TMA4300 Computer Intensive Statistical Methods Exercise 1, Spring 2021

Group members: Eide, Jonathan and Lima-Eriksen, Leik

30.01.2023

Problem A

Part 1 - the exponential distribution

The exponential distribution with rate parameter λ has CDF $F(x) = 1 - e^{-\lambda x}$. Sampling from this distribution is equivalent to sampling uniformly on the inverse CDF, i.e.:

$$u \sim U[0,1] \tag{1}$$

$$f(x) = F^{-1}(u) = -\frac{1}{\lambda}\log u \tag{2}$$

This is implemented in R the following way:

```r
sample_exponential = function(n, lambda) {
  # Sample n times from U[0, 1]
  u = runif(n)

  # Evaluate inverse CDF in u:
  return(-1/lambda * log(u))
}
```

We then want to make sure that our function works correctly. We start off by checking that the sample mean matches the expected value $E[x] = \frac{1}{\lambda}$, and that the sample variance matches the variance $Var[x] = \frac{1}{\lambda^2}$:

```r
# Parameters
n = 100000
lambda = 3.2

# Draw n samples from exponential distribution
# with rate parameter lambda:
x = sample_exponential(n, lambda)

# Compare theoretical and sample mean and variance:
cat("Theoretical mean:", 1/lambda, "Sample mean", mean(x), "\n")
```

```
## Theoretical mean: 0.3125 Sample mean 0.3124931
```

```
cat("Theoretical variance:", 1/lambda^2, "Sample variance", var(x), "\n")
```

```
## Theoretical variance: 0.09765625 Sample variance 0.09661628
```

We see that there is a good match between the theoretical and obtained values, which indicates that the function works as expected. Of course they would not be exactly equal because we only obtain a finite number of samples.

We then proceed to verify that the samples ensemble the same distribution as the exponential distribution. A good way to visualize this would be to plot a histogram of the samples, and then superimpose the pdf $f(x) = \lambda e^{-\lambda x}$. This is shown in the figure below:

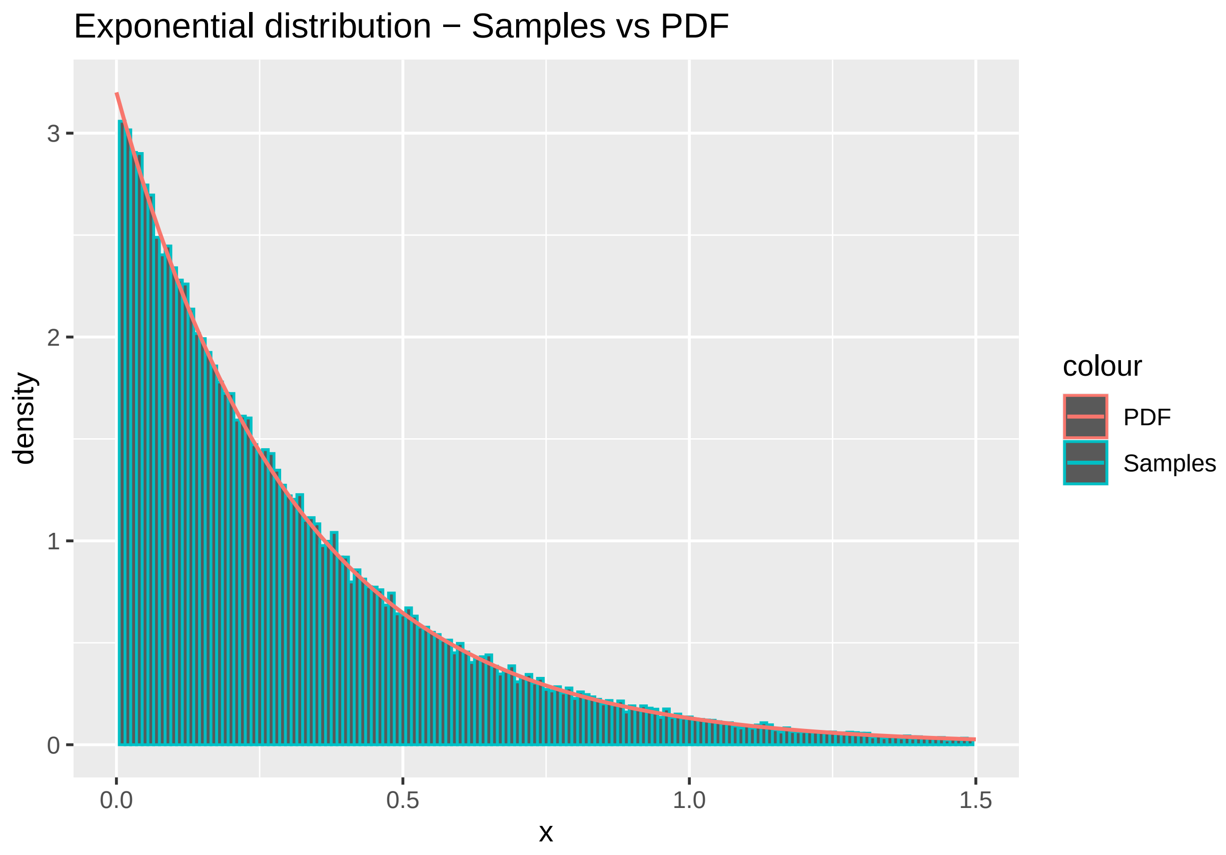
```
library(ggplot2)
# Plot the samples which we generated in the
# previous code block, together with the pdf
df = data.frame(x=x)
ggplot(df, aes(x=x)) +
  geom_histogram(aes(y = ..density.., color = "Samples"), binwidth=0.01) +
  stat_function(fun=dexp,geom = "line",size=0.7,args=(mean=lambda),aes(color="PDF")) +
  ggtitle("Exponential distribution - Samples vs PDF") +
  labs(x='x', y='density') +
  xlim(0, 1.5)
```

```
## Warning: Using ‘size‘ aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use ‘linewidth‘ instead.

## Warning: The dot-dot notation (‘..density..‘) was deprecated in ggplot2 3.4.0.
## i Please use ‘after_stat(density)‘ instead.

## Warning: Removed 796 rows containing non-finite values (‘stat_bin()‘).

## Warning: Removed 2 rows containing missing values (‘geom_bar()‘).
```



From the plot we see that the samples follow the exponential distribution perfectly. We thereby conclude that the function samples as expected.

Part 2 - Piecewise continuous pdf

We then want to create a function for sampling from the piecewise continuous pdf $g(x)$ as shown below.

$$g(x) = \begin{cases} cx^{\alpha-1}, & 0 < x < 1, \\ ce^{-x}, & 1 \le x, \\ 0, & \text{otherwise.} \end{cases}$$

First, note that the normalizing constant c can be found by using the property $\int_{-\infty}^{\infty} g(x)dx = 1$:

$$\begin{aligned} 1 &= \int_0^1 cx^{\alpha-1}dx + \int_1^{\infty} ce^{-x}dx && (3) \\ &= \frac{c}{\alpha} + \frac{c}{e} && (4) \\ \Rightarrow c &= \frac{\alpha e}{\alpha + e} && (5) \end{aligned}$$

Our aim is then to find an analytic expression for the inverse CDF, so that we can sample uniformly from it to generate samples from $g(x)$ in the same way as we did in Part 1. We start off by first finding the CDF $G(X) = P(X \le x) = \int_{-\infty}^{x} g(x)dx$:

$$G(x) = \begin{cases} \int_0^x cx^{\alpha-1}dx, & x \in (0,1) \\ \int_0^x cx^{\alpha-1}dx + \int_1^x ce^{-x}dx, & x \in [1,\infty) \end{cases} \tag{6}$$

$$= \begin{cases} \frac{c}{\alpha}x^{\alpha}, & x \in (0,1) \\ \frac{c}{\alpha} + c(e^{-1} - e^{-x}), & x \in [1,\infty) \end{cases} \tag{7}$$

Then the inverse CDF, $G^{-1}(x)$ is found by solving $G^{-1}(G(x)) = x$. The definition limits for the inverse function is obviously different, and is found by evaluating $G^{-1}(x) = 1$, since this is the x-value for which the analytic expression changes for the CDF.

$$G^{-1}(x) = \begin{cases} (\frac{\alpha}{c}x)^{1/\alpha}, & x \in (0, \frac{c}{\alpha}), \\ \ln \frac{c}{1-x}, & x \in [\frac{c}{\alpha}, 1), \\ 0, & \text{otherwise.} \end{cases}$$

