Stochastic Simulation Coursework

Alexander Juxon Cobb, 01713494

02/12/2021

Implementation

The following notebook provides an implementation of the ratio-of-uniforms scheme to generate random variates from the probability density function $f_X(x)$, where

$$f_X(x) \propto \begin{cases} \frac{1}{x(1-x)} \mathrm{e}^{\frac{-1}{7}(-0.6+\log(\frac{x}{1-x}))^2}, & 0 < x < 1 \\ 0 \ , & \text{otherwise} \end{cases}$$

Furthermore, we will provide verification and analysis of our scheme via diagnostic plots and statistical tests, and implement a Monte Carlo procedure to estimate the normalising constant associated with $f_X(\cdot)$.

It should be noted that throughout this notebook $h(x) = \frac{1}{x(1-x)} \mathrm{e}^{\frac{-1}{7}(-0.6+\log(\frac{x}{1-x}))^2}$, and therefore $f = \frac{h}{\int(h)}$. In addition, it is assumed that the reader has familarity with the Stochastic Simulation lecture notes provided by Emma McCoy. As such, any results proved in Emma McCoy's lecture notes will be called upon without proof in the following notebook. Finally, I have chosen to implement the ratio-of-uniforms scheme, as opposed to rejection sampling, since rejection sampling was the subject of the formative coursework and therefore a more familar topic.

Visualisation of our function $h(x)$

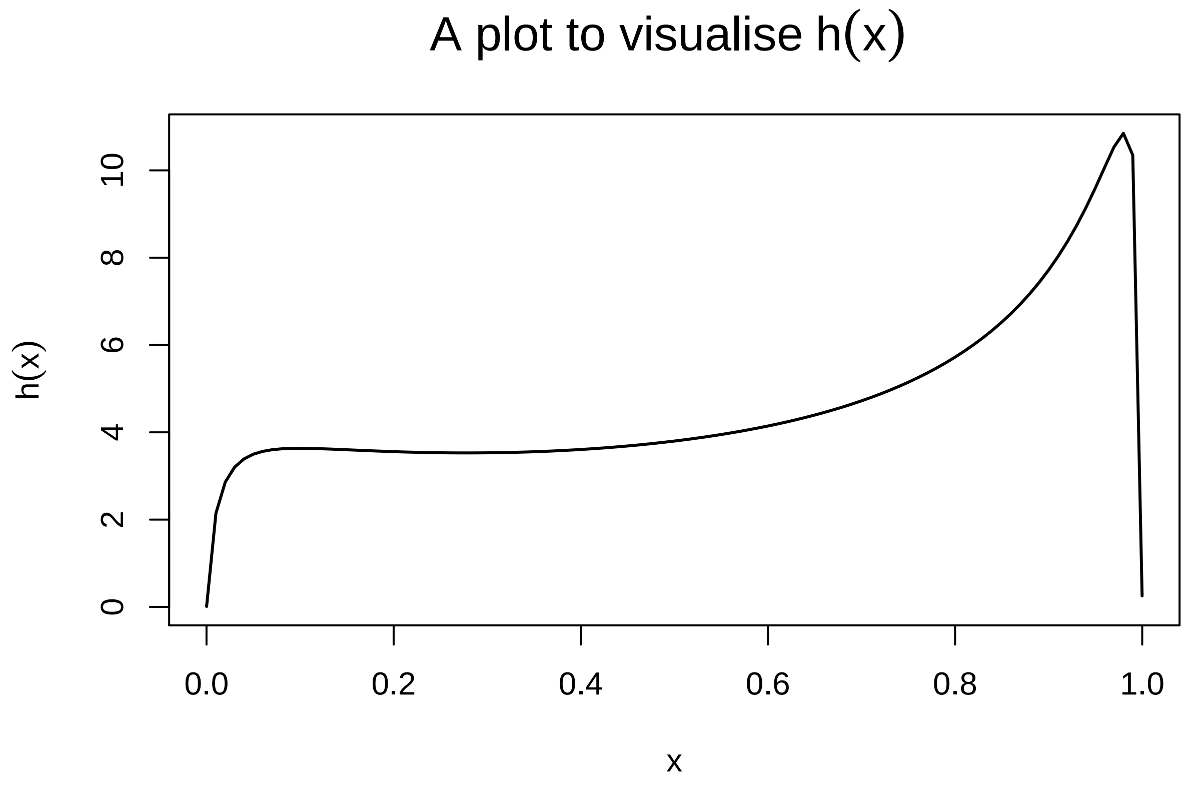
```r
max(primeFactors(1713494))
```

```
## [1] 953
```

From the params2021.pdf provided we see that our parameters are:

a = 0, b = 1, c = 7, d = -0.6

```r
h <- function(x) (exp((-1/7) * (-0.6 + log(x/(1 - x)))^2))/(x * (1 - x))  # This variable will be used

# Plot our function h(x) to gain a better understanding of what it looks like.
curve(h, from = 1e-04, to = 0.9999, xlab = "", ylab = "", main = "", cex = 4, lwd = 1.5)
title(main = expression("A plot to visualise" ~ h(x) ~ ""), xlab = "x", ylab = expression("" ~
    h(x) ~ ""), cex.main = 1.5)
```



```r
h_integral <- as.numeric(integrate(h, 0, 1)[1])  # Calculate the integral of h(x) over our domain x.
print(h_integral)
```

```
## [1] 4.689473
```

We use the built-in R function `integrate()` to calculate the integral of $h(x)$ for 2 main reasons:

1. To check the integral does indeed exist.
2. So that we have a numerical approximation for $\int h(x)dx$, which is used in calculating the acceptance probability of the algorithm and defining $f_X(x)$ in the 'Verification' section of this notebook.

Later in the notebook this value for the integral is verified via 3 methods of Monte Carlo integration and has also been cross-checked on WolphramAlpha to ensure its validity.

Generating from $f_X(x)$ using Ratio-of-Uniforms

The general ratio-of-uniforms algorithm to simulate a random variable X $\sim f_X(x)$ involves generating pairs of standard uniform random variables, manipulating these pairs into values U and V according to a pre-determined bounding rectangle and checking if the values U and V satisfy an acceptance condition (i.e. U and V are in the region C_h defined below). If U and V satisfy our acceptance conditon, $X = \frac{U}{V}$ is a random variable with probability density function $f_X(x)$.

