Computational Statistics - Lab 04

Annalena Erhard (anner218) and Maximilian Pfundstein (maxpf364)

2019-02-05

Contents

1 Question 1: Computations with Metropolis-Hastings

Consider the following probability density function:

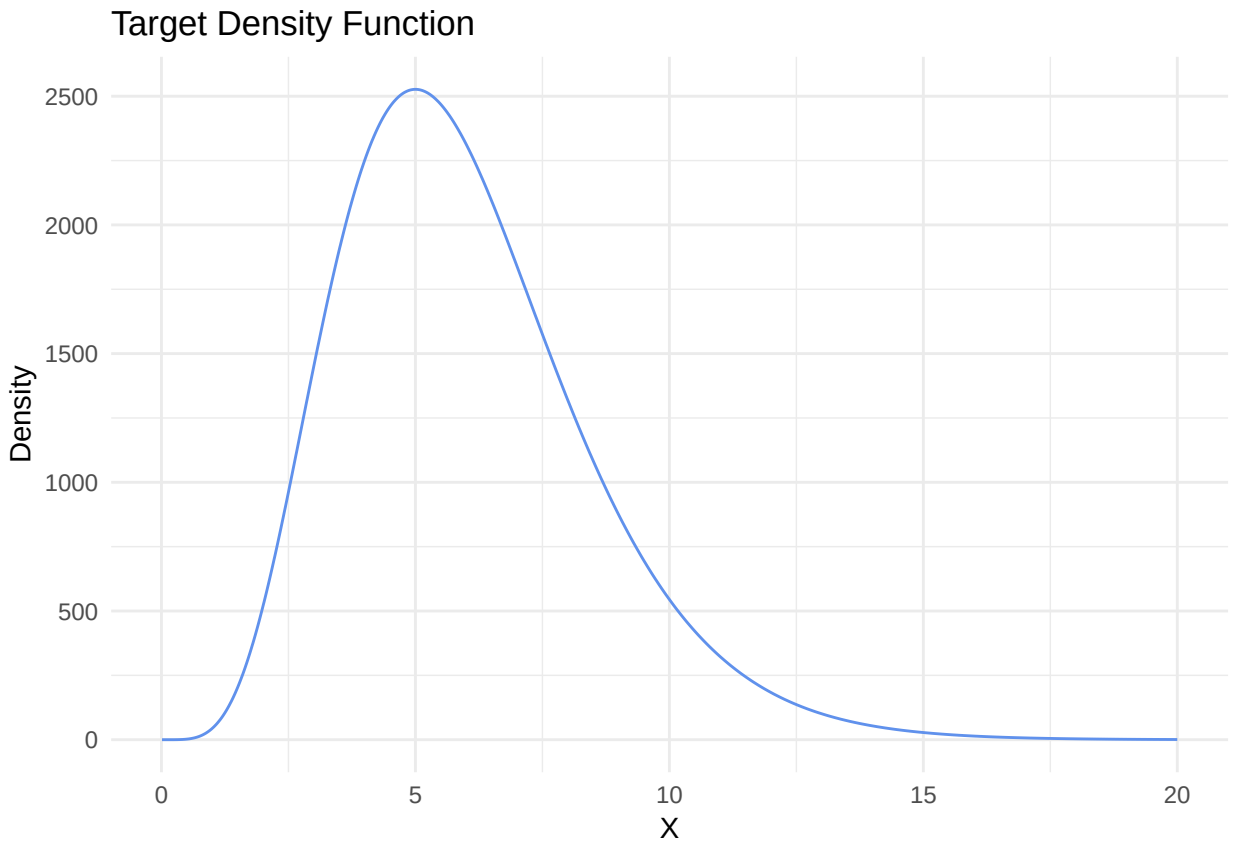
$$f(x) \propto x^5 e^{-x}, \quad x > 0.$$

You can see that the distribution is known up to some constant of proportionality. If you are interested (**NOT** part of the Lab) this constant can be found by applying integration by parts multiple times and equals 120.

1. Use Metropolis–Hastings algorithm to generate samples from this distribution by using proposal distribution as log–normal $LN(X_t, 1)$, take some starting point. Plot the chain you obtained as a time series plot. What can you guess about the convergence of the chain? If there is a burn–in period, what can be the size of this period?