To sample $x \sim g(x)$ is then equivalent to sampling $u \sim U[0,1]$, and evaluate \$ x = G^{-1}(u)\$. Below the `density_g` function allows us to evaluate the density analytically for given values of x. Furthermore, `sample_g` is used to sample from g using the inverse CDF as previously specified.

```
# Define the pdf for g(x):
density_g = function(x, alpha) {
  # Normalizing constant c:
  c = alpha*exp(1)/(alpha + exp(1))

  # Create an empty vector of same length as x:
  density = vector(length = length(x))

  # All elements corresponding to x < 1:
  density[x < 1.] = c*x[x<1.]^(alpha-1)

  # All elements corresponding to x >= 1:
  density[x >= 1.] = c*exp(-x[x>=1.])
  return(as.double(density))
}

# Sampler for g(x):
sample_g = function(n, alpha) {
  # Normalizing constant c:
  c = (alpha * exp(1)) / (alpha + exp(1))

  # Draw n samples from U[0, 1]
  u = runif(n)

  # Create an empty vector of length n:
  samples = vector(length=n)

  # Set all elements corresponding to u < c/alpha
  # equal to the equation as defined in the inverse CDF:
  samples[u < c/alpha] = (alpha/c*u[u < c/alpha])^(1/alpha)

  # Set all elements corresponding to u >= c/alpha
  # equal to the equation as defined in the inverse CDF:
  samples[u >= c/alpha] = log(c / (1 - u[u >= c/alpha]))
  return(samples)
}
```

We then want to compare the expected value and variance with the empirical mean and variance respectively. The moments can be calculated as follows:

$$E[X] = \int_0^\infty x g(x) dx = \frac{c}{\alpha+1} + 2\frac{c}{e}$$

$$Var[X] = E[X^2] - E[X]^2 = \int_0^\infty x^2 g(x) dx - E[X]^2 = \frac{c}{\alpha+2} + 5\frac{c}{e} - (\frac{c}{\alpha+1} + 2\frac{c}{e})^2$$

For $n = 1\text{E}{+}5$ samples with $\alpha = 0.7$, the empirical and theoretical mean and variances are respectively:

```
  # Parameters:
  n = 100000
  alpha = 0.7

  # Draw n samples from our g(x) sampler
  # with parameter alpha:
  x = sample_g(n, alpha)

  # Normalizing constant:
  c = (alpha * exp(1)) / (alpha + exp(1))

  # Theoretical mean and variance use the expressions as defined above
  cat("Theoretical mean:", c/(alpha+1) + 2*c/exp(1), "Sample mean:", mean(x), "\n")
```

```
## Theoretical mean: 0.7370055 Sample mean: 0.7377679
```

```
  cat("Theoretical variance:", c/(alpha+2) + 5*c/exp(1) - (c/(alpha+1) + 2*c/exp(1))^2, "Sample varianc
```

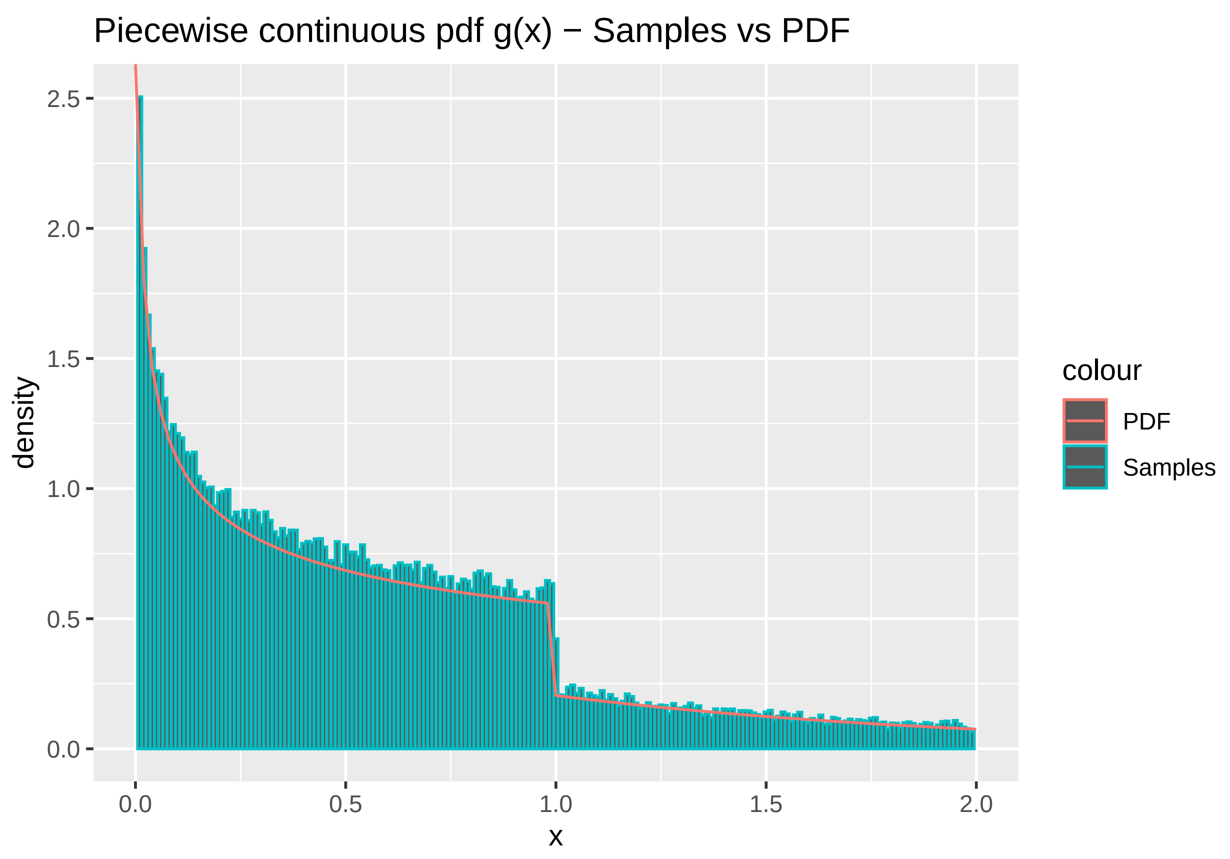
```
## Theoretical variance: 0.6868969 Sample variance: 0.6893989
```

Lastly, we create a histogram of the samples with the pdf superimposed. They seem to coincide well.

```
  library(ggplot2)
  df = data.frame( x=x)
  ggplot(df, aes(x = x)) +
    geom_histogram(aes(y = ..density.., color = "Samples"), binwidth=0.01) +
    stat_function(fun=density_g,geom = "line",size=0.5,args=(mean=alpha),aes(color="PDF")) +
    xlim(0, 2) +
    ggtitle("Piecewise continuous pdf g(x) - Samples vs PDF") + labs(x='x', y='density')
```

```
## Warning: Removed 7567 rows containing non-finite values (`stat_bin()`).
```

```
## Warning: Removed 2 rows containing missing values (`geom_bar()`).
```



Part 3 - The Box-Muller Algorithm

In order to generate n independent samples from the standard normal distribution, we can use the Box-Muller algorithm. The algorithm makes it possible to generate two i.i.d. samples from $N(0,1)$ according to $y_1 = \sqrt{x}_2 \cos x_1$ and $y_2 = \sqrt{x}_2 \sin x_1$, where $x_1 \sim U[0, 2\pi]$ and $x_1 \sim \exp(1/2)$. x_1 can be generated using `runif(0, 2*pi)`. For x_2 we can reuse the function `sample_exponential` we created in Problem A, Part 1 to sample from the exponential distribution with rate parameter $\lambda = 1/2$. Since all samples are i.i.d., this also scales to n samples. The algorithm has been implemented in a vectorized fashion for optimal performance. Since it only works for generating an even number of samples, we have to remove the last sample we generated if n is odd.

```r
# Function to sample n times from N(0, 1):
sample_standard_normal = function(n) {
  num_samples = n
  if((n %% 2) == 1) {
    # n is odd, so we should sample one extra:
    num_samples = num_samples + 1
  }

  # Sample from U[0, 2*pi]:
  x1 = runif(num_samples/2, 0, 2*pi)
  # Sample from exp(1/2):
  x2 = sample_exponential(num_samples/2, lambda=1/2)

  # Perform the transformations as defined above:
  y1 = sqrt(x2) * cos(x1)
  y2 = sqrt(x2) * sin(x1)

  # Remove the last sample if n is odd,
  # and concatenate the vectors together to retrieve
  # n samples:
  return(c(y1, y2)[0:n])
}
```

We then evaluate the validity in terms of moments by comparing theoretical and empirical variance and mean when drawing $n = 1E+5$ samples using the `sample_standard_normal` function as defined above:

```
  # Parameters:
  n = 100000

  # Draw n samples from N(0,1):
  x = sample_standard_normal(n)

  cat("Theoretical mean:", 0, "Sample mean:", mean(x), "\n")
```