General Ratio-of-Uniforms Algorithm

1. Find the bounding rectangle for C_h, where:

$$C_h = \left\{ (u, v) : 0 \leq u \leq \sqrt{h\left(\frac{u}{v}\right)} \right\}$$

We therefore find a, b and c such that:

$$a = \sup_x \sqrt{h(x)} \qquad b = \inf_{x \le 0} x\sqrt{h(x)} = 0 \qquad c = \sup_{x \ge 0} x\sqrt{h(x)}$$

as shown in lectures. It is a necessary condition that $h(x)$ and $x^2h(x)$ are bounded in the domain of x for such a rectangle to exist.

2. Generate $(U_1, U_2) \sim U(0,1)$.
3. Set $U = aU_1,\ V = b + (c-b)U_2$.
4. If $U \le \sqrt{h(\frac{V}{U})}$, set $X = \frac{V}{U}$. Otherwise, repeat the algorithm (i.e. GOTO 2).

General Ratio-of-Uniforms Algorithm Acceptance Probability

The theoretical acceptance probability of the algorithm is then given by

$$\text{Probability of accepting an X} = \frac{\text{Area of } C_h}{\text{Area of bounding rectangle}} = \frac{\frac{1}{2}\int h(x)\ dx}{a(c-b)}$$

Finding a, b and c

Using the limits presented in point 3. of the general ratio-of-uniforms algorithm, we can use the following R code to calculate a, b and c:

```r
{
    x_sqrt_h <- function(x) x * sqrt(h(x))
    x_squared_h <- function(x) x^2 * h(x)

    # Built-in R function that finds the maximum of a
    # function over a given domain.
    h_max <- optimise(h, c(0, 1), maximum = TRUE)[[2]]
    x_squared_h_max <- optimise(x_squared_h, c(0, 1), maximum = TRUE)[[2]]

    cat("max(h(x)) =", format(h_max), "\n")
    cat("max(x^2*h(x)) = ", format(x_squared_h_max), "\n", "\n")

    a <- sqrt(h_max)
    b <- 0  # Since h(x) = 0 for all x < 0 and h(0) = 0.
    c <- optimise(x_sqrt_h, c(0, 1), maximum = TRUE)[[2]]

    cat("a =", format(a), ", b =", format(b), "and c =", format(c))
}
```

```
## max(h(x)) = 10.85885
## max(x^2*h(x)) =  10.48557
##
## a = 3.295278 , b = 0 and c = 3.238143
```

Clearly $\max(h(x)) = 10.85885 < \infty$ and $\max(x^2h(x)) = 10.48557 < \infty$. Our bounding rectangle therefore exists.

In addition, we have a = 3.295278 , b = 0 and c = 3.238143.

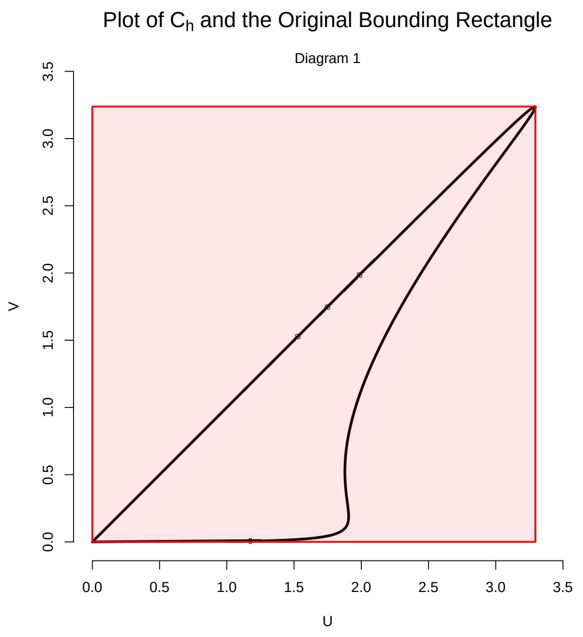
Improved Ratio-of-Uniforms Algorithm (Pre-test Squeezing and Reflection)

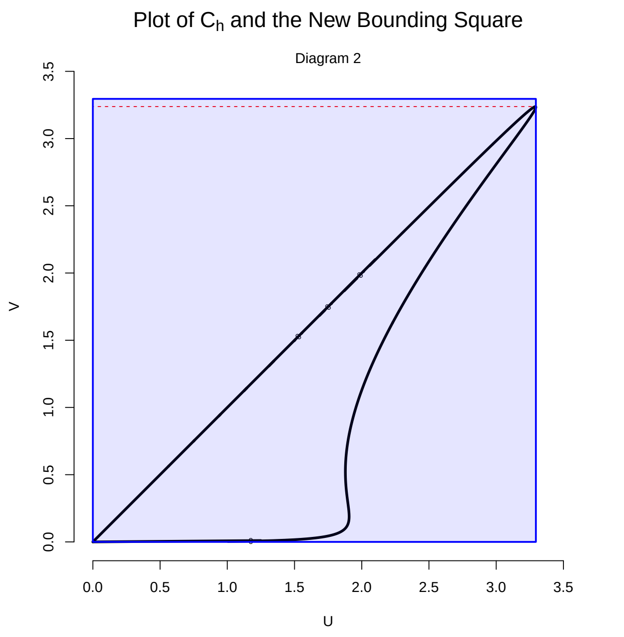
Having analysed our specific density and its corresponding C_h, it is apparent that we can perform a pre-test squeeze and implement a reflection step to reduce our computational complexity and increase our theoretical acceptance probability.