Answer: First we will define the original function with the constant set to get a brief overview how the function actually looks like. We also define the given function without the constant, as we're going to us it later as the target distribution. We use the function without the constant to show that Metropilis-Hastings works without knowing the constant. As random samplings can actually give an x outside of the scope of the function, we will set all values of $x \leq 0$ to a relatively small value.

```
# Target function with original scaling
f = function(x) {
  return(120 * x^5 * exp(-x))
}

# Target function
df = function(x) {
  x = ifelse(x <= 0, 0.000001, x)
  return(x^5 * exp(-x))
}
```

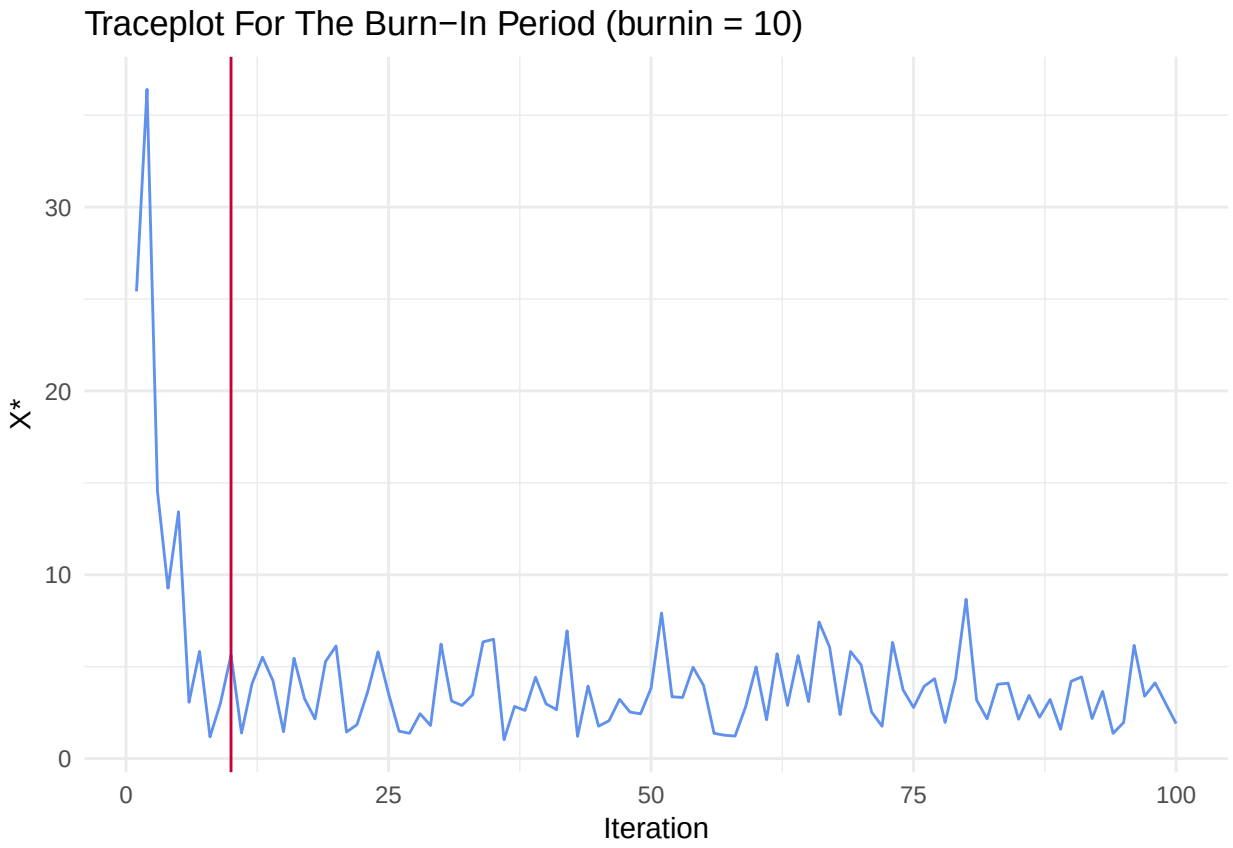


The following code chunk defines the Metropolis-Hastings algorithm. The correction factor `c` is used to handle the asymmetry in the proposal function. The code is commented.