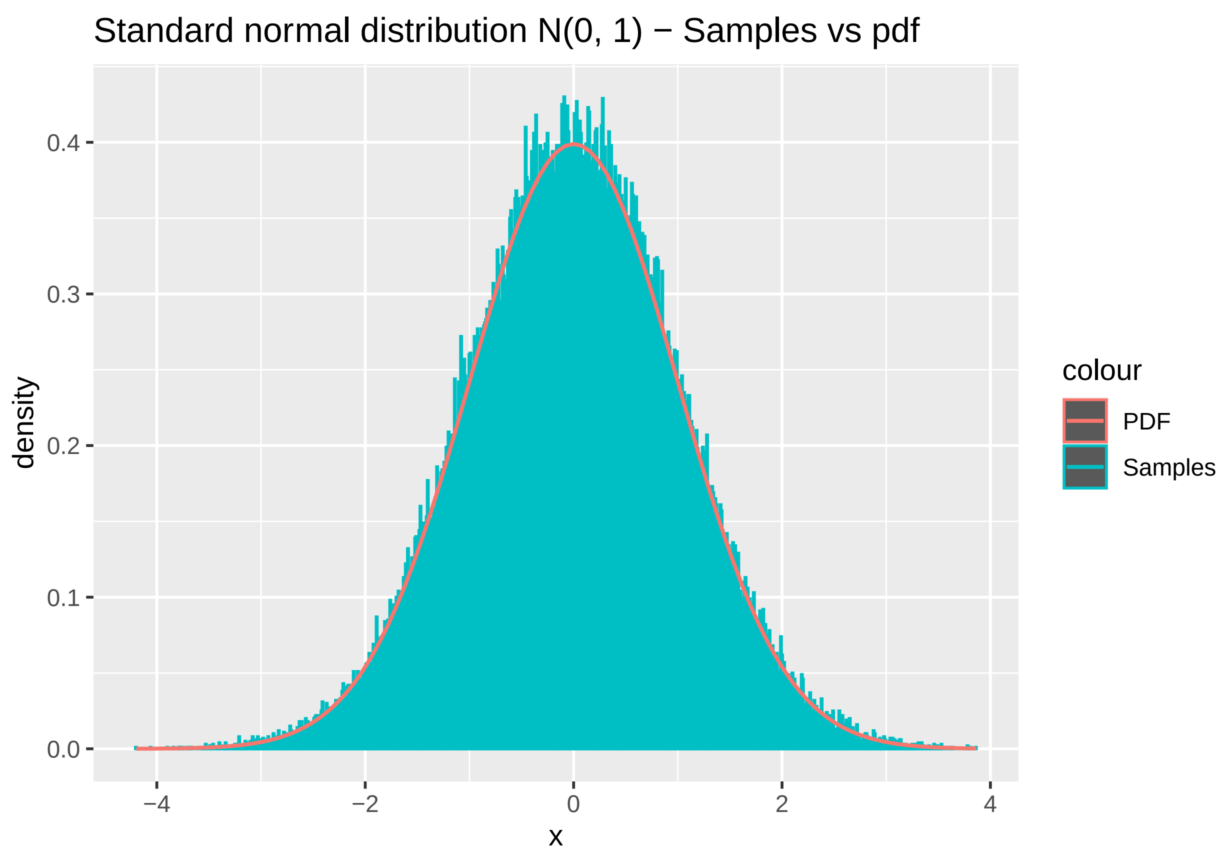
```
## Theoretical mean: 0 Sample mean: 0.001171339
```

```
  cat("Theoretical variance:", 1, "Sample variance:", var(x), "\n")
```

```
## Theoretical variance: 1 Sample variance: 0.9928945
```

As we see, the moments are matching well. As for the distribution of the samples versus the analytic distribution, this has been show below. The samples overlaps well with the analytic expression:

```
  # Plot the samples from N(0, 1) in a histogram together with the pdf
  df = data.frame(x=x)
  ggplot(df, aes(x=x)) +
    geom_histogram(aes(y = ..density.., color = "Samples"), binwidth=0.01) +
    stat_function(fun=dnorm,geom = "line",size=0.7,args=list(0,1),aes(color="PDF")) +
    ggtitle("Standard normal distribution N(0, 1) - Samples vs pdf") +
    labs(x='x', y='density')
```



Part 4 - d-variate normal distribution

Given $x \sim N_d(0, 1)$, we know that by linear transformation,

$$\boldsymbol{y} = A\boldsymbol{x} + \boldsymbol{\mu} \sim N_d(\boldsymbol{\mu}, AA^T), \quad \boldsymbol{x} \sim N_d(0, 1)$$

So in order to sample from the d-variate normal distribution $y = N_d(\mu, \Sigma)$, we can first sample from $N_d(0, 1)$ using the function `sample_standard_normal`, and get $\boldsymbol{y}$ by use of the linear transformation proposed above. The matrix A can be found by using Cholesky decomposition on Σ. This has been implemented in `sample_normal_d` as defined below.

```
# Function for sampling one time from the d-variate
# normal distribution:
sample_normal_d = function(d, mu, sigma) {
  # Draw d samples from N(0, 1):
  x = sample_standard_normal(d)

  # Apply the linear transformation:
  y = mu + t(chol(sigma)) %*% x

  return(y)
}
```

We then generate $n =$ 1E+5 samples using the $\boldsymbol{\mu}$ and Σ as defined below.

$$\mu = \begin{pmatrix} 1.5 \\ 3 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 5 & 7 \\ 7 & 10 \end{pmatrix}$$

```r
# Parameters:
d = 2
n = 100000
mu = c(1.5, 3)
sigma = matrix(c(5,7,7,10),ncol=d,nrow=d)

# We have to sample in a for-loop since our sampler
# cannot do this in a vectorized fashion:
samples = matrix(NA, ncol=d, nrow=n)
for (i in 1:n){
  samples[i,] = sample_normal_d(d,mu,sigma) # Draw n samples
}

average = apply(samples, 2, mean)
print("Mean vector:")
```

```
## [1] "Mean vector:"
```

```r
print.data.frame(data.frame(average))
```

```
##    average
## 1 1.514882
## 2 3.019893
```

```r
covmat = cov(samples)
print("Covariance matrix:")
```

```
## [1] "Covariance matrix:"
```

```r
print.data.frame(data.frame(covmat))
```

```
##         X1       X2
## 1 4.994707 6.990129
## 2 6.990129 9.983624
```

Problem B

Part 1

We want to find a way to sample from a gamma distribution $f(x)$ with parameters $\alpha \in (0, 1)$ and $\beta = 1$ as defined below. In order to do this, we can use rejection sampling with $g(x)$ as defined in Problem A, Part 2 as the proposal distribution. This is possible because the support of $g(x)$ includes the support of $f(x)$; in other words, $g(x) > 0$ whenever $f(x) > 0$. As a consequence of this, there exists a $c > 1$ such that $\left\{ \frac{f(x)}{g(x)} \leq c, x : f(x) > 0 \right\}$.

$$f(x) = \begin{cases} \frac{1}{\Gamma(\alpha)} x^{\alpha-1} e^{-x}, & 0 < x \\ 0, & \text{otherwise.} \end{cases}$$

Rejection sampling is done in two steps. First, we generate $x \sim g(x)$. We already have a sampler `sample_g` for this, which we will reuse here. We then compute $\alpha = \frac{1}{c} \cdot \frac{f(x)}{g(x)}$, and generate $u \sim U[0,1]$. If $u \leq \alpha$, then x is a sample from $f(x)$. If not,we have to redo the whole process until it is. The overall acceptance probability P_a is found to be the following:

$$P_a = P\left(U \leq \frac{1}{c} \cdot \frac{f(x)}{g(x)}\right) = \int_{-\infty}^{\infty} \frac{f(x)}{c \cdot g(x)} g(x) dx = c^{-1} \int_{-\infty}^{\infty} f(x) dx = c^{-1}$$

So in order to maximize the acceptance probability, we have to minimize c. Since we have to find a $c > \frac{f(x)}{g(x)} \quad \forall \quad x : f(x) > 0$, the lowest possible c would be the highest value the ratio $\frac{f(x)}{g(x)}$ can attain, i.e.

$$\begin{aligned} c &= \sup_x \frac{f(x)}{g(x)} && (8) \\ &= \sup_x \begin{cases} \frac{e^{-x}}{\Gamma(\alpha)} \left(\frac{1}{\alpha} + \frac{1}{e}\right), & 0 < x < 1, \\ \frac{x^{\alpha-1}}{\Gamma(\alpha)} \left(\frac{1}{\alpha} + \frac{1}{e}\right), & 1 \leq x \end{cases} && (9) \\ &= \frac{\alpha^{-1} + e^{-1}}{\Gamma(\alpha)} && (10) \end{aligned}$$