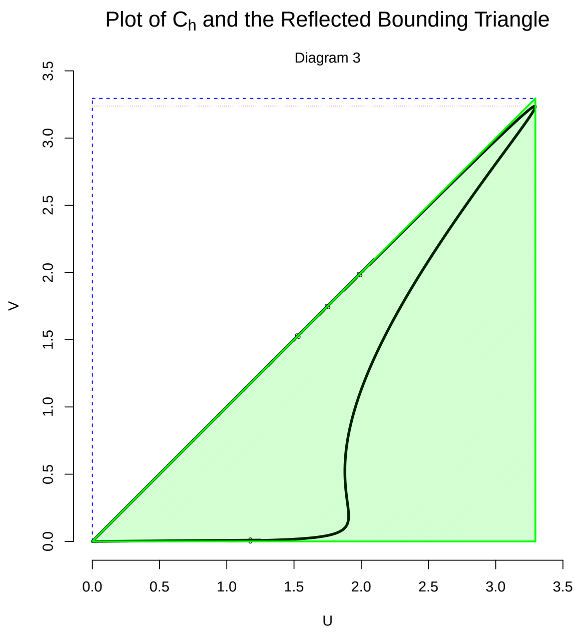
This improvement is best explained via the following 4 diagrams:

```
{
    my_fun <- function(x, y) {
        sqrt(h(y/x)) - x
    }  # A function to plot the region C_h.
    x <- seq(0, a, length = 1000)
    y <- seq(0, c, length = 1000)
    # Return a vector of the values of my_fun evaluated
    # over the region.
    z <- outer(x, y, my_fun)

    par(pty = "s", fig = c(0, 0.5, 0.5, 1), new = TRUE, mar = c(4,
        4, 4, 4))
    contour(x, y, z, level = 0, main = NULL, xlab = "U", ylab = "V",
        frame.plot = FALSE, lwd = 3, cex = 2, xlim = c(0, 3.5),
        ylim = c(0, 3.5))
    title <- as.list(expression(paste("Plot of ", C[h], " and the Original Bounding Rectangle"),
        "Diagram 1"))
    # Add the title to the plot.
    mtext(do.call(expression, title), line = c(1, -1), cex = c(1.5,
        1))
    # Additional segments added for visual enhancement:
    # since length(x) = length(y) = 1000, the contour does
    # not fully connect. To avoid increasing the knit time
    # of my markdown unnecessarily I have added these
    # additional visual segments as opposed to increasing
    # length(x) and length(y). Denoted 'Additional segment'
    # for the remainder of this code segment.
    segments(0, 0, 2.1, 2.1, lwd = 3)
    segments(0, 0, 1.25, 0.01, lwd = 3)  # Additional segment.
    # Original bounding rectangle for C_h.
    rect(0, b, a, c, border = "red", lwd = 2, col = rgb(1, 0,
        0, alpha = 0.1))

    par(pty = "s", fig = c(0.5, 1, 0.5, 1), new = TRUE, mar = c(4,
        4, 4, 4))
    contour(x, y, z, level = 0, main = NULL, xlab = "U", ylab = "V",
        frame.plot = FALSE, lwd = 3, cex = 2, xlim = c(0, 3.5),
        ylim = c(0, 3.5))
    title <- as.list(expression(paste("Plot of ", C[h], " and the New Bounding Square"),
        "Diagram 2"))
    mtext(do.call(expression, title), line = c(1, -1), cex = c(1.5,
        1))
    segments(0, 0, 2.1, 2.1, lwd = 3)  # Additional segment.
    segments(0, 0, 1.25, 0.01, lwd = 3)  # Additional segment.
    # Original bounding rectangle (from diagram 1).
    rect(0, b, a, c, border = "red", lwd = 1, lty = "dashed")
    # New bounding square.
```

```r
rect(0, b, a, a, border = "blue", lwd = 2, col = rgb(0, 0,
    1, alpha = 0.1))

par(pty = "s", fig = c(0, 0.5, 0, 0.5), new = TRUE, mar = c(4,
    4, 4, 4))
contour(x, y, z, level = 0, main = NULL, xlab = "U", ylab = "V",
    frame.plot = FALSE, lwd = 3, cex = 2, xlim = c(0, 3.5),
    ylim = c(0, 3.5))
title <- as.list(expression(paste("Plot of ", C[h], " and the Reflected Bounding Triangle"),
    "Diagram 3"))
mtext(do.call(expression, title), line = c(1, -1), cex = c(1.5,
    1))
# Original bounding rectangle (from diagram 1).
segments(0, c, c, c, col = "red", lwd = 0.5, lty = "dotted")
# Bounding square (from diagram 2).
segments(0, 0, 0, a, col = "blue", lwd = 1, lty = "dashed")
segments(0, a, a, a, col = "blue", lwd = 1, lty = "dashed")
segments(0, 0, 2.1, 2.1, col = "black", lwd = 3)  # Additional segment.
segments(0, 0, x1 = 1.25, y1 = 0.01, lwd = 3)  # Additional segment.
# New bounding triangle.
segments(0, 0, a, a, col = "green", lwd = 2)
segments(a, a, a, 0, col = "green", lwd = 2)
segments(0, 0, a, 0, col = "green", lwd = 2)

# 'Colour in' the triangle for a clearer visualisation.
polygon(c(0, a, a), c(0, a, 0), density = 1000, col = rgb(0,
    1, 0, alpha = 0.02))
par(pty = "s", fig = c(0.5, 1, 0, 0.5), new = TRUE, mar = c(4,
    4, 4, 4))
contour(x, y, z, level = 0, main = NULL, xlab = "U", ylab = "V",
    frame.plot = FALSE, lwd = 3, cex = 2, cex.main = 1.5,
    xlim = c(0, 3.5), ylim = c(0, 3.5))
title <- as.list(expression(paste("Plot of ", C[h], " and the Pre-testing Squeezed Bounding Triangl
    "Diagram 4"))  # Formatting our title for the plot.
mtext(do.call(expression, title), line = c(1, -1), cex = c(1.5,
    1))
# Original bounding rectangle (from diagram 1).
segments(0, c, c, c, col = "red", lwd = 0.5, lty = "dotted")
# Bounding square (from diagram 2).
segments(0, 0, 0, a, col = "blue", lwd = 1, lty = "dashed")
segments(0, a, a, a, col = "blue", lwd = 1, lty = "dashed")
segments(0, 0, 2.1, 2.1, col = "black", lwd = 3)  # Additional segment.
segments(0, 0, x1 = 1.25, y1 = 0.01, lwd = 3)  # Additional segment.
# Previous bounding triangle (from diagram 3).
segments(a, a, a, 0, col = "green", lwd = 1.5, lty = "dashed")
segments(1.9, 0, a, 0, col = "green", lwd = 1.5, lty = "dashed")
# New bounding triangle (pre-test squeezed).
segments(0, 0, a, a, col = "green", lwd = 2)
segments(0, 0, 1.9, 0, col = "green", lwd = 2)
segments(a, c, 1.9, 0, col = "green", lwd = 2)

polygon(c(0, a, a, 1.9), c(0, a, c, 0), density = 1000, col = rgb(0,
    1, 0, alpha = 0.02))
}
```







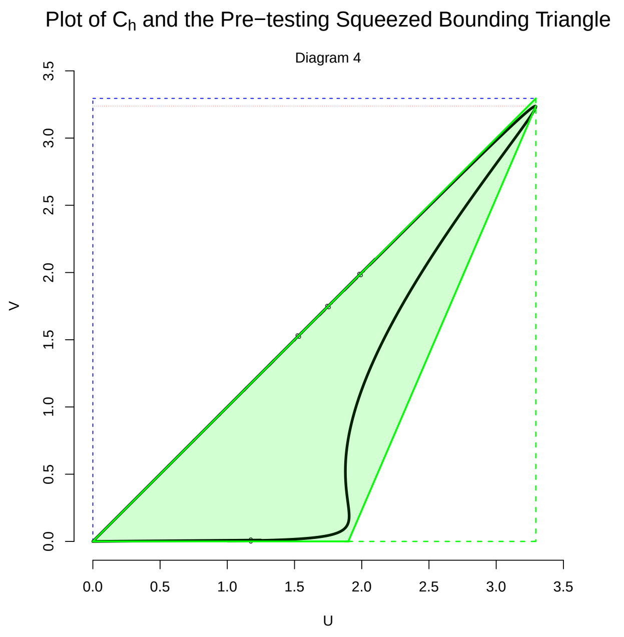


Diagram 1.