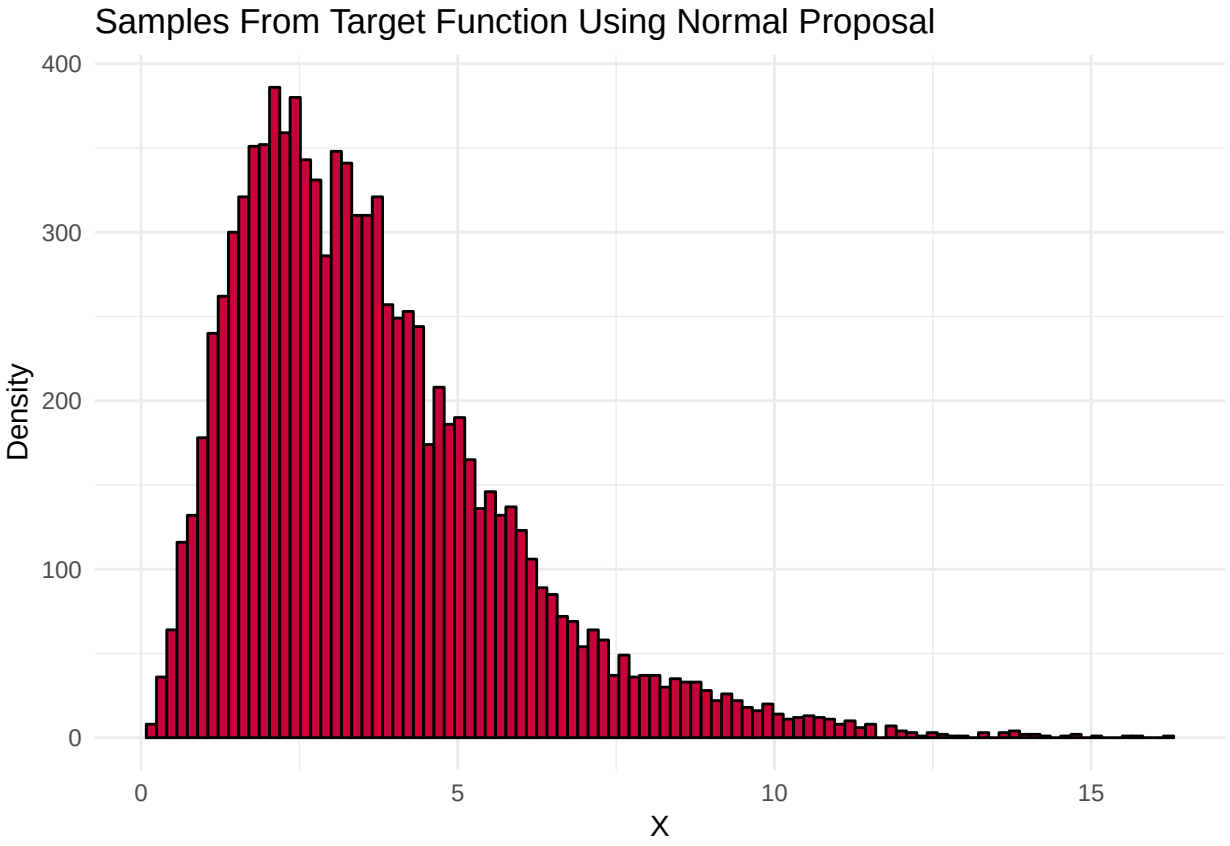
```
#' Metropolis Hasting Algorithm
#'
#' @param n Number of samples from the target distribution.
#' @param x_0 Initial state from Pi.
#' @param b Burn-in steps to remove from samples.
#' @param proposal Selects the proposal function.
#' @param keep_burnin Decides if to keep the samples during the burn-in period.
#'
#' @return Returns a list containing samples.
#' @export
#'
#' @examples
metropolis_hastings = function(n = 1, x_0 = 1, b = 50, proposal = "lnorm",
                               keep_burnin = FALSE) {

  # Vectors to store the samples in
  samples = c()

  # Samples from proposel from the random walk (MC)
  xt = x_0
  xt_1 = x_0

  while (length(samples) < n) {

    # Generate proposal state
```

```
    if (proposal == "lnorm") {
      x_star = rlnorm(n = 1, meanlog = log(xt), sdlog = 1)

      # Calculate correction factor C
      c = dlnorm(xt_1, meanlog = xt, sdlog = 1) /
      dlnorm(x_star, meanlog = xt, sdlog = 1)
    }
    else if (proposal == "chisquared") {
      x_star = rchisq(n = 1, df = floor(xt))

       # Calculate correction factor C
      c = dchisq(x = xt_1, df = floor(xt)) /
      dchisq(x_star, df = floor(xt))
    }
    else {
      stop("Invalid proposal.")
    }

    # Calculate acceptance probability alpha
    if (df(xt_1) <= 0) {
      # We need this to avoid troublesome areas where the density is so low that
      # it's 0 from a computiationally point of view.
      alpha = 0
    }
    else {
      alpha = min(1, df(x_star)/df(xt_1) * c)
    }

    # Generate u from uniform
    u = runif(n = 1, min = 0, max = 1)

    # Decide if to accept or reject the proposal
    if (u <= alpha) {
      # Accept
      xt_1 = xt
      xt = x_star
      samples = c(samples, x_star)
    }
    else {
      # Reject
      xt = xt_1
    }
  }

  # Return samples
  if (keep_burnin) return(samples)
  return(samples[b+1:length(samples)])
}
```