This means that the lowest acceptance probability we can achieve is $P_{\text{a,lowest}} = \frac{\Gamma(\alpha)}{\alpha^{-1}+e^{-1}}$. A sampling algorithm which uses this c has been implemented below. Here we exploit the fact that it on average takes c iterations to generate 1 sample from the target distribution $f(x)$. Then the actual number of samples we need in order to sample n times from $f(x)$ should be pretty close to $n \cdot c$ when n is large. This speeds up the execution significantly, since the algorithm does not require many iterations. After sampling once, we resample with c times the number of samples we are missing until we have enough samples. This should then converge pretty fast.

```r
# PDF of gamma for a given alpha and beta=1:
density_f = function(x, alpha) {
  return(dgamma(x, alpha, rate=1))
}

# Function for drawing n samples from Gamma(alpha, 1)
# When 0 < alpha < 1:
sample_f_small = function(n, alpha) {
  c = (1/alpha + exp(-1)) / gamma(alpha)
  samples = vector()
  repeat {
    num_missing_samples = n - length(samples)
    x = sample_g(as.integer(num_missing_samples*c), alpha)

    acceptance_probs = density_f(x, alpha) / (c * density_g(x, alpha))
    u = runif(as.integer(num_missing_samples*c))
    samples = c(samples, x[u <= acceptance_probs])
    if(length(samples) >= n) {
      # Break out of loop if we have enough samples:
      break
    }
  }
  # Do not return excess samples:
  return(samples[0:n])
}
```

We then want to evaluate the correctness of our sampler in terms of expectation value and variance. Below we have calculated the sample mean and variance, and compared them with their respective theoretical values. We see that they match well.

```
# Parameters:
n = 100000
alpha = 0.6 # 0 < alpha < 1

# Draw n samples from Gamma(alpha, 1):
x = sample_f_small(n, alpha)

cat("Theoretical mean:", alpha, "Sample mean:", mean(x), "\n")
```

```
## Theoretical mean: 0.6 Sample mean: 0.5986135
```

```
cat("Theoretical variance:", alpha, "Sample variance:", var(x), "\n")
```

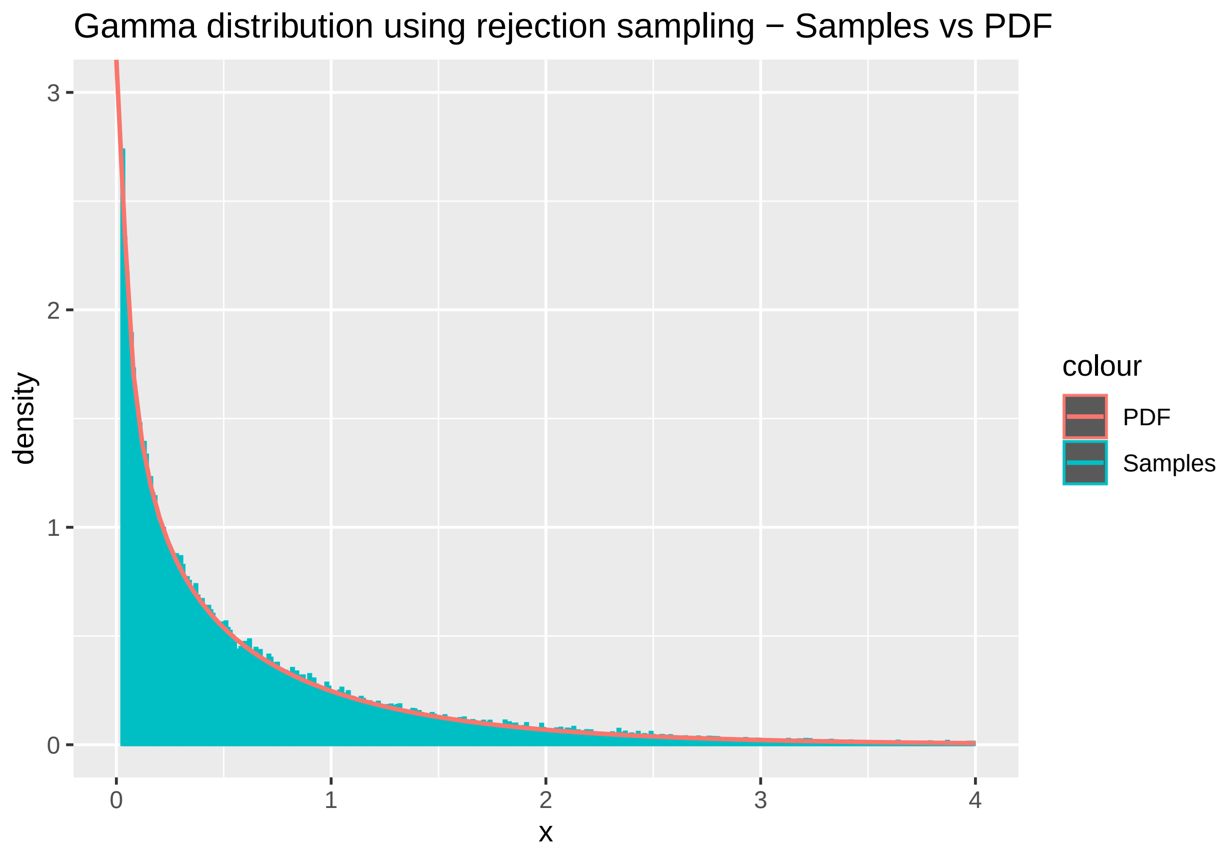
```
## Theoretical variance: 0.6 Sample variance: 0.602127
```

Furthermore, it is important to check that a histogram of the samples overlaps well with the analytic expression for $f(x)$. Such a plot has been created below. We see a good overlap, which further strengthens our belief that the sampler works as intended.

```
n = 100000
alpha = 0.6
x = sample_f_small(n, alpha)
df = data.frame(x)
ggplot(df, aes(x = x)) +
  geom_histogram(aes(y = ..density.., color = "Samples"), binwidth=0.01) +
  stat_function(fun=dgamma,geom = "line",size=0.8,args=(alpha=alpha),aes(color="PDF")) +
  xlim(0, 4) + ylim(0, 3) +
  ggtitle("Gamma distribution using rejection sampling - Samples vs PDF") + labs(x='x', y='density')
```

```
## Warning: Removed 641 rows containing non-finite values (‘stat_bin()‘).
```

```
## Warning: Removed 4 rows containing missing values (‘geom_bar()‘).
```



Part 2: Ratio of uniforms method

We now want to consider sampling from a Gamma distribution $f(x)$ with $\alpha > 1$ and $\beta = 1$. With an unbounded α, we cannot find a $c \geq \frac{f(x)}{g(x)} \quad \forall \quad x : f(x) > 0$ anymore. Therefore, sampling using ratio-of-uniforms seems more suitable. Define the area

$$C_f = \left\{ (x_1, x_2) : 0 \leq x_1 \leq \sqrt{f^*\left(\frac{x_1}{x_2}\right)} \right\}, \quad \text{where} \quad f^*(x) = \begin{cases} x^{\alpha-1}e^{-x}, & 0 < x, \\ 0, & \text{otherwise.} \end{cases}$$

and

$$a = \sqrt{\sup_x f^*(x)}, \quad b_+ = \sqrt{\sup_{x \geq 0}(x^2 f^*(x))} \quad \text{and} \quad b_- = \sqrt{\sup_{x \leq 0}(x^2 f^*(x))}$$

so that $C_f \subset [0, a] \times [b_-, b_+] \in \mathbb{R}^2$. The values of a, b_- and b_+ are found by means of differentiation:

$$\frac{d}{dx} f^*(x) = 0 \Leftrightarrow e^{-x}(\alpha - x - 1)x^{\alpha-2} = 0 \Rightarrow x = \alpha - 1 \tag{11}$$

$$\Rightarrow a = \sqrt{f^*(\alpha - 1)} \tag{12}$$

$$= \sqrt{(\alpha - 1)^{\alpha-1}e^{1-\alpha}} \tag{13}$$

$$\left\{\frac{d}{dx}x^2 f^*(x) = 0, x \geq 0\right\} \Leftrightarrow e^{-x}(\alpha - x + 1)x^\alpha = 0 \Rightarrow x = \alpha - 1 \tag{14}$$

$$\Rightarrow b_+ = \sqrt{x^2 f^*(\alpha - 1)} \tag{15}$$

$$= \sqrt{(\alpha + 1)^{\alpha+1} e^{-(\alpha+1)}} \tag{16}$$

$$b_- = 0$$

Then $y = x_1/x_2 \sim \text{Gamma}(\alpha, \beta = 1)$ if x_1 and x_2 are sampled uniformly inside C_f. This can be accomplished by letting $x_1 \sim U[0, a]$ and $x_2 \sim U[b_-, b_+]$, and selecting only $(x_1, x_2) \in C_f$. Note that α can grow arbitrarily large, which makes the x^α term in $f(x)$ grow even larger. This would result in numerical overflow in practical implementations, and is a problem. To solve this, we can sample in log-scale. Then $\ln x_1$ and $\ln x_2$ are sampled as follows:

$$x_1 = a \cdot U[0, 1] \tag{17}$$

$$\Rightarrow \ln x_1 = \ln a + \ln U[0, 1] \tag{18}$$

$$x_2 = b_- + (b_+ - b_-) \cdot U[0, 1] \tag{19}$$

$$\Rightarrow \ln x_2 = \ln b_+ + \ln U[0, 1] \tag{20}$$

And the condition for checking $(x_1, x_2) \in C_f$ is equivalent to

$$\ln x_1 \leq \ln \sqrt{f^*\left(\frac{x_2}{x_1}\right)} \tag{21}$$

$$\Rightarrow 2\ln x_1 \leq (\alpha - 1)(\ln x_2 - \ln x_1) - \exp(\ln x_2 - \ln x_1) \tag{22}$$

