n)
  print(paste("Mean",mean,"Lower␣CI", CI[1], "Upper␣CI", CI[2]))
  print(paste("pm", qt(0.05/2,n-1)*sqrt(s2/n),"SD",sqrt(s2)))
}

## simulate with arrival process of poisson with lambda=1 and service process as exponentiel with 

Nsim = 10000


blocked1 = rep(NA,10)
for(i in 1:10){
  temp = simulation(Pois(lambda=1), Exp(rate=1/8), 10, Nsim)
  blocked1[i] = temp$blocked
}
blocked1 = blocked1/Nsim

Bform(10,1,8)
Bconfint2(blocked1)

# Erlang distribution with mean of 1 and hyper exponential distribution of p1=0.8, lam1=0.8333, p2

hyperdist = function(p1,lam1,p2,lam2) {p1*Exp(rate=lam1)+p2*Exp(rate=lam2) }

blocked2 = rep(NA,10)
for(i in 1:10){
  temp = simulation(arrivalDist = Gammad(1,1), serviceDist = hyperdist(0.8,0.8333,0.2,5),serviceUn
  blocked2[i] = temp$blocked
}
blocked2 = blocked2/Nsim

#### Further experiments #####

# Pareto distribution

paretodist = function(beta, k){ beta*(Unif(Min=0,Max=1)^(1/k)) }

blockedPareto = rep(NA,10)
for(i in 1:10){
  temp = simulation(arrivalDist = Pois(lambda=1), serviceDist = paretodist(1,1.05),serviceUnits = 
  blockedPareto[i] = temp$blocked
}
blockedPareto = blockedPareto/Nsim

paretodist = function(beta, k){ beta*(Unif(Min=0,Max=1)^(1/k)) }

blockedPareto2 = rep(NA,10)
for(i in 1:10){
  temp = simulation(arrivalDist = Pois(lambda=1), serviceDist = paretodist(1,2.05),serviceUnits = 
  blockedPareto2[i] = temp$blocked
}
blockedPareto2 = blockedPareto2/Nsim
```

```
# Made partly with inspiration from:
# https://github.com/eugene/02443-Stochastic-simulation/tree/master/delivery1/
```

9.2.5 Exercise 5

```
# import libraries
library(R6)
library(distr)
library(xtable)
library(liqueueR)
library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")
loadfonts(device = "win")


### Exercise 5

# function to compute the confidence interval using t-distribution
diagnose = function(x){
  n = length(x)
  m = mean(x)
  sd = sd(x)

  CI = m + c(-1,1)* sd/sqrt(n)*qt(1-0.05/2,n-1)
  print(paste("Mean:",m, "SD:", sd, "CI:", CI[1],",",CI[2]))
}


# we always make 100 samples:
n=100

#### Monte Carlo evaluation ####

monteCrude = function(N){
  U = runif(N)
  Y = exp(U)
  diagnose(Y)
}

monteCrude(n)

# the theoretical theory:
exp(1)-1


#### antithetic variables ####

antiVar = function(N){
  U = runif(N)
  expU = exp(U)
  Y = (expU+exp(1)/expU)/2
  diagnose(Y)
}

antiVar(n)

# we see that the variance is greatly reduced

#### Control variables ####

# new function to evaluate with reduced variances
CI = function(m,v,n){
  CI = m + c(-1,1)* sqrt(v/n)*qt(1-0.05/2,n-1)
  print(paste("Mean:",m, "SD:", sqrt(v)))
  print(paste("CI:", CI[1],",",CI[2]))
}

# Here we use Ui as the control variable for
# xi=e^Ui

controlVar = function(N){
  Ui = runif(N)
  Xi = exp(Ui)
  c = -cov(Xi,Ui)/var(Ui)
  meanZi = mean(Xi+c*(Ui-0.5))
  varPro = var(Xi)-cov(Xi,Ui)^2/var(Ui)
  print(paste('Mean␣of␣Xi:',mean(Xi)))
  #print(paste("The mean", mean(Xi), "the variance", varPro))
  CI(meanZi,varPro,N)
}

controlVar(100)
# we see that the variance is just reduced sligtly


#### Stratisfied sampling ####

stratisfied = function(Nsample,Ninterval){
  Ui = runif(Nsample)
  W = rep(0,Nsample)
  for(i in 1:Nsample){
    # for each Wi
    for(j in 1:Ninterval){
      W[i] = W[i] + exp((j-1+Ui[i])/Ninterval)
    }
    W[i] = W[i]/Ninterval
  }

  varTotal = var(W)
  CI(mean(W),varTotal,Nsample)
  #print(paste("The mean",mean(W), "the variance:", varTotal))
}

# 100 samples, 10 intervals
stratisfied(100,10)

# 100 samples, 100 intervals
stratisfied(100,100)
```

```
# 10 samples, 100 intervals
stratisfied(10,100)

# we see that the variance is of the same magnitude as 100, 100.
# This tells us that more precise. This is because we use the same ui
# and if we have more intervals but the same random number, we get a lower variance
# than few intervals and many different random numbers.