For the lognormal distribution as the proposal we selected a starting point which is far from the converging value to illustrate the burn-in period. We can see that we reach convergence rather quickly after around 8 iterations. We increased the limit to 10 to be sure to get into the area of the converging value.

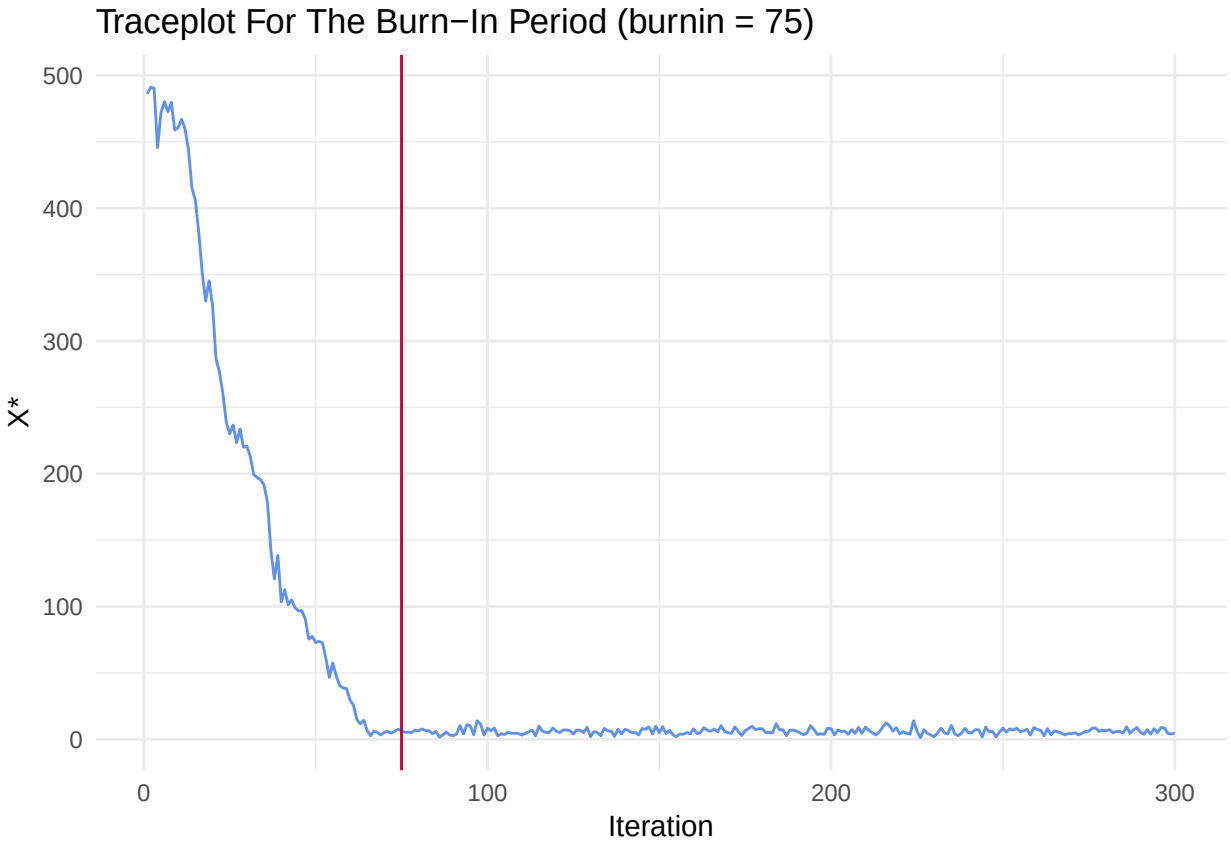


The following plot shows `10.000` samples with $x_0 = 5$ as a starting point and burn-in iterations set to `100`.