Furthermore,

$$\ln a = \frac{\alpha - 1}{2} \cdot (\ln(\alpha - 1) - 1) \tag{23}$$

$$\ln b_+ = \frac{\alpha + 1}{2} \cdot (\ln(\alpha + 1) - 1) \tag{24}$$

```r
# Function for drawing n samples from
# Gamma(alpha, 1) when alpha > 1:
sample_f_large = function(n, alpha) {
  loga = (alpha - 1)/2 * (log(alpha - 1) - 1)
  logbp = (alpha + 1)/2 * (log(alpha + 1) - 1)

  samples = vector()
  num_iterations = 0

  while(length(samples) < n) {
    num_iterations = num_iterations + 1

    # Sample in log-scale
    log_x1 = loga + log(runif(1))
    log_x2 = logbp + log(runif(1))

    if(2*log_x1 <= (alpha - 1)*(log_x2 - log_x1) - exp(log_x2 - log_x1)) {
      # Accept the sample:
      samples = c(samples, exp(log_x2 - log_x1))
    }
  }

  return(list(samples=samples, num_iterations=num_iterations))
}
```

We then evaluate the correctness in terms of comparing sampled and theoretical mean and variance:

```r
  # Parameters:
  n = 10000
  alpha = 50 # alpha > 1

  # Draw n samples from Gamma(alpha, 1) when alpha > 1
  samples = sample_f_large(n, alpha)$samples

  cat("Theoretical mean:", alpha, "Sample mean:", mean(samples), "\n")
```

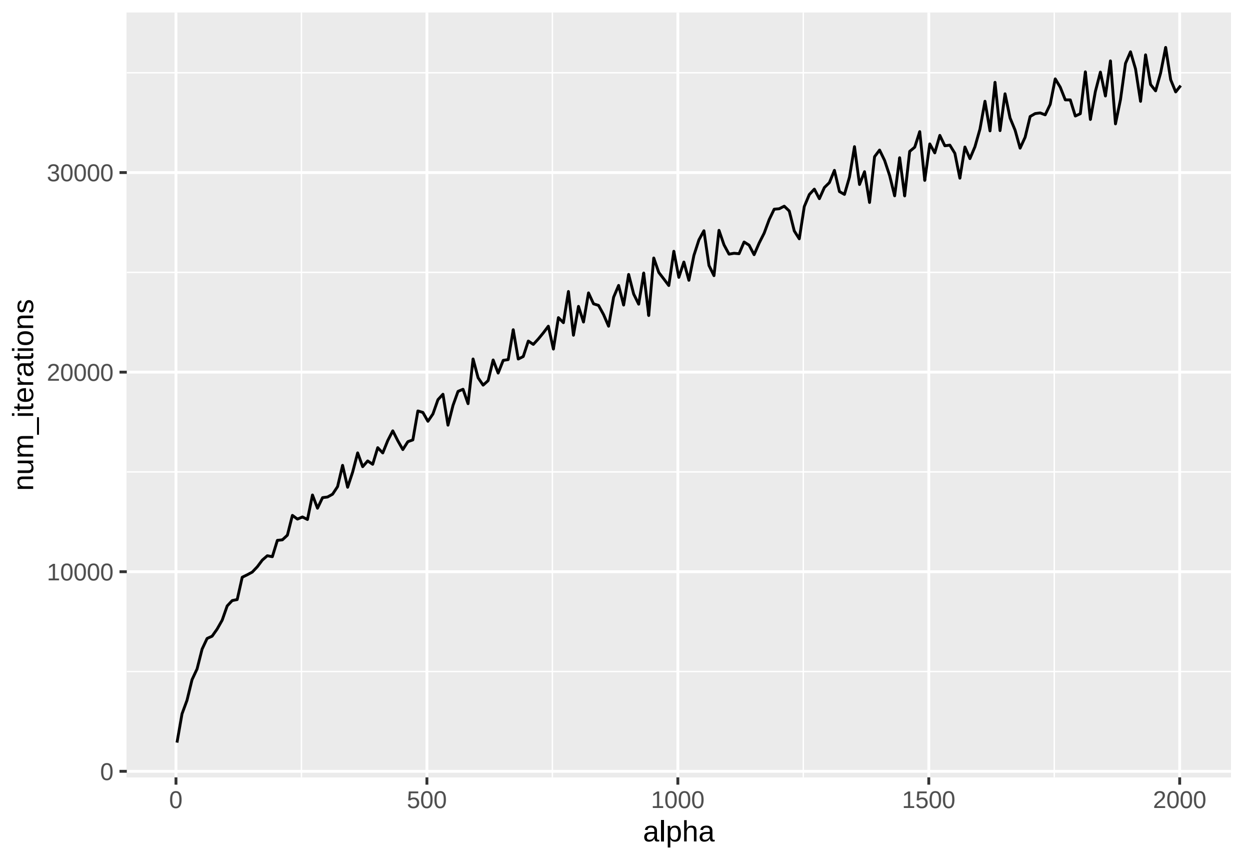
```
## Theoretical mean: 50 Sample mean: 50.08225
```

```r
  cat("Theoretical variance:", alpha, "Sample variance:", var(samples), "\n")
```

```
## Theoretical variance: 50 Sample variance: 49.47051
```

Another important aspect of the algorithm is how fast it executes. We let $n = 1000$, and check how many iterations are required for sampling when $\alpha \in (1, 2000] \subset \mathbb{Z}$:

```r
n = 1000
alpha = seq(2, 2002, 10)
num_iterations = vector()
for(a in alpha) {
  num_iterations = c(num_iterations, sample_f_large(n, a)$num_iterations)
}
df = data.frame(x=alpha, num_iterations=num_iterations)
ggplot(df, aes(x = alpha, y = num_iterations)) + geom_line()
```



From the plot it seems that the number of iterations n_i required for generating n samples has a $\log \alpha$ dependency, i.e. $n_i \propto \log \alpha$ for a fixed n. This is not too bad, since it implies that by increasing α a lot, n_i will increase only slightly.

Part 3 - Gamma sampler with arbitrary parameters

Recall that we created a Gamma sampler for $\alpha \in (0, 1)$ in Part 1 and $\alpha \in (1, \infty)$ in Part 2 for $\beta = 1$. We now want to create a Gamma sampler for an arbitrary $\alpha \in \mathbb{R}_+$ and β. First, two observations should be made; β is simply an inverse scale parameter. In other words, sampling from Gamma(α, β) is equivalent to sampling from Gamma$(\alpha, 1)$ and then divide by β. Second, Gamma$(1, 1)$ is the unscaled exponential distribution. And we already have a sampler for that distribution(`sample_exponential`). So our Gamma sampler should choose one of the three mentioned samplers based on α, and then divide the samples by β. The R code for our Gamma sampler is provided below:

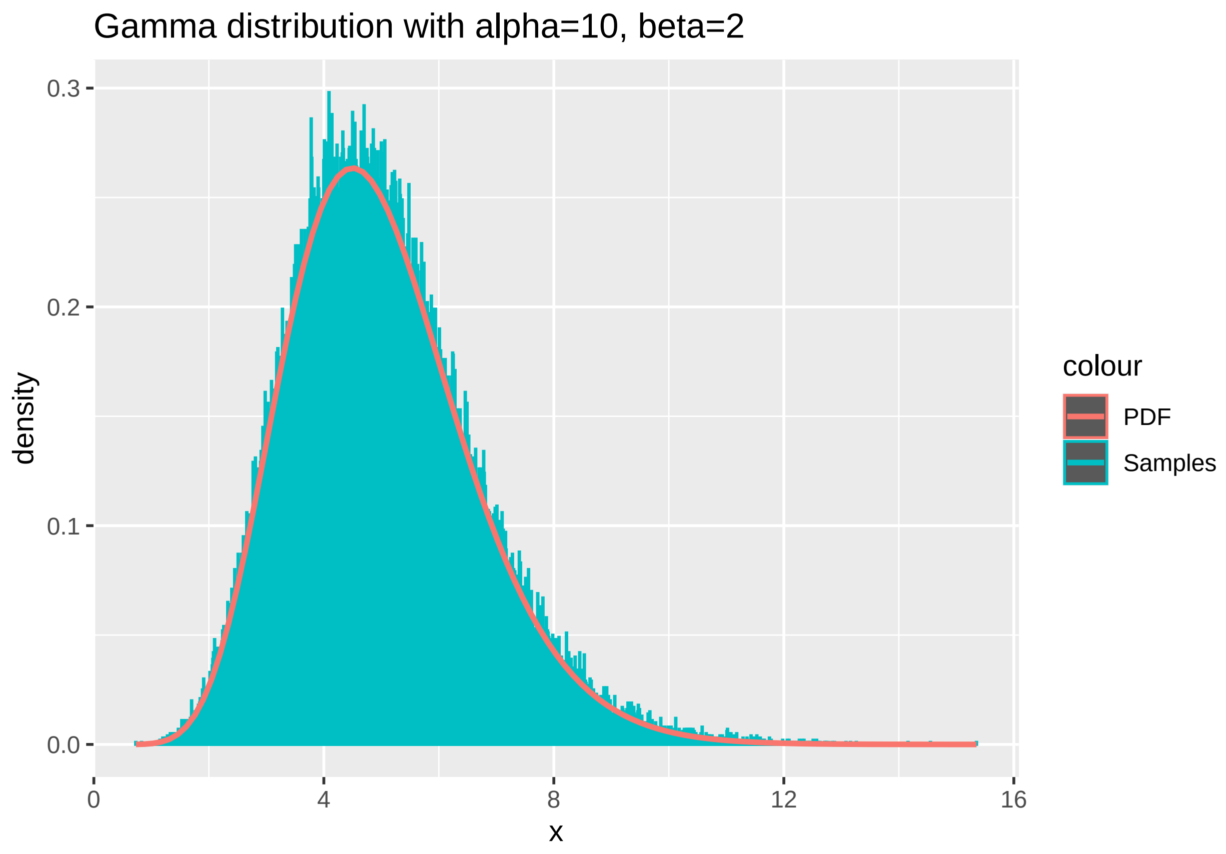
```r
sample_gamma = function(n, alpha, beta) {
  unscaled_samples = vector(length=n)
  if(alpha == 1.) {
    unscaled_samples = sample_exponential(n, 1)
  } else if(alpha < 1.) {
    unscaled_samples = sample_f_small(n, alpha)
  } else { # alpha > 1.
    unscaled_samples = sample_f_large(n, alpha)$samples
  }
  samples = unscaled_samples/as.double(beta)
  return(samples)
}
```

Now we can sample from e.g. Gamma$(10, 2)$ and check that it works as expected:

```
# Parameters:
n = 100000
alpha = 10
beta = 2

# Draw n samples from Gamma(alpha, beta)
x = sample_gamma(n, alpha, beta)

# Plot the samples in a histogram together with the pdf
df = data.frame(x)
ggplot(df, aes(x = x)) +
    geom_histogram(aes(y = ..density.., color = "Samples"), binwidth=0.01) +
    stat_function(fun=dgamma,geom = "line",size=1,aes(color='PDF'),args = list(shape=alpha, rate=beta
    ggtitle("Gamma distribution with alpha=10, beta=2") + labs(x='x', y='density')
```



## Part 4

Now we want to create a sampler for the beta(α, β) distribution. Define the independent stochastic variables $x \sim \text{Gamma}(\alpha, 1)$, $y \sim \text{Gamma}(\beta, 1)$. Their joint distribution is

$$f_{x,y}(x, y) = f_x(x) \cdot f_y(y) = \frac{1}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1} y^{\beta-1} \exp{(x+y)} \quad \forall \quad (x, y) \in \mathbb{R}^2$$

Define the transformations

$$z = g_1(x,y) = \frac{x}{x+y} \tag{25}$$
$$z' = g_2(x,y) = x+y \tag{26}$$

The inverse transformations then becomes

$$x = g_1^{-1}(z,z') = z \cdot z' \tag{27}$$
$$y = g_2^{-1}(z,z') = z' - z \cdot z' \tag{28}$$

The Jacobi determinant of the transformation is found as

$$\det J_{x,y}(z,z') = \det \begin{pmatrix} \frac{\partial x}{\partial z} & \frac{\partial x}{\partial z'} \\ \frac{\partial y}{\partial z} & \frac{\partial y}{\partial z'} \end{pmatrix} = \det \begin{pmatrix} z' & z \\ -z' & 1-z \end{pmatrix} = z'(1-z) + z \cdot z'$$

From bivariate transformations we know that

$$f_{z,z'}(z,z') = f_{x,y}(g_1^{-1}(z,z'), g_2^{-1}(z,z')) \cdot \det J_{x,y}(z,z') \tag{29}$$
$$= f_{x,y}(z \cdot z', z' - z \cdot z') \cdot \det J_{x,y}(z,z') \tag{30}$$
$$= |z'| \frac{1}{\Gamma(\alpha)\Gamma(\beta)} (z \cdot z')^{\alpha-1} (z' - z \cdot z')^{\beta-1} e^{-z'} \tag{31}$$

The total probability theorem then yields

$$f_Z(z) = \int_{-\infty}^{\infty} f_{Z,Z'}(z,z') dz' \tag{32}$$
$$= \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} z^{\alpha-1}(1-z)^{\beta-1}, \quad z \in (0,1) \subset \mathbb{R} \tag{33}$$
$$\Rightarrow z \sim \text{beta}(\alpha,\beta) \tag{34}$$

We can thus create a sampler by first sampling $x \sim \text{Gamma}(\alpha, 1)$, $y \sim \text{Gamma}(\beta, 1)$, and then applying the transformation $z = \frac{x}{x+y} \sim \text{beta}(\alpha, \beta)$:

```r
sample_beta = function(n, alpha, beta) {
  x = sample_gamma(n, alpha, 1)
  y = sample_gamma(n, beta, 1)
  z = x/(x+y)
  return(z)
}
```

The sampler should first be evaluated in terms of mean and variance. This has been done in the code block below:

```r
# Parameters:
n = 100000
alpha = 5
beta = 10
```

```r
# Draw n samples from beta(alpha, beta):
x = sample_beta(n, alpha, beta)

cat("Theoretical mean:", alpha/(alpha+beta), "Sample mean:", mean(x), "\n")
```

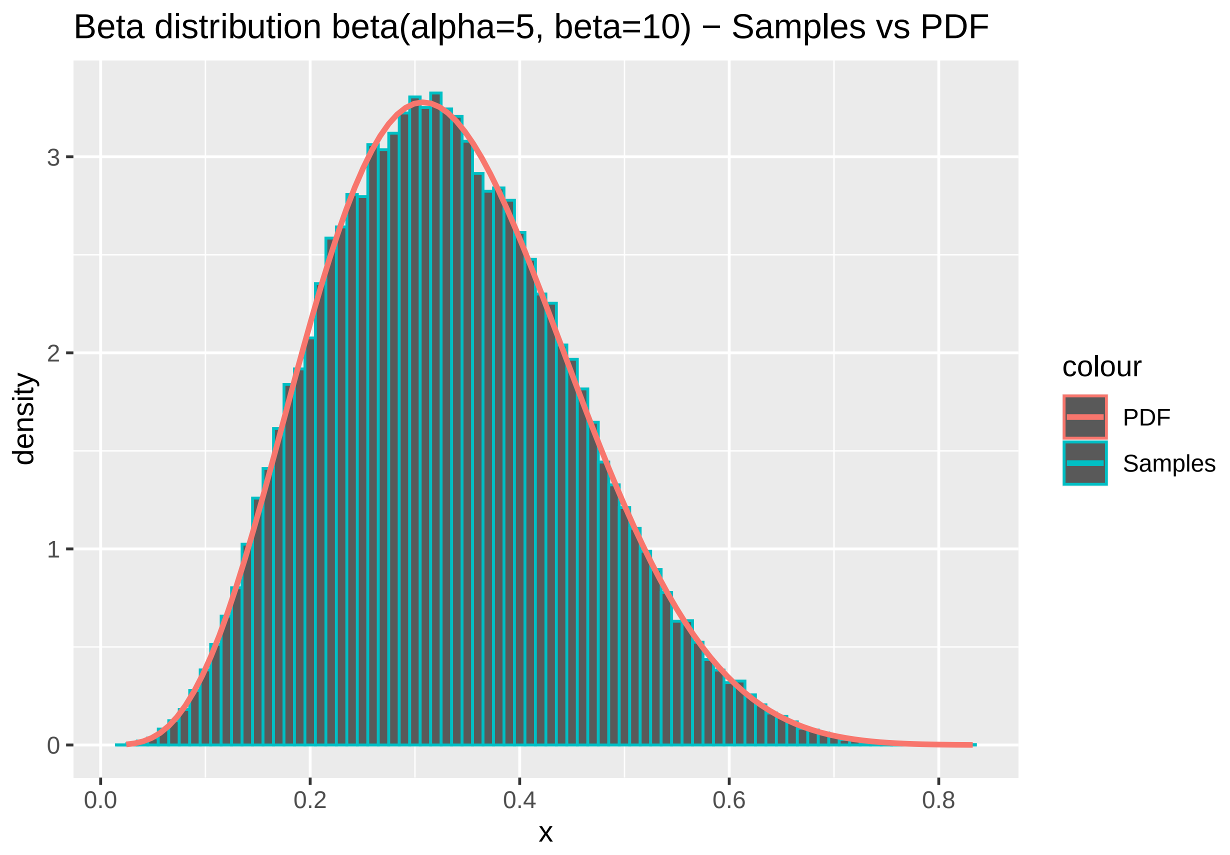
```
## Theoretical mean: 0.3333333 Sample mean: 0.3331395
```

```r
cat("Theoretical variance", alpha*beta/((alpha+beta)^2*(alpha + beta + 1)), "Sample variance:", var(x
```

```
## Theoretical variance 0.01388889 Sample variance: 0.01387597
```