#### Exercises 5.5 ####

#### define the relevant structures ####

Event <- setClass(
  "Event",
  type = ,
  time = 0

)

Event <- R6Class("Event",list(
  # A - arriaval, S - service
  type = NULL,
  time = NULL,

  initialize = function(type,world,currentTime){
    if(type == 'A'){
      duration = r(world$arriveD)(1)
      self$type = 'A'
      world$arrivalVec =  c(world$arrivalVec,duration)
    } else if(type== 'S'){
      duration = r(world$serviceD)(1)
      self$type = 'S'

      world$serviceVec = c(world$serviceVec, duration)
    } else{
      print("invalied␣type␣specified")
    }
    self$time = currentTime + duration
  })
)



PriorityQueueFun <- setRefClass("PriorityQueueFUN",
                                contains = "Queue",
                                fields = list(
                                  priorities = "numeric"
                                ),
                                methods = list(
                                  push = function(item, priority) {
                                    'Inserts␣element␣into␣the␣queue,␣reordering␣according␣to␣prior
                                    callSuper(item)
                                    priorities <<- c(priorities, priority)
                                    #
                                    order = order(priorities, partial = size():1)
                                    #
                                    data <<- data[order]
                                    priorities <<- priorities[order]
                                  },
                                  #
                                  pop = function() {
                                    'Removes␣and␣returns␣head␣of␣queue␣(or␣raises␣error␣if␣queue␣i
                                    if (size() == 0) stop("queue␣is␣empty!")
                                    priorities <<- priorities[-1]
                                    callSuper()
                                  })
)

# initialize using
#Event$new(type,world,currentTime)

world = R6Class("World", list(
  arriveD = NULL,
  serviceD = NULL,

  eventList = NULL,
  unitsFree = NULL,

  blocked = NULL,
  served  = NULL,
  limit = NULL,

  arrivalVec = NULL,
  serviceVec = NULL,

  initialize= function(){
    self$arriveD <- Pois(lambda=1)
    self$serviceD <- Exp(rate=8)
    self$unitsFree <- 10
    self$eventList <- PriorityQueueFun()
    self$blocked <-0
    self$served <- 0
    self$arrivalVec <- c()
    self$serviceVec <- c()
    self$limit = 10000
  }
))



arriveFunction = function(world, event){

  if(world$unitsFree > 0){
    serviceEvent = Event$new('S',world,event$time)
    world$eventList$push(serviceEvent, priority=serviceEvent$time)

    world$unitsFree = world$unitsFree - 1
  } else {
    world$blocked = world$blocked + 1
  }
```

```
}

serviceFunction = function(World, event){
  World$unitsFree = World$unitsFree + 1
  World$served = World$served + 1
}

simulation = function(arrivalDist, serviceDist,serviceUnits, limit){
  # initialization
  newWorld = world$new()

  newWorld$arriveD = arrivalDist
  newWorld$serviceD = serviceDist
  newWorld$unitsFree = serviceUnits
  newWorld$limit = limit
  initEvent = Event$new('A',newWorld,0.0)
  newWorld$eventList$push(initEvent,priority = initEvent$time)

  # run the simulation:
  while(newWorld$eventList$size() > 0 && newWorld$served + newWorld$blocked < newWorld$limit){

    tempEvent = newWorld$eventList$pop()
    if(tempEvent$type == 'A'){


      # handle the arrive event
      arriveFunction(newWorld,tempEvent)

      # make the next arrive event
      nextEvent = Event$new('A',newWorld,tempEvent$time)
      newWorld$eventList$push(nextEvent,priority = nextEvent$time)
    } else if(tempEvent$type == 'S') {
      serviceFunction(newWorld, tempEvent)
    } else {
      print("Invalid␣event␣type␣in␣simulation")
    }
  }

  return(newWorld)
}



#### Find the control variates ####
# simulate with arrival process of poisson with lambda=1 and service process as exponentiel with ra

# number of simulations
nsim = 100
blocked1 = rep(NA,nsim)
servedMean = rep(NA,nsim)
arrivalMean = rep(NA,nsim)

# simulate 100 times with a limit on 10,000 each time
for(i in 1:nsim){
  print(i)
  temp = simulation(Pois(lambda=1), Exp(rate=1/8), 10, 10000)
  blocked1[i] = temp$blocked
  servedMean[i] = mean(temp$serviceVec)
  arrivalMean[i] = mean(temp$arrivalVec)
}