We see that our original bounding rectangle, as defined in the general ratio-of-uniforms algorithm, completely covers our region C_h. However, it is clear that, when randomly generating from this rectangle, we will have many values that are inside the rectangle but not inside C_h (because C_h covers less than a third of the bounding rectangle). This will make our acceptance probability very low (i.e. $\text{P}(\text{accept } X = \frac{U}{V}) < \frac{1}{3}$) and is unnecessarily computationally complex (since we will calculate $\sqrt{h(\frac{V}{U})}$ for points we can easily check are not inside our region C_h with a simpler function).

Diagram 2.

We extend our bounding rectangle to a bounding square, with sides of length a. Whilst this increases the area of our bounding region, temporarily making our algorithm less efficient, it is necessary to allow us to implement our reflection step correctly. If we were to bisect our rectangle diagonally and reflect samples in the upper triangle along this line, we would skew our samples since the areas would not perfectly overlap (and therefore generating a sample within some regions would have 0 probability and other regions would have double probability, resulting in a non-uniform sample over the triangle). If this is unclear, please read on to diagram 3 where the reflection step is discussed in more detail. In extending our bounding rectangle to this bounding square we now have $V = aU_2$ (whereas before we had $V = b + (c-b)U_2 = cU_2$). This is to ensure we still sample uniformly over the region.

Diagram 3.

We know that if we bisect our new bounding square diagonally (i.e. along the line $V = U$) and reflect the upper triangle onto the lower triangle along this line (see the visual change from diagram 2 to 3) we will have two perfectly overlapping regions. We know that this is a valid operation since $V > U \implies \frac{V}{U} > 1 \implies h(\frac{V}{U})$ is undefined. Hence, it is clear that our region C_h is below the line $V = U$ and therefore our reflected triangle will still bound the region C_h. As such, any samples generated in the upper triangle of our bounding square can be reflected along this line (by simply swapping the the random variables U and V, where $U = aU_1$ and $V = aU_2$) leading to a uniform sample over the bounding triangle and therefore doubling our acceptance probability.

Diagram 4.

We now perform a pre-testing squeeze to remove samples in the lower region of our triangle (see the visual change from diagram 3 to 4) before we implement our general rejection step (i.e calculating $\sqrt{h(\frac{V}{U})}$). We do this since any calculation involving $\sqrt{\cdot}$, $log(\cdot)$, $\exp(\cdot)$ etc. is very computationally complex and we can quickly reject samples in this lower region of the triangle according to the much simpler and computationally efficient linear inequality $V < \ U(\frac{c}{a-1.9}) + \frac{1.9c}{1.9-a}$ (which was found via trial and error in R). In reducing the number of unnecessary calculations involving $\sqrt{h(\frac{V}{U})}$ we drastically reduce the compuational time of our code.

Improved Ratio-of-Uniforms Algorithm (for our h(x))

1. Find the bounding rectangle for C_h, where:

$$C_h = \left\{(u,v) : 0 \leq u \leq \sqrt{h\Big(\frac{u}{v}\Big)}\right\}$$

   We therefore find a, b and c such that:

$$a = \sup_x \sqrt{h(x)} \qquad b = \inf_{x\leq 0} x\sqrt{h(x)} = 0 \qquad c = \sup_{x\geq 0} x\sqrt{h(x)}$$

   as shown in lectures. It is a necessary condition that $h(x)$ and $x^2h(x)$ are bounded in the domain of x for such a rectangle to exist.

2. Find the bounding square for C_h. We therefore have a square with vertices (0, 0), (a, 0), (a, a), (0, a). This is equivalent to setting c = a.

3. Generate $(U_1, U_2) \sim U(0,1)$.

4. Set $U = aU_1,\ V = aU_2$.

5. Reflect along the line $V = U$ (conventionally the line $y = x$). This is equivalent to: $V > U \implies$ swap U and V.

6. Pre-test squeeze and remove values below the line $V = U(\frac{c}{a-1.9}) + \frac{1.9c}{1.9-a}$. This is equivalent to: $V < U(\frac{c}{a-1.9}) + \frac{1.9c}{1.9-a} \implies$ repeat the algorithm (i.e. GOTO 3).

7. If $U \leq \sqrt{h(\frac{V}{U})}$, set $X = \frac{V}{U}$. Otherwise, repeat the algorithm (i.e. GOTO 3).

Improved Algorithm Acceptance Probability

For our function $h(x)$ and the general ratio-of-uniforms acceptance probability, as defined above, we have:

$$\text{Probability of accepting an X} = \frac{\frac{1}{2}\int h(x)\ dx}{a(c-b)} \approx \frac{\frac{1}{2} \times 4.689}{3.295(3.238-0)} \approx 0.219$$

However, this probability can be increased via our improved ratio-of-uniforms algorithm:

$$\text{Probability of accepting an X} = \frac{\frac{1}{2}\int h(x)dx}{\frac{1}{2}a^2} \approx \frac{\frac{1}{2} \times 4.689}{\frac{1}{2} \times 3.295^2} \approx 0.432$$

Where $\frac{1}{2}a^2$ is the area of our green region in diagram 3 (i.e. our reflected triangle, without the pre-testing squeeze). We calculate our acceptance probability using the region in diagram 3 (as opposed to the smaller region in diagram 4) because the samples removed via our pre-testng squeeze are still ‘lost’. That is, we do not accept any samples removed via our pre-testing squeeze.

It is clear that $\frac{0.432}{0.219} \approx 2$, and therefore our improved ratio-of-uniforms algorithm approximately doubles the acceptance probability.