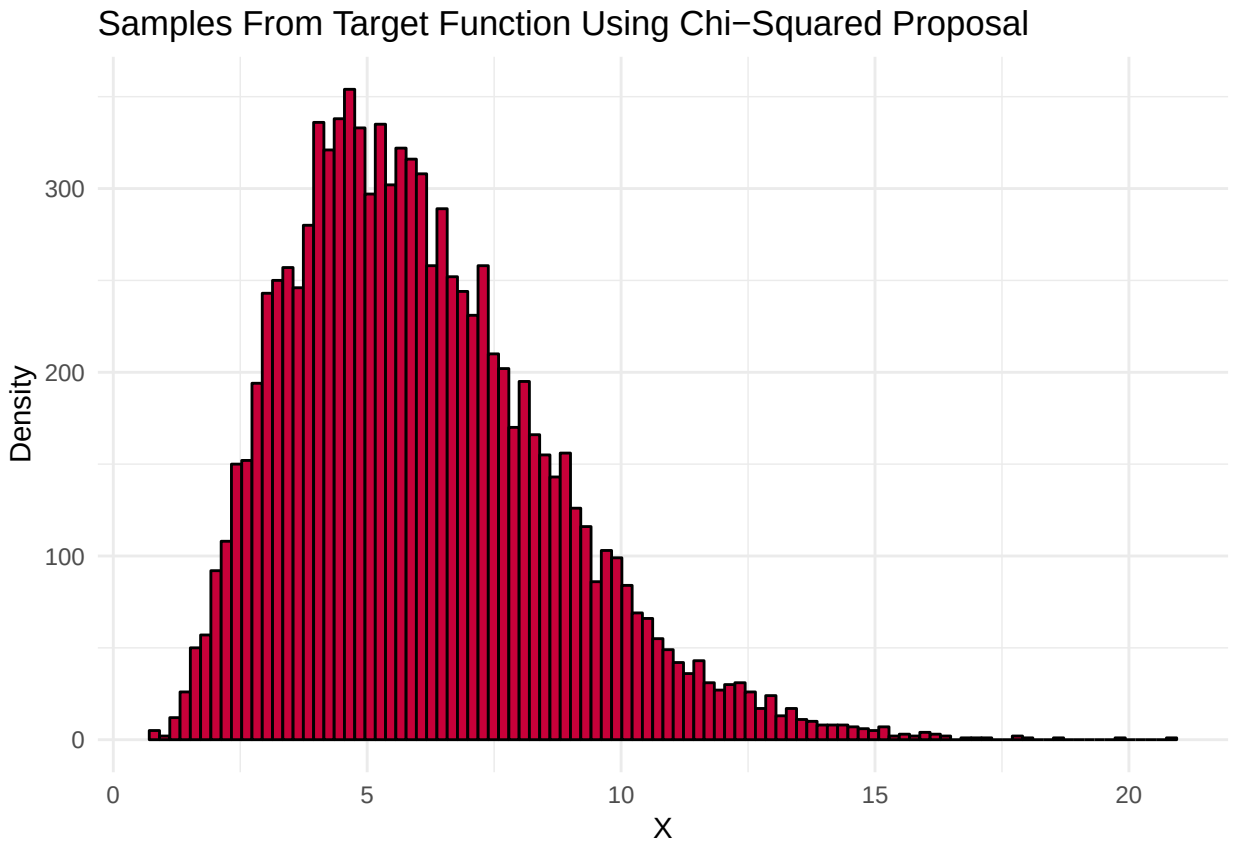


2. Perform Step 1 by using the chi–square distribution $\chi^2(\lfloor X_t + 1 \rfloor)$ as a proposal distribution, where $\lfloor x \rfloor$ is the floor function, meaning the integer part of x for positive x, i.e. $\lfloor 2.95 \rfloor = 2$.

Answer: The trace plot shows the burn-in period for the chi-square distribution. We see that it takes quite longer to achieve convergence. We set the burn-in iterations to `75` to be sure to converge.



Again you can see `10.000` samples with $x_0 = 5$ as a starting point and burn-in iterations set to `75`.



3. Compare the results of Steps 1 and 2 and make conclusions.

Answer: The samples seem to actually be taken from the real distribution and in this context it doesn't matter which proposal distribution we take. Comparing the burn-in times we see that it seems to take longer for the chi-squared proposal to converge. We have to keep in mind that this mostly depends on the initial state of the MCMC. It's not so easy to take an initial value which is far of from the converging value as we get densities around `0` which makes it quite diffucult to actually calculate `c` and α. As long as we take an initial state not too far off from a low density, we should be fine, as the algorithm automatically avoids really low density regions.