# calculates the probabilities

blocked1 = blocked1/10000

# plot the candidates
plot(blocked1,servedMean)
plot(blocked1,arrivalMean)
plot(blocked1,servedMean-arrivalMean)

# Correlation
cor(blocked1,servedMean)
cor(blocked1,arrivalMean)
cor(blocked1,servedMean-arrivalMean)


#compute the reduced variance and the c for each candidate

(varBlock1Serve = var(blocked1)- cov(blocked1,servedMean)^2/var(servedMean))
(varBlock1Arrival = var(blocked1)- cov(blocked1,arrivalMean)^2/var(arrivalMean))
(varBoth = var(blocked1) - cov(blocked1,servedMean - arrivalMean)^2/var(servedMean - arrivalMean))
cServed = -cov(blocked1,servedMean)/var(servedMean)
cArrival = -cov(blocked1,arrivalMean)/var(arrivalMean)
cBoth = -cov(blocked1,servedMean-arrivalMean)/var(servedMean-arrivalMean)

# compute the means
mean(blocked1)
mean(blocked1+cServed*(servedMean-8))
mean(blocked1+cArrival*(arrivalMean-1))
mean(blocked1+cBoth*(servedMean-arrivalMean-7))

# function to compute the confidence intervals
BconfintVar = function(Bvec,givenVar){
  n = length(Bvec)
  mean = mean(Bvec)
  CI = c()
  CI[1] = mean + qt(0.05/2,n-1)*sqrt(givenVar)/sqrt(n)
  CI[2] = mean + qt(1-0.05/2,n-1)*sqrt(givenVar)/sqrt(n)
  print(paste("Mean", mean,"Lower␣CI", CI[1], "Upper␣CI", CI[2]))
  print(paste("pm", qt(0.05/2,n-1)*sqrt(givenVar)/sqrt(n),"SD",sqrt(givenVar)))
}

BconfintVar(blocked1+cServed*(servedMean-8),varBlock1Serve)
BconfintVar(blocked1+cArrival*(arrivalMean-1),varBlock1Arrival)
BconfintVar(blocked1+cBoth*(servedMean-arrivalMean-7),varBoth)
```

9.2.6 Exercise 6

```
# import relevant libraries
library(R6)
library(distr)
library(liqueueR)
library('spgs')
library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")
loadfonts(device = "win")


#### Exercise 6 ####

#### chi square ####
chisq.func = function(x,dist, quantile){
  nclass = length(dist)
  t.stat = 0
  for(i in 1:nclass){
    t.stat = t.stat + (x[i]-dist[i])^2/dist[i]
  }
  quantDist = qchisq(quantile, df=nclass-1,lower.tail = F)
  print(paste("Test␣statistics:␣", t.stat, "␣␣Here␣the␣", quantile, "%␣quantile␣is␣", quantDist ))
}


# proposal function uniform distribution in the discrete states

#### Functinos for the alias method ####
aliasmethod = function(U,p){
  out = rep(NA,length(U))
  k = length(p)
  FL = aliastable(p)
  for(i in 1:length(out)){
    I = floor(k*U[i])+1
    if(runif(1)<=FL$Ff[I]){
      out[i] = I
    } else {
      out[i] = FL$L[I]
    }
  }
  return(out)
}

aliastable = function(p){
  k = length(p)
  L = 1:k
  Ff = k*p
  G = which(Ff >= 1)
  S = which(Ff<=1)
  eps = 10^(-10)
  while(!(length(S)==0)){
    i = G[1]
    j = S[1]
    L[j] = i
    Ff[i] = Ff[i] - (1-Ff[j])
    if(Ff[i]<(1-eps)){
      G  = G[-1]
      S = c(S, i)
    }
    S = S[-1]
  }
  return(data.frame(Ff = Ff, L = L))
}


#### point distribution ####
# this object makes a point distribution
# it returns points from 1:number
PointDist <- R6Class("PointDist",list(
  # Number of points
  number = NULL,
  Ff = NULL,
  L = NULL,

  initialize = function(Number){
    self$number = Number
    temp = aliastable(rep(1/self$number, self$number))
    self$Ff = temp$Ff
    self$L = temp$L
  },

  # evaluation function NOTE: the -1
  eval = function(x0){
    I = floor(self$number*runif(1))+1
    if(runif(1) <= self$Ff[I]){
      return(data.frame(cand=I-1,dense=1))
    } else {
      return(data.frame(cand=self$L[I]-1,dense=1))
    }
  })
)