We see that the sample mean and variance matches well with the theoretical values. Furthermore we want to check that the samples have the correct distribution:

```r
df = data.frame(x=x)
ggplot(df, aes(x = x)) +
    geom_histogram(aes(y = ..density.., color = "Samples"), binwidth=0.01) +
    stat_function(fun=dbeta,geom = "line",size=1,aes(color='PDF'),args = list(shape1=alpha, shape2=be
    ggtitle("Beta distribution beta(alpha=5, beta=10) - Samples vs PDF") + labs(x='x', y='density')
```



So the distribution of the samples matches well with the pdf. We conclude that our beta sampler works as intended.

Problem C - Monte Carlo integration and variance reduction

Part 1 - Monte carlo estimation

In order to estimate $\theta = P(x > 4)$ given $x \sim N(0,1)$ it is possible to generate n samples from the normal distribution and check how many are above 4. This is implemented in R in the following way:

```r
#Estimate probability by monte carlo integration
estimate_monte_carlo = function(n) {
  x = sample_standard_normal(n)
  samples = rep(NA, n)

  #Set all values above 4 to 1 and all below to 0
  samples[x > 4] = 1
  samples[x <= 4] = 0
  return(samples)
}
```

We then verify that the estimator works as intended by using the function with $n = 100000$.

```r
n = 100000
x = estimate_monte_carlo(n)
est = mean(x)
confidence_interval = est + c(-1, 1)*qnorm(.975)*sqrt(var(x)/n) #Calculate variance

cat("Prob(x>4): ", est, "\n")
```

```
## Prob(x>4):  3e-05
```

```r
cat("Confidence:", confidence_interval, "\n")
```

```
## Confidence: -3.947233e-06 6.394723e-05
```

This seems to converge to what you would find in a table (approximated to 0.00003 in tables). Also note that by choosing larger n, the confidence interval becomes smaller - which is to be expected.

Part 2 - Importance sampling

Monte Carlo estimation is a brute force estimator which would converge slowly when the support of the pdf is small for the domain of interest. In our case, we want to estimate the portion of samples that falls outside 4 standard deviations on the right side of the distribution. This portion we know in fact is quite small as demonstrated in the example above, and thus we seek to find an estimator which converges faster. Importance sampling seems to be a better fit here. By choosing a proposal density which has its support consentrated around the region of interest, this should be more efficient than simply monte carlo. In other words, we get an estimator with significantly smaller variance for a given number of samples n.

A good proposal density in this case would be the $g(x)$ as defined below. This comes from the fact that it has good support in the region of interest, but in addition the weights needed are easy to calculate.

$$g(x) = \begin{cases} cx^{-\frac{1}{2}x^2}, & 4 > x \\ 0, & \text{otherwise.} \end{cases}$$

Then our estimator becomes

$$\hat{\theta}_{IS} = \frac{1}{n}\sum_{i=0}^{n-1} 1_{x_i \geq 4} \cdot w(x_i), \quad \text{where} \quad \frac{f(x)}{g(x)} = \frac{1}{c\sqrt{2\pi}x}$$

Here, $1_{x_i \geq 4}$ is the identity function, i.e. it is 1 when $x_i \geq 4$ and zero otherwise. The normalizing constant is easily found by noting that

$$1 = \int_4^\infty cg(x)dx \tag{35}$$

$$\Rightarrow c = e^8 \tag{36}$$

In order to sample from g(x), we can use inversion sampling as done in Problem A, Part 1, since an inverse of the CDF $G(x) = \int_{-\infty}^{x} g(x)dx$ is analytically tractable. The inverse of the cdf is

$$G^{-1}(u) = \sqrt{16 - 2ln(1-u)}$$

Sampling from $g(x)$ is then equivalent to sampling $u \sim U[0,1]$ and evaluating $x = G^{-1}(u) \sim g(x)$. We first generate n samples $\{x_1, x_2, ..., x_n\}$. Next up we have to calculate the weighted samples $w(x_i) \cdot 1_{x_i \geq 4}$ for each x_i. Lastly, we sum over all the weighted samples and divide by n in order to generate the estimate. An implementation of the algorithm has been provided below.

```r
generate_weighted_samples = function(n) {
  samples = rep(NA, n)
  sample_u = runif(n) #Generate uniform samples
  sample_x = sqrt(16-2*log(1-sample_u)) #Inverse sampling from G(x)
  samples[ sample_x <= 4.] = 0

  #Calculate 1*w(x) for all samples of X > 4
  samples[ sample_x > 4.] = 1/(exp(8)*sqrt(2*pi)*sample_x[ sample_x > 4.])

  return(samples)
}
```

In order to verify that the estimator is working, we test it for $n = 100000$ samples and check the output:

```r
n = 100000
samples = generate_weighted_samples(n)
est = mean(samples)

confidence_interval = est + c(-1, 1)*qnorm(.975)*sqrt(var(samples)/n)

cat("Estimated:", est, "\n")
```

```
## Estimated: 3.166838e-05
```

```r
cat("Confidence interval:", confidence_interval, "\n")
```

```
## Confidence interval: 3.165876e-05 3.167801e-05
```

Observe that the importance sampling estimate is much closer to the real value. Also, the confidence interval is significantly smaller. This confirms that estimating by the means of importance sampling is much better than using Monte Carlo, which is also what we initially suspected.

Part 3a)

An implementation of the algorithm presented in C2 with antithetic variables is implemented in R the following way:

```r
#Draw sample from g(x) using inverse transform

generate_weighted_samples_anth = function(n) {
  samples = rep(NA)
  samples_u_1 = runif(n)
  samples_u_2 = 1 - samples_u_1 # Calculate the antithetic variates
  samples_u = c(samples_u_1, samples_u_2) # Combine the variates
  sample_x = sqrt(16-2*log(1-samples_u)) #Inverse sampling from G(x)
  samples[ sample_x <= 4.] = 0

  #Calculate 1*w(x) for all samples of X > 4
  samples[ sample_x > 4.] = 1/(exp(8)*sqrt(2*pi)*sample_x[ sample_x > 4.])

  return(samples)
}
```

Part 3b)

To get a fair comparison of the functions, 50000, half of 100000 is used as it generates twice the amount of samples for each n.

```r
n = 50000

samples = generate_weighted_samples_anth(n)

est = mean(samples)

confidence_interval = est + c(-1, 1)*qnorm(.975)*sqrt(var(samples)/(2*n)) #Multiply n by 2 as number of

cat("Estimated:", est, "\n")
```

```
## Estimated: 3.167293e-05
```

```r
cat("Confidence interval:", confidence_interval, "\n")
```

```
## Confidence interval: 3.166329e-05 3.168257e-05
```

One can see that the confidence intervals are approximately the same for both functions, even though half the uniform variables are created. There are two advantages for using this technique; first, it reduces the number of samples we need to obtain since we can use the same sample twice. Second, it reduces the variance of the sample paths. So the method is much more efficient than the previously proposed.

Part D - Rejection sampling and importance sampling

Part 1 - Rejection sampling algorithm

In order to sample from

$$f(\theta|\mathbf{y}) \propto (2+\theta)^{y_1}(1-\theta)^{y_2+y_3}\theta^{y_4}, \quad \theta \in (0,1)$$

a rejection sampling algorithm may be used. $f(\theta|\mathbf{y})$ is sampled using $U(0,1)$ as the proposal density. This can be used as the proposal density as the uniform density is nonzero for all values where $f(\theta|\mathbf{y})$ is nonzero.