Implementation of our Improved Rejection Algorithm in R

In the following code, familiarity with the above ‘Improved Ratio-of-Uniforms Algorithm’ is assumed.

```r
sampling_function <- function(n) {
    start_time <- Sys.time()  # Used to initialise the run-time of the function.

    # Initialise the theoretical acceptance probability, as
    # discussed above.
    t_a_p <- ((1/2) * h_integral)/((1/2) * a^2)

    # Expected number of generations needed, given this
    # acceptance probability. This improves the efficiency.
    expected_n <- ceiling(n * 1/t_a_p)  # We note that this will be greater than n.

    # Intialising the count and total.
    count <- 0
    total <- 0

    # Vector to store the random generations from f.
    x <- vector("numeric")

    # Loop until we have generated n times from f.
    while (count < n) {
        u <- a * runif(expected_n)
        # Normally, we would set: v <- b + (c - b) *
        # runif(expected_n).  However, since we want to
        # extend our bounding rectangle to a square (so
        # that we can reflect value above V = U), we set:
        v <- a * runif(expected_n)

        # Implement the reflection step: if U < V, we will
        # swap the values for U and V (i.e. let V -> U and
        # U -> V). This mimics a reflection along the line
        # line V = U.
        for (i in 1:length(u)) {
            if (u[i] < v[i]) {
                var_u <- u[i]
                var_v <- v[i]
                u[i] <- var_v
                v[i] <- var_u
            }
        }

        # Implement the squeeze to reduce the function's
        # compuational complexity and run-time.  We note
        # that this is performed after the reflection step
        # to encompass the reflected sample points too.
        dummy_u <- u[v > u * (c/(a - 1.9)) + (1.9 * c)/(1.9 -
            a)]
        dummy_v <- v[v > u * (c/(a - 1.9)) + (1.9 * c)/(1.9 -
            a)]
        u <- dummy_u
        v <- dummy_v

        # Rejection step.
        rejection_quanitity <- sqrt(h(v/u))  # Rejection value is specific to each (U,V) pair.
        x <- c(x, v[u < rejection_quanitity]/u[u < rejection_quanitity])
        count <- length(x)
        total <- total + expected_n
    }

    end_time <- Sys.time()  # Used to capture the finish time of the function.

    # Print useful contextual information for each sample.
    cat("For a sample of size", format(n), "and the improved ratio-of-uniforms algorithm: \n")
    cat("\t Theoretical acceptance probability = ", format(round(t_a_p,
        4)), "\n")
    cat("\t Actual acceptance probability = ", format(round(count/total,
        4)), "\n")
    cat("\t Computational time to run code = ", format(round(end_time -
        start_time, 4)), "\n")

    # Index and return the first n random quantities.
    return(x[1:n])
}
```

Implementation of the General Ratio-of-Uniforms Algorithm (For Comparison Purposes Only)

```r
sampling_function_general <- function(n) {
    start_time <- Sys.time()
    t_a_p <- ((1/2) * h_integral)/(a * (c - b))
    expected_n <- ceiling(n * 1/t_a_p)
    count <- 0
    total <- 0
    x <- vector("numeric")
    while (count < n) {
        u <- a * runif(expected_n)
        v <- b + (c - b) * runif(expected_n)
        rejection_quanitity <- sqrt(h(v/u))
        x <- c(x, v[u < rejection_quanitity]/u[u < rejection_quanitity])
        x <- na.omit(x)  # Since h(v/u) is not defined for v/u > 1.
        count <- length(x)
        total <- total + expected_n
    }
    end_time <- Sys.time()
    cat("For a sample of size", format(n), "and the generic ratio-of-uniforms algorithm: \n")
    cat("\t Theoretical acceptance probability = ", format(round(t_a_p,
        4)), "\n")
    cat("\t Actual acceptance probability = ", format(round(count/total,
        4)), "\n")
    cat("\t Computational time to run code = ", format(round(end_time -
        start_time, 4)), "\n")
    return(x[1:n])
}
```

Explore the Differences Between our Two Algorithms

We will work with a single, large sample generated from each algorithm (general and improved). This is done to explicitly show our improved algorithm is indeed an improvement whilst remaining concise.

```r
comparison_rou_sample <- sampling_function(10000)
```

```
## For a sample of size 10000 and the improved ratio-of-uniforms algorithm:
##   Theoretical acceptance probability =  0.4319
##   Actual acceptance probability =  0.435
##   Computational time to run code =  0.0225 secs
```

```r
comparison_general_sample <- sampling_function_general(10000)
```

```
## For a sample of size 10000 and the generic ratio-of-uniforms algorithm:
##   Theoretical acceptance probability =  0.2197
##   Actual acceptance probability =  0.22
##   Computational time to run code =  0.0199 secs
```

From the above data, it is clear that our improved ratio-of-uniforms algorithm is computationally more efficient and has a higher acceptance probability. Specifically, we see that our acceptance probability is approximately doubled and (as expected, therefore) our computational time approximately halved for the improved ratio-of-uniforms algorithm and a sample of 10,000. However, it should be noted that both algorithms are $O(n)$, since we know that $O(\alpha n) = O(n)\ \forall \alpha > 0$ by the Big O positive scale-invariance property. In general, our improved ratio-of-uniforms algorithm is $O(\frac{n}{\text{acceptance probability}}) \approx O(2.32n) = O(n)$

Verification

We present multiple diagnostic plots and statistical tests to verify that our function `sampling_function(n)` does indeed generate random samples from our distribution $f_X(x)$. We note that the remainder of the notebook is only concerned with the improved ratio-of-uniforms algorithm.

```
{
    # Intialise some samples to work with throughout the
    # verification process.
    sample_100 <- sampling_function(100)
    sample_1000 <- sampling_function(1000)
    sample_10000 <- sampling_function(10000)
}
```