# make object to make the Erlang B-equation
Beq <- R6Class("Beq",list(
  # parameters
  # the matrix
  A = NULL,
  m = NULL,
  # the denominator
  denom = NULL,

  initialize = function(Lam,Mu,M){
    self$A = Lam/Mu
    self$m = M
    tempDenom = 1
    for(jj in 1:(self$m-1)){
      tempDenom = tempDenom + self$A^(jj)/factorial(jj)
    }
    self$denom = tempDenom
  },
  # this part can be called when the structure has been initilized
  eval = function(i){
    return(log(self$A^i/factorial(i)/self$denom))
  })
)


#### Metropolis hastings algorithm ####


MH = function(x0, post, jump, iter){

  chain = rep(NA,iter)
  lnpPobVec = rep(NA, iter)
  naccept = 0

  # initialize
  chain[1] = x0
  lnProbx0 = post$eval(x0)

  # loop
  for(i in 1:iter){
    temp = jump$eval(x0)
    xCand = as.double(temp[1])
    dense = as.double(temp[2])

    # compute Hastings-ratio
    lnProbCand = post$eval(xCand)
    H = exp(lnProbCand-lnProbx0) * dense

    # accept//reject
    if(runif(1)<min(1,H)){
      lnProbx0 = lnProbCand
      x0 = xCand
      naccept = naccept + 1
    }
    chain[i] = x0
  }

  # return chain
  return(list(chain=chain))
}

# initialize
testN = 10000
post = Beq$new(1,1/8,11)
jump = PointDist$new(11)

# test if the proposal function is correct:
testProb = rep(NA,testN)
testCand = rep(NA,testN)
for(i in 1:testN){
  temp = jump$eval(2)
  xCand = temp[1]
  dence = temp[2]
  testProb[i]=exp(post$eval(as.double(xCand)))
  testCand[i] = as.double(xCand)
}
```

```
# should be uniformly distributed
barplot(table(testCand))
# looks fine

# simulate
test2 = as.double(MH(1, post,jump,testN)$chain)


# compute probabilities
table1 = table(test2)/testN


plot(1:11,c(table1[1:11]),xlab='x',ylab="Probability",type='l',col='red')
dist =  exp(post$eval(0:11))
lines(1:12,dist)
legend( "topleft", legend=c("MCMC␣simulation","True␣Distribution"),
        col=c("red","black"), lty=1, cex=0.8,
        box.lty=1)


# compute the chisq
chisq.func(table1,dist=dist,0.95)



#### Exercise 6 continued ####

# compute the A matrix

makeMatrix = function(A1, A2, n){
  A <- matrix(0,nrow=n+1,ncol=n+1)
  for(i in 1:(n+1)){
    for(j in 1:(n+1)){
      if(i-1 + j <= n+1){
        A[i,j] = A1^(i-1)/factorial(i-1)*A2^(j-1)/factorial(j-1)
      }
    }
  }
  return(A/sum(A))
}

# Make an updated version of the Metropolis Hastings algorithm

MH2 = function(x0, post, jump, iter){

  dim = length(x0)
  chain = matrix(NA, nrow=dim,ncol=iter)
  #lnpPobVec = rep(NA, iter)
  #naccept = 0

  # initialize
  chain[,1] = x0
  lnProbx0 = post$eval(x0)

  # loop
  for(i in 1:iter){
    temp = jump$eval(x0,i)
    xCand = as.double(temp[,1])
    dense = as.double(temp[1,2])

    # compute Hastings-ratio
    lnProbCand = post$eval(xCand)
    H = exp(lnProbCand-lnProbx0) * dense

    # accept//reject
    if(runif(1)<min(1,H)){
      lnProbx0 = lnProbCand
      x0 = xCand
      #naccept = naccept + 1
    }
    chain[,i] = x0
  }
  # return chain
  return(data.frame(i=chain[1,],j=chain[2,]))
}


#### point distribution and test of it ####
postDist <- R6Class("p6con",list(
  # parameters
  A1 = NULL,
  A2 = NULL,
  n = NULL,
  matrix = NULL,

  initialize = function(A1,A2,n){
    self$A1 = A1
    self$A2 = A2
    self$n = n
    self$matrix = makeMatrix(self$A1,self$A2,self$n)
  },
  eval = function(ij){
    return(log(self$matrix[ij[1]+1,ij[2]+1]))
  })
)

post2 = postDist$new(4,4,10)
sum(sum(post2$matrix))
post2$eval(c(1,1))

#### coordinate wise point distribution ####

PointDistTwo <- R6Class("PointDist",list(
  # Number of points
  number = NULL,
  Ff = NULL,
  L = NULL,

  initialize = function(Number){
    self$number = Number + 1
    temp = aliastable(rep(1/self$number, self$number))
    self$Ff = temp$Ff
    self$L = temp$L
  },


  eval = function(x0,iter){
    I1 = self$number
    I2 = self$number
    while(I1+I2 >= self$number){
      I1 = floor(self$number*runif(1))
      I2 = floor(self$number*runif(1))
      if(runif(1) > self$Ff[I1+1]){
        I1 = self$L[I1]
      }
      if(runif(1) > self$Ff[I2+1]){
        I2 = self$L[I2]
      }
    }

    return(data.frame(cand=c(I1,I2),dense=1))
  })
)