This is implemented in R the following way:

```r
#Function to be sampled from
f_function = function(x) {
    return((2 + x)^125 * (1 - x)^38 * x^34)
}

#Find highest value of f_function to set as max c
target_c = optimize(f_function, c(0, 1), maximum = TRUE)

#Find normalizing constant
n_c = integrate(f_function, 0, 1)$val

generate_samples = function(n) {
    samples = rep(NA, n)
    run_cnt = 0
    for (i in 1:n) {
        done = 0
        while (done == 0) {
            x_sample = runif(1)  # Proposal sample
            a = f_function(x_sample)/target_c$objective #Acceptance probability
            u = runif(1) #Used to determine acceptance
            # Accept if u<a(alpha)
            if (u <= a) {
                samples[i] = x_sample
                done = 1
            }
            run_cnt = run_cnt + 1
        }
    }
    return(list(samples = samples, iterations = run_cnt))
}
```

Part 2 - Posterior mean

The posterior mean θ can be estimated by Monte-Carlo integration using $N = 10000$ samples from $f(\theta|y)$.

```r
n = 10000

# Function used to find actual mean
g = function(x) {
  return(x*f_function(x)*x)
}
```

```r
f_func_density = function(x) {
  return(f_function(x)/n_c)
}


y = generate_samples(n) #Generate samples from the distribution

est_mean = sum(y$samples)/n

cat("Estimated mean: ", sum(y$samples)/n, "Theoretical mean: ", integrate(g, 0, 1)$val/n_c, "\n")
```

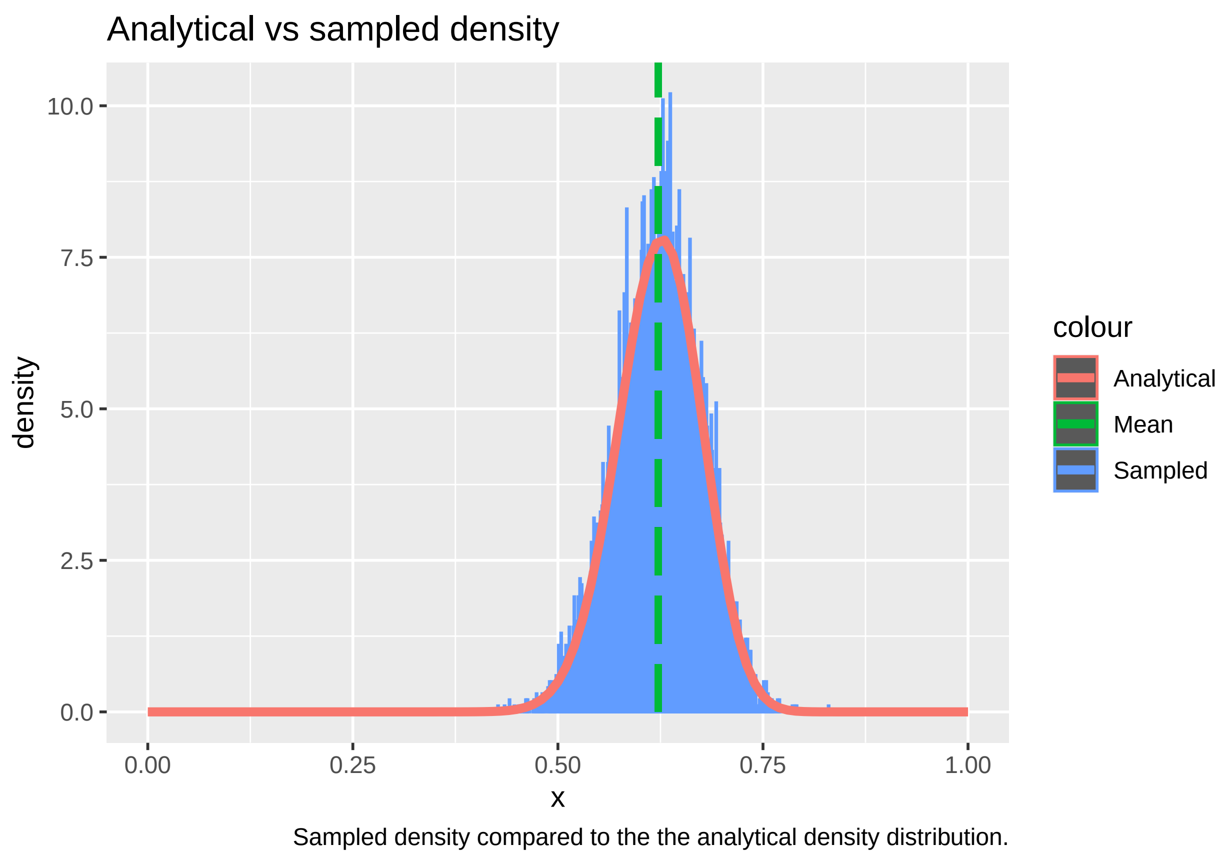
```
## Estimated mean:  0.622286 Theoretical mean:  0.3904824
```

```r
#Generate histogram

df = data.frame(theo = seq(0, 1, length.out = n), value = y$samples)

ggplot(df, aes(x = value)) + geom_histogram(aes(y = ..density.., colour = "Sampled"),
    binwidth = 0.001) + stat_function(fun = f_func_density, geom = "line",
    size = 1.6, aes(colour = "Analytical")) + ggtitle("Analytical vs sampled density") +
    xlim(0, 1) + geom_segment(x = sum(y$samples)/n, xend = sum(y$samples)/n,
    y = 0, yend = 100, aes(colour = "Mean"), size = 1.2, linetype = "longdash") +
    labs(x = "x", y = "density", caption = paste("Sampled density compared to the the analytical densit
```

```
## Warning: Removed 2 rows containing missing values (`geom_bar()`).
```



Part 3 - Average random numbers per sample

The expected number of random numbers the rejection sampling algorithm has to create to obtain one sample from $f(\theta|y)$ is $c \geq f(x)/g(x)$. c is chosen as small as possible for the algorithm to be as efficient as possible, resulting in $c = maxf(x)/g(x)$. $g(x) = U(0,1) = 1$, meaning $c = maxf(x)$.

```
cat("Average random numbers per sample: ",y$iterations/n)
```

```
## Average random numbers per sample:  7.7956
```

```
cat("Theoretical expected numbers per sample: ", target_c$objective/n_c)
```

```
## Theoretical expected numbers per sample:  7.799308
```

As one can see the average random numbers per sample and theoretical expected random numbers coincide.

Part 4 - Obtaining samples without uniform prior

In this task the posterior mean of θ will be estimated using $Beta(1,5)$ as prior. The posterior pdf is then

$$f(\theta|\mathbf{y}) \propto (2+\theta)^{y_1}(1-\theta)^{y_2+y_3}\theta^{y_4}\frac{\Gamma(1+5)}{\Gamma(1)\Gamma(5)}\theta^0(1-\theta)^4.$$

To improve efficiency, the old function could be used as the proposal density:

$$f_{old}(\theta|\mathbf{y}) \propto (2+\theta)^{y_1}(1-\theta)^{y_2+y_3}\theta^{y_4}$$

The importance sampling posterior mean is then

$$\hat{\mu}_{IS} = \frac{1}{n}\sum \theta_i w_i.$$

The weights will be

$$w_i = \frac{f_{new}(\theta_i|y)}{f_{old}(\theta_i|y)} \propto \pi_n(\theta_i)/\pi_o(\theta_i) = \frac{\Gamma(1+5)}{\Gamma(1)\Gamma(5)}(1-\theta_i)^4$$

```
# Target pdf, unscaled
posterior_target_unscaled = function(x) {
    return(f_func_density(x) * dbeta(x, 1, 5))
}

#Compute normalizing constant
n_c_posterior = integrate(posterior_target_unscaled, 0, 1)$value

# Normalized posterior target pdf
posterior_target_scaled = function(x) {
    return(posterior_target_unscaled(x)/n_c_posterior)
}

# Density: f_new(x) * x
posterior_target_mean_density = function(x) {
    return(x * posterior_target_scaled(x))
}

theoretical_mean = integrate(posterior_target_mean_density, 0, 1)$value

weights = posterior_target_scaled(y$samples)/f_func_density(y$samples)
posterior_mean = sum(y$samples * weights)/n
cat("Posterior mean estimate:", posterior_mean, "Analytical posterior mean:", theoretical_mean, "\n")
```

```
## Posterior mean estimate: 0.5987256 Analytical posterior mean: 0.5959316
```

The IS posterior mean estimate and analytical mean are close to each other, meaning the bias introdused by estimating the new mean is small enough.