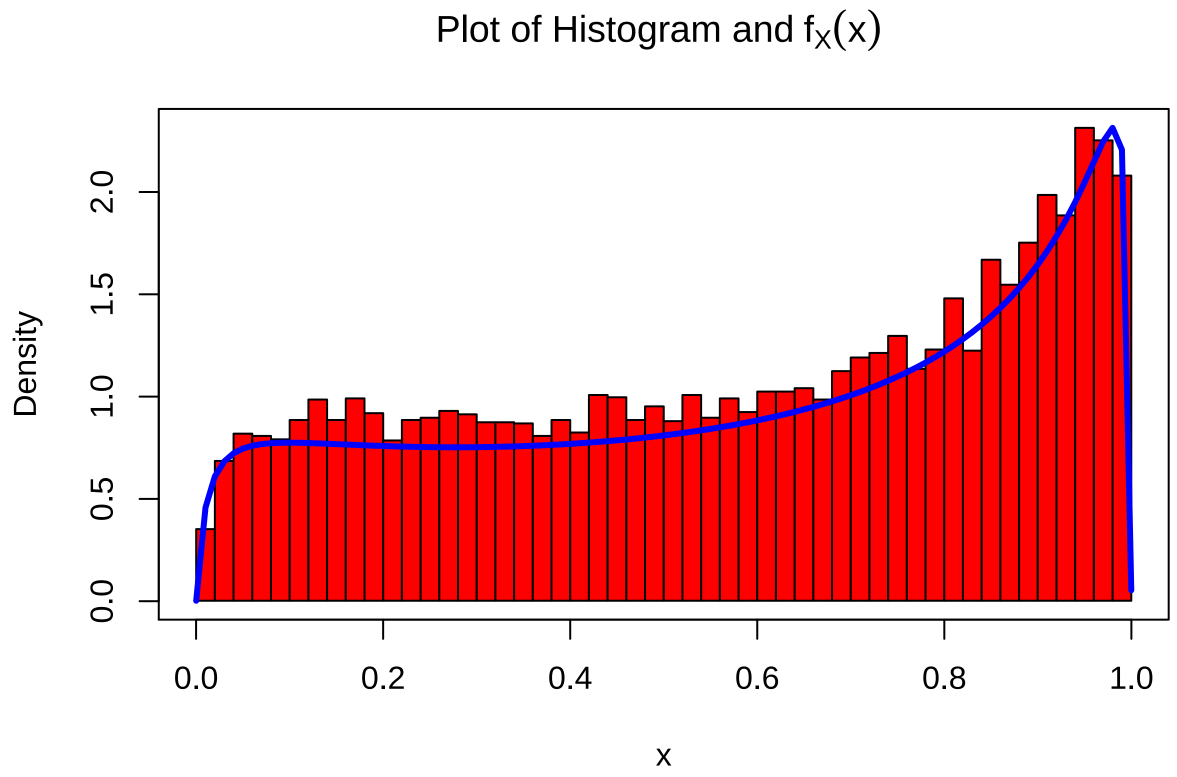
```
## For a sample of size 100 and the improved ratio-of-uniforms algorithm:
##   Theoretical acceptance probability =  0.4319
##   Actual acceptance probability =  0.431
##   Computational time to run code =  4e-04 secs
## For a sample of size 1000 and the improved ratio-of-uniforms algorithm:
##   Theoretical acceptance probability =  0.4319
##   Actual acceptance probability =  0.4344
##   Computational time to run code =  8e-04 secs
## For a sample of size 10000 and the improved ratio-of-uniforms algorithm:
##   Theoretical acceptance probability =  0.4319
##   Actual acceptance probability =  0.4275
##   Computational time to run code =  0.0181 secs
```

We also intialise the probability density function $f_X(x)$ since the true probability density function will be useful for our visualisations. We note that this is a valid operation due to the properties of $h(x)$ and $\int h(x)dx$ discussed above.

```
f <- function(x) (exp((-1/7) * (-0.6 + log(x/(1 - x)))^2))/(x * (1 - x))/h_integral  # Scaling our func
```

Histogram

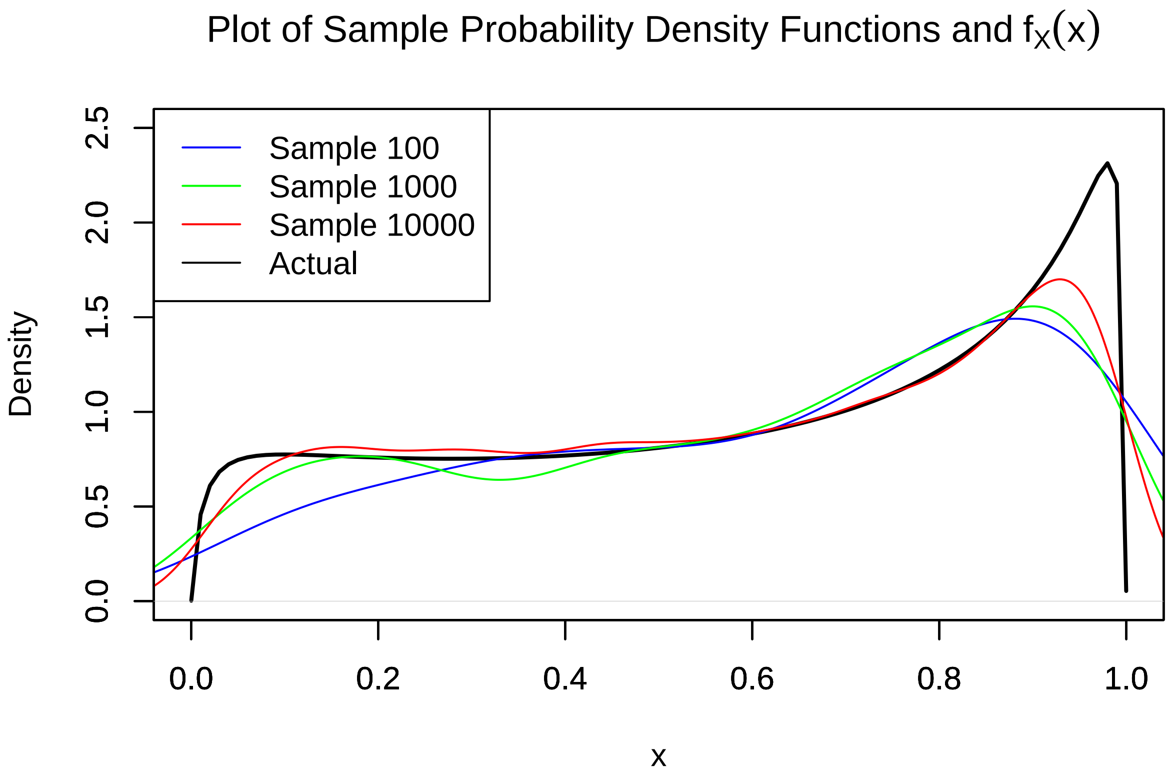
```
{
    # We use the largest sample (sample_10000) since this
    # is the most consistent representation of our
    # sampling_function (due to the Law of Large Numbers).
    hist(sample_10000, breaks = 50, freq = FALSE, col = "red",
        xlab = "", ylab = "", main = "", axes = FALSE)  # freq=FALSE gives a density histogram
    par(new = TRUE)
    curve(f, from = 1e-04, to = 0.9999, xlab = "x", ylab = "Density",
        col = "blue", main = expression("Plot of Histogram and" ~
            f[X](x) ~ ""), lwd = 3)
}
```



There is a clear similarity between the histogram of our size-10000 sample and the theoretical probability density function $f_X(x)$. This is evidence for us having implemented our rejection-of-uniforms scheme correctly.