# test if the coordinate-wise direct distribution is OK

Ntest= 50000
JumpDist = PointDistTwo$new(10)
testI1 = rep(NA, Ntest)
testI2 = rep(NA, Ntest)

for(i in 1:Ntest){
  temp = JumpDist$eval()
  testI1[i] = temp[1,1]
  testI2[i] = temp[2,1]
}


barplot(table(abs(testI1))/Ntest,main="I1")
barplot(table(abs(testI2))/Ntest,main="I2")


#### Test with the direct proposal distribution ####

# initialize the the distribution objects
post = postDist$new(4,4,10)
jump = PointDistTwo$new(10)

# simulate
testN = 100000
testVec2 = MH2(c(0,0),post,jump,testN)

# count occurences
Asim = matrix(0,nrow=11,ncol=11)
for(ii in 1:testN){
  # note: we have the +1 because R is 1 index and we have zeros
  Asim[testVec2$i[ii]+1,testVec2$j[ii]+1] = Asim[testVec2$i[ii]+1,testVec2$j[ii]+1] + 1
}
```

```
# calcualte probability
simProbMatrix = Asim/(testN)

# take only the subset:
subset = post$matrix>0


# 1D plot
par(mfrow=c(1,1))
plot(as.vector(simProbMatrix[subset]),type='l',main="Direct␣Proposal␣Distribution",xlab='n',ylab='
lines(as.vector(post$matrix[subset]),type='l')
legend( "topright", legend=c("MCMC␣simulation","True␣Distribution"),
        col=c("red","black"), lty=1, cex=0.8,
        box.lty=1)

# contour plots
levels = round(seq(0,max(post$matrix), length.out = 10),3)
xaxis = seq(0,10,length.out = 11)
contour(xaxis, xaxis, post$matrix, levels = levels, main = "Contours␣-␣Direct␣Proposal␣Distributio
contour(xaxis, xaxis, simProbMatrix, levels = levels, col = "red", add = T)
legend("topright", legend=c("True␣distribution","MCMC␣simulation"),
       col=c("black","red"), lty=1, cex=0.8 )


#### chisquare function ####
chisq.func = function(obs, theo, quantile){
  nclass = length(theo)
  t.stat = 0
  for(i in 1:nclass){
    t.stat = t.stat + (obs[i]-theo[i])^2/theo[i]
  }
  quantDist = qchisq(quantile, df=nclass-1,lower.tail = F)
  print(paste("Test␣statistics:␣", t.stat, "␣␣Here␣the␣", quantile*100, "%␣quantile␣is␣", quantDist

  if(t.stat < quantDist){
    print("We␣can␣not␣reject␣the␣null␣hypothesis")
  } else {
    print("Null␣hypothesis␣is␣rejected")
  }
}

# compute the chisquare statistics
chisq.func(as.vector(simProbMatrix[subset]),as.vector(post$matrix[subset]),1-0.05)


#### coordinate wise proposal distribution
CoorWise <- R6Class("PointDist",list(
  # Number of points
  number = NULL,
  Ff = NULL,
  L = NULL,

  initialize = function(Number){
    self$number = Number + 1
    temp = aliastable(rep(1/self$number, self$number))
    self$Ff = temp$Ff
    self$L = temp$L
  },


  eval = function(x0,iteration){
    I1 = self$number
    I2 = self$number
    if(iteration%%2){
        I1 = x0[1]
        I2 = floor((self$number-I1)*runif(1))
    } else {
        I2 = x0[2]
        I1 = floor((self$number-I2)*runif(1))
    }

    return(data.frame(cand=c(I1,I2),dense=1))
  })
)

# test if the coordinate-wise proposal distribution is OK
Ntest= 50000
JumpDist = CoorWise$new(10)
testI1 = rep(NA, Ntest)
testI2 = rep(NA, Ntest)
x0 = c(0,0)

for(i in 1:Ntest){
  temp = JumpDist$eval(x0,i)
  x0 = as.double(temp[,1])

  testI1[i] = x0[1]
  testI2[i] = x0[2]

}


barplot(table(abs(testI1))/Ntest,main="I1")
barplot(table(abs(testI2))/Ntest,main="I2")