We assume as long as the proposal distribution mostly covers the target distribution and the initial state is selected quite reasonable (which is not always the case in the "real world") the Metropolis-Hastings works quite well.

4. Generate 10 MCMC sequences using the generator from Step 2 and starting points 1, 2, . . . , or 10. Use the Gelman–Rubin method to analyze convergence of these sequences.

Answer: We can see that the Gelman-Rubin factor is close to `1` with a sequence length of `1000` which indicates that our sequences have converged. The usage of the implementation from the 'code' package can be found below.

This self written function is also called with a sequence length of just `5` to demonstrate that the Gelman-Ruben factor increases if the sequence has not converged.

```
gelman_rubin_factor = function(sequence_matrix) {

  k = nrow(sequence_matrix)
  n = ncol(sequence_matrix)

  B = n/(k-1) * sum((rowMeans(sequence_matrix) - mean(sequence_matrix))^2)
```

```
  S_squared = colSums((sequence_matrix - rowMeans(sequence_matrix))^2 / (n - 1))
  W = sum(S_squared / k)
  V = ((n - 1) / n * W) + (1/n * B)
  R = sqrt(V / W)
  return(R)
}
```

```
## [1] 1.000394
```

```
## [1] 1.041849
```

The following output is based three sets of multiple sequences (with starting points 1...10) and used the package `coda`. The first set of sequences uses `5`, the second `10` and the last uses `50` iterations. We can see that the point estimate (at `95%` confidence) converges towards `1` with an increasing sequence length which indicates that the set has converged.

The Gelman-Rubin factor works by comparing the variance of the sequences within a set. As more iterations are performed, the MCMC depends less on the starting point until it finally holds no more information about that and the sequences have the same variance compared to themselves and compares to the others. This is when the Gelman-Ruby factor is near `1`.

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]       1.58        6.7
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]       1.07        1.3
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]       1.01       1.04
```

5. Estimate

$$\int_0^\infty x f(x) dx$$

using the samples from Steps 1 and 2.

Answer: To estimate this integral we take our previous samples. We then calculate:

$$\int_0^\infty x f(x) dx = E[x] \sim \frac{1}{n} \sum_{i=1}^{n} x_i$$

where x_i are our samples. As the decomposed left-side function is just x, this is all we have to do. The limits are given by the proposing function.

```
f_integrate = function(x) {
  return(x * df(x))
}

valid_samples_lnorm = results_lnorm[!is.na(results_lnorm)]

sum(valid_samples_lnorm) / length(valid_samples_lnorm)
```

```
## [1] 3.658112
```

```
valid_samples_chisquared = results_chisquared[!is.na(results_chisquared)]

sum(valid_samples_chisquared) / length(valid_samples_chisquared)
```

```
## [1] 6.057624
```

6. The distribution generated is in fact a gamma distribution. Look in the literature and define the actual value of the integral. Compare it with the one you obtained.

Answer: We can see that it is the following Gamma distribution and that the calculated integral is (almost) the same as the one we would expect.

$$\Gamma(6,1) \Rightarrow E[\Gamma(6,1)] = 6 * 1 = 6$$

Taking into account that $c = 120$ we can easily calculate the full integral which is:

$$\int_0^\infty x f(x) dx = E[\Gamma(6,1)] * c = 6 * 120 = 720$$

2 Question 2: Gibbs Sampling

concentration of a certain chemical was measured in a water sample, and the result was stored in the data `chemical.RData` having the following variables:

- X : day of the measurement
- Y : measured concentration of the chemical.

The instrument used to measure the concentration had certain accuracy; this is why the measurements can be treated as noisy. Your purpose is to restore the expected concentration values.

1. Import the data to R and plot the dependence of Y on X. What kind of model is reasonable to use here?
2. A researcher has decided to use the following (random–walk) Bayesian model (n=number of observations, $\vec{\mu} = (\mu_1, ..., \mu_n)$ are unknown parameters):

$$Y_i = \mathcal{N}(\mu_i, \text{variance} = 0.2), \quad i = 1, ..., n$$

where the prior is

$$p(\mu_1) = 1$$

$$p(\mu_{i+1}|\mu_i) = \mathcal{N}(\mu_i, 0.2), i = 1, ..., n1.$$

Present the formulae showing the likelihood $p(\vec{Y}|\vec{\mu})$ and the prior $p(\vec{\mu})$. **Hint:** a chain rule can be used here $p(\vec{\mu}) = p(\mu_1)p(\mu_2|\mu_1)p(\mu_3|\mu_2)...p(\mu_n|\mu_n 1)$

3. Use Bayes' Theorem to get the posterior up to a constant proportionality, and then find out the distributions of $(\mu_i|\vec{\mu}_{-i}, \vec{Y})$, where μ_{-i} is a vector containing all μ values except of μ_i.

Hint A: consider for separate formulae for $(\mu_1|\vec{\mu}_{-1}, \vec{Y}), (\mu_n|\vec{\mu}_{-n}, \vec{Y})$ and then a formula for all remaining $(\mu_i|\vec{\mu}_{-i}, \vec{Y})$.

Hint B:

$$e^{-\frac{1}{d}\left((x-a)^2+(x-b)^2\right)} \propto e^{-\frac{(x-(a+b)/2)^2}{d/2}}$$

Hint C:

$$e^{-\frac{1}{d}\left((x-a)^2+(x-b)^2+(x-c)^2\right)} \propto e^{-\frac{(x-(a+b+c)/3)^2}{d/3}}$$

4. Use the distributions derived in Step 3 to implement a Gibbs sampler that uses $\vec{\mu}^0 = 0, ..., 0$ as a starting point. Run the Gibbs sampler to obtain 1000 values of $\vec{\mu}$ and then compute the expected value of $\vec{\mu}$ versus X and Y versus X in the same graph. Does it seem that you have managed to remove the noise? Does it seem that the expected value of $\vec{\mu}$ can catch the true underlying dependence between Y and X?

5. Make a trace plot for μ_n and comment on the burn–in period and convergence.