Sample Probability Density Functions

```
{
    curve(f, from = 1e-04, to = 0.9999, xlim = c(0, 1), ylim = c(0,
        2.5), xlab = "x", ylab = "Density", main = expression("Plot of Sample Probability Density Functi
        f[X](x) ~ ""), lwd = 2)
    par(new = TRUE)
    plot(density(sample_100), xlim = c(0, 1), ylim = c(0, 2.5),
        col = "blue", ann = FALSE)
    par(new = TRUE)
    plot(density(sample_1000), xlim = c(0, 1), ylim = c(0, 2.5),
        col = "green", ann = FALSE)
    par(new = TRUE)
    plot(density(sample_10000), xlim = c(0, 1), ylim = c(0, 2.5),
        col = "red", ann = FALSE)
    legend("topleft", legend = c("Sample 100", "Sample 1000",
        "Sample 10000", "Actual"), col = c("blue", "green", "red",
        "black"), lty = 1)
}
```

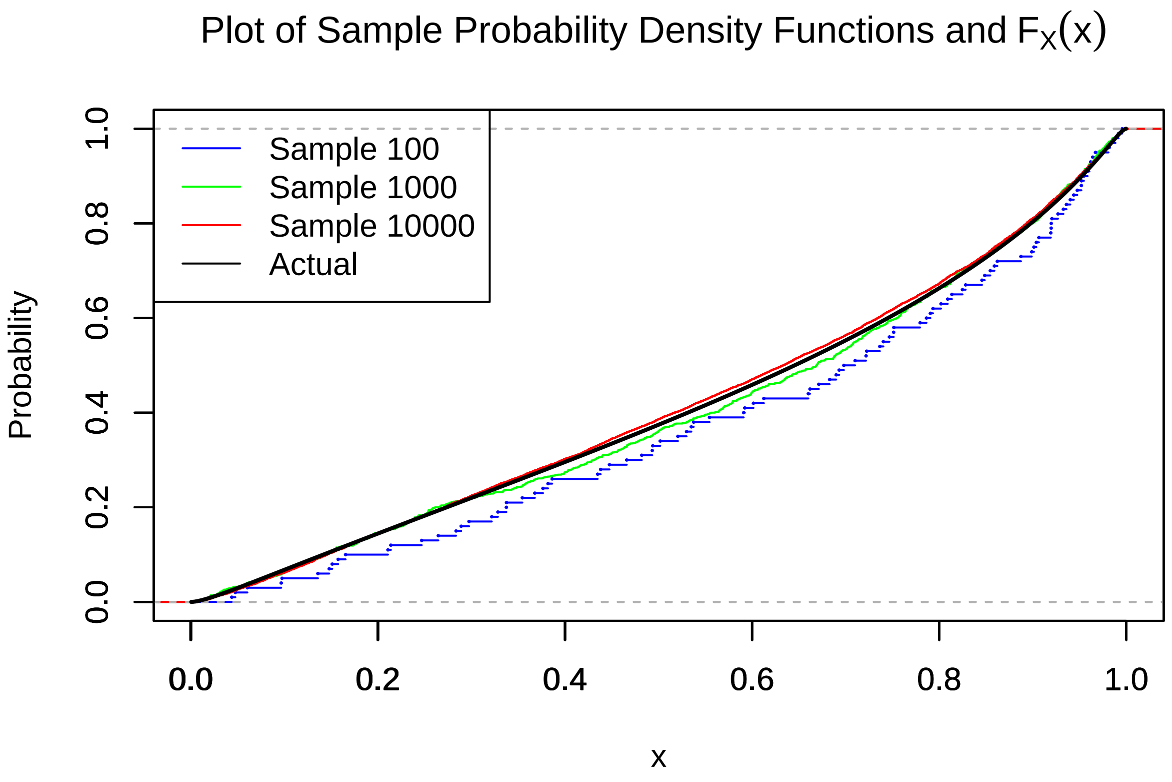


Again, we see that there is a clear similarity between our sampled data and $f_X(x)$. It is interesting to visualise the improvement in similarity as the sample size increases, which can be attributed to the Law of Large Numbers. We would therefore expect a very accurate representation of $f_X(x)$ for a large sample size (e.g. $n > 10^6$).

Sample Cumulative Distribution Functions

Since we cannot solve $\int f_X(x)\,dx$ analytically, we obtain an accurate function approximation via the built-in R function `cumsum()`. Whilst this is not a perfect solution, our small dx value (0.0001) ensures it has the required accuracy for the purposes of this notebook.