#### Test of the coordinate wise proposal distribution

# initialize the distributions
post = postDist$new(4,4,10)
jump = CoorWise$new(10)

# simulate the data
testN = 100000
testVec2 = MH2(c(0,0),post,jump,testN)

Asim = matrix(0,nrow=11,ncol=11)

for(ii in 1:testN){
  # note: we have the +1 because R is 1 index and we have zeros
  Asim[testVec2$i[ii]+1,testVec2$j[ii]+1] = Asim[testVec2$i[ii]+1,testVec2$j[ii]+1] + 1
}
```

```
simProbMatrix = Asim/(testN)

# take only the subset:
subset = post$matrix>0


# 1D plot
par(mfrow=c(1,1))
plot(as.vector(simProbMatrix[subset]),type='l',main="Coordinate␣wise␣Proposal␣Distribution",xlab='
lines(as.vector(post$matrix[subset]),type='l')
legend( "topright", legend=c("MCMC␣simulation","True␣Distribution"),
        col=c("red","black"), lty=1, cex=0.8,
        box.lty=1)

# contour plots
levels = round(seq(0,max(post$matrix), length.out = 10),3)
xaxis = seq(0,10,length.out = 11)
contour(xaxis, xaxis, post$matrix, levels = levels, main = "Contours␣-␣Coordinate␣wise␣Proposal␣Di
contour(xaxis, xaxis, simProbMatrix, levels = levels, col = "red", add = T)
legend("topright", legend=c("True␣distribution","MCMC␣simulation"),
       col=c("black","red"), lty=1, cex=0.8 )



# compute the chisquare statistics
chisq.func(as.vector(simProbMatrix[subset]),as.vector(post$matrix[subset]),1-0.05)



#### Simulation with the



##### Gibbs sampling ####


Gibbs <- R6Class("PointDist",list(
  # Number of points
  number = NULL,
  Ff = NULL,
  L = NULL,

  initialize = function(Number){
    self$number = Number + 1
    temp = aliastable(rep(1/self$number, self$number))
    self$Ff = temp$Ff
    self$L = temp$L
  },


  eval = function(x0,iteration){

    if(runif(1)< 0.5){
      I1 = floor(runif(1)*(self$number-x0[2]))
      I2 = floor(runif(1)*(self$number-I1))
    } else {
      I2 = floor(runif(1)*(self$number-x0[1]))
      I1 = floor(runif(1)*(self$number-I2))
    }
    return(data.frame(cand=c(I1,I2),dense=1))
  })
)


# initialize
Ntest= 50000
JumpDist = Gibbs$new(10)
testI1 = rep(NA, Ntest)
testI2 = rep(NA, Ntest)
x0 = c(0,0)

for(i in 1:Ntest){
  temp = JumpDist$eval(x0,i)
  x0 = as.double(temp[,1])

  testI1[i] = x0[1]
  testI2[i] = x0[2]

}

# check that the proposal function is OK
par(mfrow=c(1,2))
barplot(table(abs(testI1))/Ntest,main="I1")
barplot(table(abs(testI2))/Ntest,main="I2")

#### Simulation with Gibbs ####

#### initialize with Gibbs:
post = postDist$new(4,4,10)
jump = Gibbs$new(10)


# simulate

testN = 100000
testVec2 = MH2(c(0,0),post,jump,testN)

# count occurences
Asim = matrix(0,nrow=11,ncol=11)
for(ii in 1:testN){
  # note: we have the +1 because R is 1 index and we have zeros
  Asim[testVec2$i[ii]+1,testVec2$j[ii]+1] = Asim[testVec2$i[ii]+1,testVec2$j[ii]+1] + 1
}
# calcualte probability
simProbMatrix = Asim/(testN)


# take only the subset:
subset = post$matrix>0


# 1D plot
```

```
par(mfrow=c(1,1))
plot(as.vector(simProbMatrix[subset]),type='l',main="Gibbs␣Sampling",xlab='n',ylab='Probability',c
lines(as.vector(post$matrix[subset]),type='l')
legend( "topright", legend=c("MCMC␣simulation","True␣Distribution"),
        col=c("red","black"), lty=1, cex=0.8,
        box.lty=1)

# contour plots
levels = round(seq(0,max(post$matrix), length.out = 10),3)
xaxis = seq(0,10,length.out = 11)
contour(xaxis, xaxis, post$matrix, levels = levels, main = "Contours␣-␣Gibbs␣Sampling", xlab = "i␣
contour(xaxis, xaxis, simProbMatrix, levels = levels, col = "red", add = T)
legend("topright", legend=c("True␣distribution","MCMC␣simulation"),
       col=c("black","red"), lty=1, cex=0.8 )