```
# Loading RData
#data2 = get(load("data.RData"))
#head(data2)
```

3 Source Code

```
knitr::opts_chunk$set(echo = TRUE, cache = FALSE, include = TRUE, eval = TRUE)
library(knitr)
library(readxl)
library(ggplot2)
library(gridExtra)
library(coda)
set.seed(12345)

# Target function with original scaling
f = function(x) {
  return(120 * x^5 * exp(-x))
}

# Target function
df = function(x) {
  x = ifelse(x <= 0, 0.000001, x)
  return(x^5 * exp(-x))
}


sequence = seq(from = 0.01, to = 20, by = 0.01)

real_f = f(sequence)
plotdf = data.frame(sequence, real_f)

ggplot(plotdf) +
  geom_line(aes(x = sequence, y = real_f), color = "#6091ec") +
  labs(title = "Target Density Function", y = "Density",
  x = "X", color = "Legend") +
  theme_minimal()
```

```r
#' Metropolis Hasting Algorithm
#'
#' @param n Number of samples from the target distribution.
#' @param x_0 Initial state from Pi.
#' @param b Burn-in steps to remove from samples.
#' @param proposal Selects the proposal function.
#' @param keep_burnin Decides if to keep the samples during the burn-in period.
#'
#' @return Returns a list containing samples.
#' @export
#'
#' @examples
metropolis_hastings = function(n = 1, x_0 = 1, b = 50, proposal = "lnorm",
                               keep_burnin = FALSE) {

  # Vectors to store the samples in
  samples = c()

  # Samples from proposel from the random walk (MC)
  xt = x_0
  xt_1 = x_0

  while (length(samples) < n) {

    # Generate proposal state
    if (proposal == "lnorm") {
      x_star = rlnorm(n = 1, meanlog = log(xt), sdlog = 1)

      # Calculate correction factor C
      c = dlnorm(xt_1, meanlog = xt, sdlog = 1) /
      dlnorm(x_star, meanlog = xt, sdlog = 1)
    }
    else if (proposal == "chisquared") {
      x_star = rchisq(n = 1, df = floor(xt))

       # Calculate correction factor C
      c = dchisq(x = xt_1, df = floor(xt)) /
      dchisq(x_star, df = floor(xt))
    }
    else {
      stop("Invalid proposal.")
    }

    # Calculate acceptance probability alpha
    if (df(xt_1) <= 0) {
      # We need this to avoid troublesome areas where the density is so low that
      # it's 0 from a computiationally point of view.
      alpha = 0
    }
    else {
      alpha = min(1, df(x_star)/df(xt_1) * c)
    }
```

```
    # Generate u from uniform
    u = runif(n = 1, min = 0, max = 1)

    # Decide if to accept or reject the proposal
    if (u <= alpha) {
      # Accept
      xt_1 = xt
      xt = x_star
      samples = c(samples, x_star)
    }
    else {
      # Reject
      xt = xt_1
    }
  }