# compute the chisquare statistics
chisq.func(as.vector(simProbMatrix[subset]),as.vector(post$matrix[subset]),1-0.05)







#### Temp functions that failed ####
PointDistGibbs <- R6Class("PointDist",list(
  # Number of points
  number = NULL,
  Ff = NULL,
  L = NULL,

  initialize = function(Number){
    self$number = Number + 1
    temp = aliastable(rep(1/self$number, self$number))
    self$Ff = temp$Ff
    self$L = temp$L
  },


  eval = function(x0){
    I1 = floor(self$number*runif(1))
    I2 = floor(I1*runif(1))
    return(data.frame(cand=c(I1,I2),dense=1))
  })
)


JumpDist = PointDistGibbs$new(10)
test3 = rep(NA, 10000)

for(i in 1:10000){
  temp = JumpDist$eval()
  test3[i] = temp[2,1] + temp[1,1]
}
```

```
barplot(table(test3))
# this is the problem as this is not at all as before. Therefore, we move to dependens on position



PointDistGibbs <- R6Class("PointDist",list(
  # Number of points
  number = NULL,
  Ff = NULL,
  L = NULL,

  initialize = function(Number){
    self$number = Number + 1
    temp = aliastable(rep(1/self$number, self$number))
    self$Ff = temp$Ff
    self$L = temp$L
  },


  eval = function(x0){

    I1 = x0[1] + sample(c(-1,1),1)
    I2 = x0[2] + sample(c(-1,1),1)
    if(I1 > self$number){
      I1 = self$number
    } else if (I1 < 0){
      I1 = 0
    }
    if(I2 > self$number){
      I2 = self$number
    } else if(I2 <0){
      I2 = 0
    }
    return(data.frame(cand=c(I1,I2),dense=1))
  })
)
```

```
testVec2 = MH2(c(0,0),post,JumpDist,testN)

Asim = matrix(0,nrow=11,ncol=11)

for(ii in 1:testN){
  Asim[testVec2$i[ii]+1,testVec2$j[ii]+1] = Asim[testVec2$i[ii]+1,testVec2$j[ii]+1] + 1
}
simProbMatrix = Asim/(testN)

par(mfrow=c(1,1))
plot(as.vector(simProbMatrix),type='l')
lines(as.vector(post$matrix),type='l',col='red')
legend( "topright", legend=c("Simulated", "True␣Distribution"),
        col=c("black", "red"), lty=1, cex=0.8,
        box.lty=1)

# contour plots
levels = round(seq(0,max(post$matrix), length.out = 10),3)
xaxis = seq(0,10,length.out = 11)
par(mfrow= c(1,1))
contour(xaxis, xaxis, post$matrix, levels = levels, main = "Fat␣Bitches", xlab = "i␣index", ylab =
contour(xaxis, xaxis, simProbMatrix, levels = levels, col = "red", add = T)
legend("topright", legend=c("True␣distribution","MCMC␣simulation"),
       col=c("black","red"), lty=1, cex=0.8 )
```

9.2.7 Exercise 7

```
# import relevant libraries
library(R6)
library(distr)
library(liqueueR)
library('spgs')
library(extrafont)
loadfonts()
par(family = "Latin␣Modern␣Roman")
loadfonts(device = "win")

### Exercise 7 ###

#### 7.1 a ####

# compute the matrix with Eucledian distance as cost:

costEuclid = function(n){
  A = matrix(0, nrow=n,ncol=n)
  for(i in 1:n){
    for(j in 1:n){
      if(i!=j){
        A[i,j] = sqrt(i^2+j^2)
      }
    }
  }
  return(A)
}

# compute the cost of a given path
pathCostFun = function(path,cost){
  p = 0
  for(i in 2:length(path)){
    p = p + cost[path[i-1],path[i]]
  }
  return(p)
}

# function to come up with a new path
newCandFun = function(path){
  # expects the input to be of len n+1
  I1 = floor(runif(1,1,length(path)))
  I2 = floor(runif(1,1,length(path)))
  temp = path[I1]
  path[I1] = path[I2]
  path[I2] = temp
  path[length(path)] = path[1]
  return(path)
}

# simulated anealing with the function
simAneal = function(x0, costMatrix,pathCost, jump, iter){
  dim = length(x0)
  cost = rep(NA,iter)
  #chain = matrix(NA, nrow=dim,ncol=iter)
  #chain[,1] = x0
  costx0 = pathCost(x0,costMatrix)

  # loop
  for(i in 1:iter){
    newPath = jump(x0)
    costNewPath = pathCost(c(newPath),costMatrix)
    #Tfun = 1/sqrt(1+i)
    #Tfun = -log(i+1)
    # use a slightly different T
    Tfun = 1/(1/(iter/1000)+(i/iter))

    # accept//reject
    if(costx0 > costNewPath ||
       runif(1)<exp(-(costNewPath-costx0)/(Tfun))){
      x0 = newPath
      costx0 = costNewPath
    }
    cost[i] = costx0
  }
  # return chain
  out = list(cost=cost,x0=x0)
  return(out)
}

#### simulation with Eucledian distance ####
# initialize
ncity = 20
x0 = c(1,2:ncity,1)
costMatrixEU = costEuclid(ncity)
iterations = 50000
outEU = simAneal(x0=x0,costMatrix=costMatrixEU, pathCost=pathCostFun,jump=newCandFun,iter=iteratio
plot(outEU$cost,xlab='iteration',ylab='cost')
```

```
#### Test with library function ####
library(TSP)
atsp <- ATSP(as.matrix(costMatrixEU))
(tour <- solve_TSP(atsp))
image(atsp,tour)
cities = as.integer(tour)


#### 7.1 b ####

# import the relevant matrix
costTSP = as.matrix(read.table("C:\\Users\\Nicolaj\\OneDrive␣-␣Danmarks␣Tekniske␣Universitet\\DTU␣


# initialize TSP:
ncity = dim(costTSP)[1]
x0 = c(1,2:ncity,1)
iterations = 50000
out = simAneal(x0=x0,costMatrix=costTSP, pathCost=pathCostFun,jump=newCandFun,iter=iterations)
plot(out$cost,type='l')

# test how many times it gets caught
caught = rep(NA,100)
for(i in 1:100){
  print(i)
  out = simAneal(x0=x0,costMatrix=costTSP, pathCost=pathCostFun,jump=newCandFun,iter=iterations)
  caught[i] = out$cost[iterations]
}

mean(caught)
sd(caught)

hist(caught)
```

9.2.8 Exercise 8

```
# Exercise 8


#### Exercise 13 ####

xi = c(56,101,78,67,93,87,64,72,80,69)

meanXi = mean(xi)
hits = 0
r = 1000

for(i in 1:r){
  bootMean = mean(sample(xi,10,replace = T))
  if(-5 < bootMean - meanXi && bootMean - meanXi < 5){
    hits = hits + 1
  }
}
```

```
(prob = hits/r)


#### Exercise 15 ####

xi15  = c(5, 4, 9, 6, 21, 17, 11, 20, 7, 10, 21, 15, 13, 16, 8)

r = 1000
varInnerRun = rep(NA,r)

Nrun = 100
varOuterRuns = rep(NA,Nrun)

for(j in 1:Nrun){
  for(i in 1:r){
    varInnerRun[i] = var(sample(xi15,15,replace = T))
  }
  varOuterRuns[j] = var(varInnerRun)
}

mean(varOuterRuns)+ c(-1,1)*qnorm(1-0.05/2)*sd(varOuterRuns)/sqrt(Nrun)


#### Exercise 8 ####

# Pareto distribution

dist.pareto = function(N,beta,k){
  return(beta*((runif(N))^(-1/k)-1))
}

dist.pareto.beta = function(N,beta,k){
  return(beta*((runif(N))^(-1/k)))
}

N = 200
beta = 1
k = 1.05
XiPareto = dist.pareto.beta(N,beta,k)

boots = function(Xi){
  meanSample = mean(Xi)
  medSample = median(Xi)
  n = length(Xi)
  r=100

  meanVec = rep(NA,r)
  medVec = rep(NA,r)
  for(i in 1:100){
    temp = sample(Xi,200,replace = T)
    meanVec[i] = mean(temp)
    medVec[i] = median(temp)
  }
  out = list(mean=meanSample, varMean=var(meanVec),med=medSample, varMed = var(medVec))
  return(out)
}

boots(XiPareto)

# we see the mean is much more unstable
```

10 References

References

[1] Justin Ellis. *A Practical Guide to MCMC Part 1: MCMC Basics, accessed 14/6-2020*. `https://jellis18.github.io/post/2018-01-02-mcmc-part1/?fbclid=IwAR3z2cqxUVhI2Bib_oCS-aVRNBItkgLoVMGdoJsR8J51ovCmrMoA8Ol`

[2] Martin Haugh. *Leture notes for IEOR E4703, Monte-Carlo Simulation, Spring 2017, IEOR Department, Columbia University*. `https://martin-haugh.github.io/teaching/monte-carlo/`.

[3] Bo Friis Nielsen. *Course page for 02443, Stochastic Simulation, June 2020*. `http://www2.imm.dtu.dk/courses/02443/`.

[4] Sheldon Ross. "Chapter 8 - Statistical Analysis of Simulated Data". In: *Simulation (Fifth Edition)*. Ed. by Sheldon Ross. Fifth Edition. Academic Press, 2013, pp. 135–152. ISBN: 978-0-12-415825-2. DOI: `https://doi.org/10.1016/B978-0-12-415825-2.00008-5`. URL: `http://www.sciencedirect.com/science/article/pii/B9780124158252000085`.