  # Return samples
  if (keep_burnin) return(samples)
  return(samples[b+1:length(samples)])
}


burnin = 10

results = metropolis_hastings(100, x_0 = 40, b = burnin, proposal = "lnorm",
                              keep_burnin = TRUE)

plotdf = data.frame(index = 1:length(results), values = results)

ggplot(plotdf) +
  geom_line(aes(x = index, y = values), color = "#6091ec") +
  labs(title = "Traceplot For The Burn-In Period (burnin = 10)", y = "X*",
  x = "Iteration", color = "Legend") +
  geom_vline(xintercept = burnin, color = "#C70039") +
  theme_minimal()


results_lnorm = metropolis_hastings(10000, x_0 = 5, b = 100, proposal = "lnorm")

plotdf = data.frame(results_lnorm)

ggplot(plotdf) +
  geom_histogram(aes(x = results_lnorm),
                 color = "#000000", fill = "#C70039", bins = length(results_lnorm)/100) +
  labs(title = "Samples From Target Function Using Normal Proposal", y = "Density",
  x = "X", color = "Legend") +
  theme_minimal()


burnin = 75

results = metropolis_hastings(300, x_0 = 500, b = burnin, proposal = "chisquared", keep_burnin = TRUE)
```

```r
plotdf = data.frame(index = 1:length(results), values = results)

ggplot(plotdf) +
  geom_line(aes(x = index, y = values), color = "#6091ec") +
  labs(title = "Traceplot For The Burn-In Period (burnin = 75)", y = "X*",
  x = "Iteration", color = "Legend") +
  geom_vline(xintercept = burnin, color = "#C70039") +
  theme_minimal()


results_chisquared = metropolis_hastings(10000, x_0 = 5, b = 100, proposal = "chisquared")

plotdf = data.frame(results_chisquared)

ggplot(plotdf) +
  geom_histogram(aes(x = results_chisquared),
                 color = "#000000", fill = "#C70039", bins = length(results_chisquared)/100) +
  labs(title = "Samples From Target Function Using Chi-Squared Proposal", y = "Density",
  x = "X", color = "Legend") +
  theme_minimal()


gelman_rubin_factor = function(sequence_matrix) {

  k = nrow(sequence_matrix)
  n = ncol(sequence_matrix)

  B = n/(k-1) * sum((rowMeans(sequence_matrix) - mean(sequence_matrix))^2)
  S_squared = colSums((sequence_matrix - rowMeans(sequence_matrix))^2 / (n - 1))
  W = sum(S_squared / k)
  V = ((n - 1) / n * W) + (1/n * B)
  R = sqrt(V / W)
  return(R)
}


k = 10 # row
n = 1000 # col

sequence_matrix = matrix(NaN, nrow = k, ncol = n)

for (i in 1:k) {
  sequence_matrix[i,] = metropolis_hastings(n, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE)
}

print(gelman_rubin_factor(sequence_matrix))

k = 10 # row
n = 5 # col

sequence_matrix = matrix(NaN, nrow = k, ncol = n)
```

```r
for (i in 1:k) {
  sequence_matrix[i,] = metropolis_hastings(n, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE)
}

print(gelman_rubin_factor(sequence_matrix))


results5 = list()
results10 = list()
results50 = list()
k = 10

for (i in 1:k) {
  results5[[i]] = mcmc(metropolis_hastings(5, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE))
}

for (i in 1:k) {
  results10[[i]] = mcmc(metropolis_hastings(10, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE))
}

for (i in 1:k) {
  results50[[i]] = mcmc(metropolis_hastings(50, x_0 = i, b = 0,
                                    proposal = "chisquared", keep_burnin = TRUE))
}

mcmc_list5 = mcmc.list(results5)
gelman.diag(mcmc_list5)

mcmc_list10 = mcmc.list(results10)
gelman.diag(mcmc_list10)

mcmc_list50 = mcmc.list(results50)
gelman.diag(mcmc_list50)


f_integrate = function(x) {
  return(x * df(x))
}

valid_samples_lnorm = results_lnorm[!is.na(results_lnorm)]

sum(valid_samples_lnorm) / length(valid_samples_lnorm)


valid_samples_chisquared = results_chisquared[!is.na(results_chisquared)]

sum(valid_samples_chisquared) / length(valid_samples_chisquared)
```

```
# Loading RData
#data2 = get(load("data.RData"))
#head(data2)
```