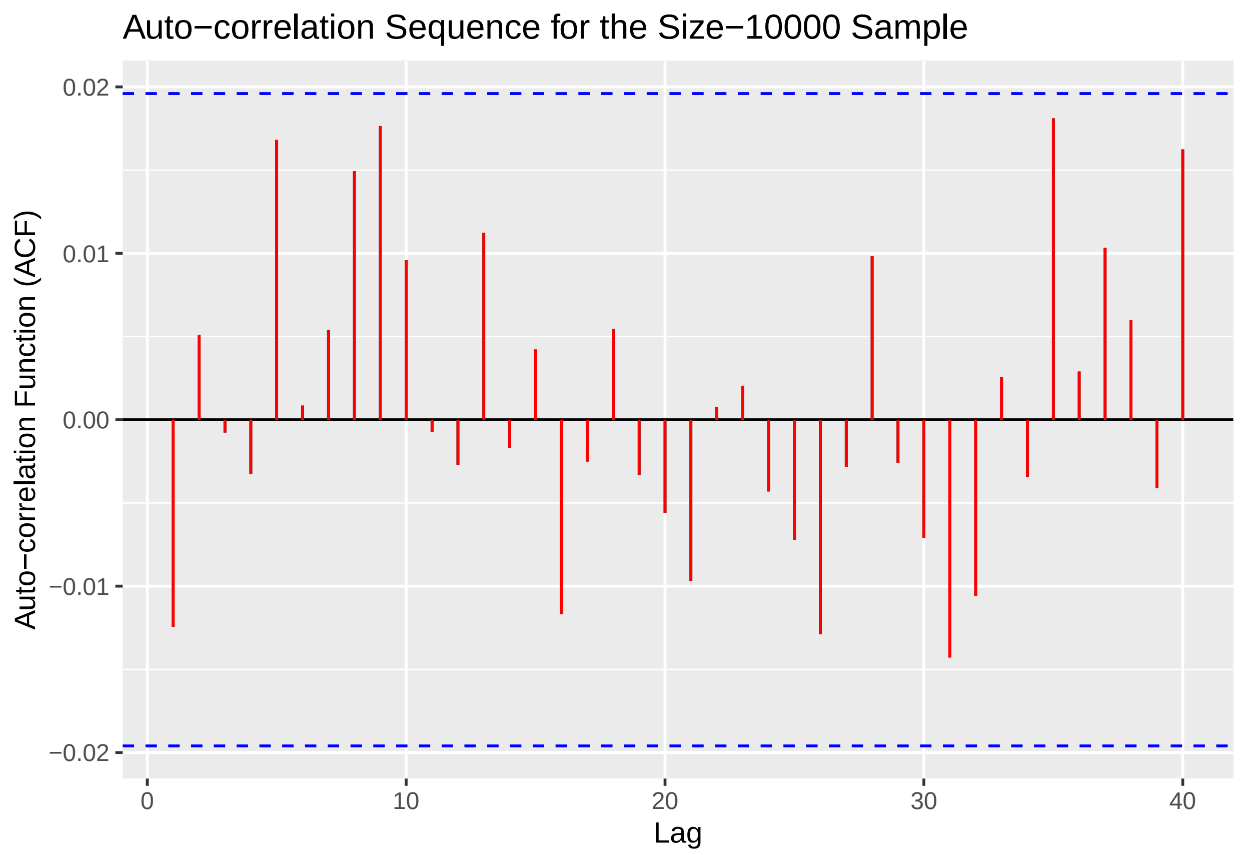
```
{
    fx <- Vectorize(f)  # Vectorize f to apply the cumsum() function below.
    dx <- 1e-04
    x_vals <- seq(0.001, 1, by = dx)

    plot(ecdf(sample_100), xlim = c(0, 1), ylim = c(0, 1), ylab = "",
        xlab = "", main = "", col = "blue", cex = 0.1)
    par(new = TRUE)
    plot(ecdf(sample_1000), xlim = c(0, 1), ylim = c(0, 1), ylab = "",
        xlab = "", main = "", col = "green")
    par(new = TRUE)
    plot(ecdf(sample_10000), xlim = c(0, 1), ylim = c(0, 1),
        ylab = "", xlab = "", main = "", col = "red")
    par(new = TRUE)
    plot(x_vals, cumsum(fx(x_vals) * dx), type = "l", xlab = "x",
        ylab = "Probability", main = expression("Plot of Sample Probability Density Functions and" ~
            F[X](x) ~ ""), col = "black", lwd = 2)
    legend("topleft", legend = c("Sample 100", "Sample 1000",
        "Sample 10000", "Actual"), col = c("blue", "green", "red",
        "black"), lty = 1)
}
```



As in the previous plot, we can see that there is a clear similarity between the sample cumulative distribution functions and $F_X(x)$ and that this similarity improves as the sample size increases. Since every probability density function defined on the real numbers (as ours is) is uniquely indentified by a cumulative distribution function, the above plot is strong evidence that we have implemented our ratio-of-uniforms algorithm correctly.

Auto-correlation Sequence

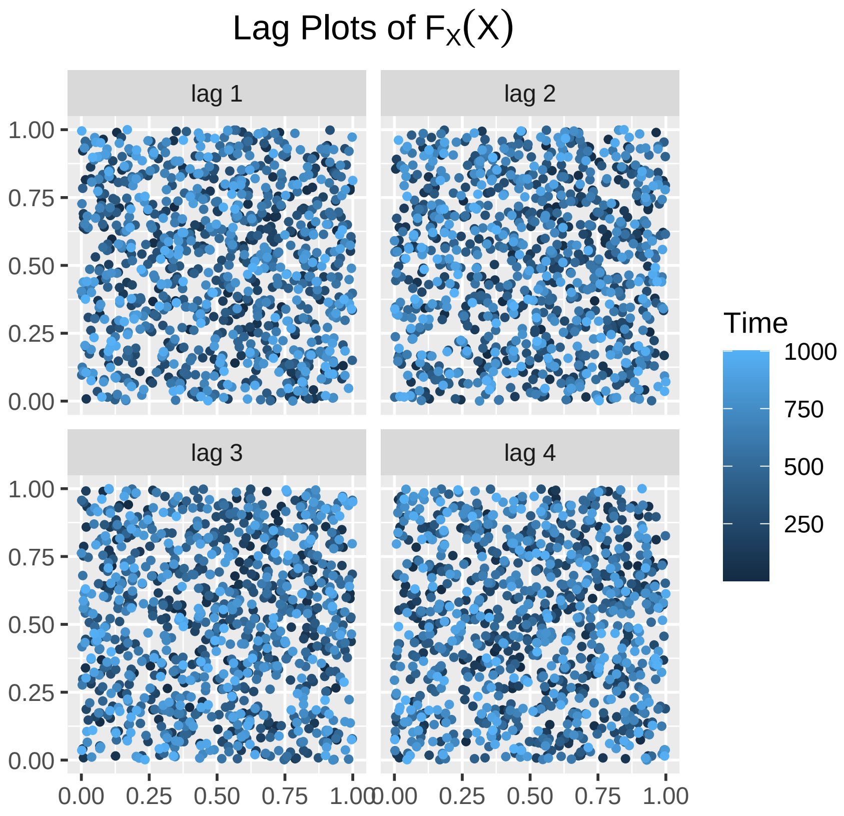
```
ggAcf(sample_10000) + theme_update(plot.title = element_text(hjust = 0.5)) +
    labs(title = "Auto-correlation Sequence for the Size-10000 Sample",
        x = "Lag", y = "Auto-correlation Function (ACF)") + geom_segment(color = "red")
```



The auto-correlation plot shows no signs of significant correlation at any non-zero lag and no discernable pattern. This provides evidence that the samples in our data is generated independantly and randomly.

Lag Plots

```r
# Create a function that can be evaluated for values of x
# and is equivalent to the cdf of f.
f_integral <- function(x) as.numeric(integrate(f, 0, x)[1])

integral_vector <- c()  # Vector of the cdf evaluated at each sample in our data.

# We use sample_1000 to make visualisation easier.
for (i in sample_1000) {
    integral_vector <- c(integral_vector, f_integral(i))  # Where f_integral(i) = P(X<i).
}


gglagplot(integral_vector, lags = 4, diag = FALSE, do.lines = FALSE) +
    labs(title = expression("Lag Plots of" ~ F[X](X)))
```



For any valid probability density function $f_X(x)$, $X \sim f_X(x) \implies F_X(x) \sim U(0,1)$, and hence the lag scatter plots of $F_X(x_i)$ should show a random scatter. Visually, the plots above do show a random scatter, which further supports that our data is generated independantly and randomly from $f_X(\cdot)$.

The diagnostic plots above suggest that our data is generated randomly and independantly from the probability distribution function $f_X(x)$. We provide further evidence for this via the following statistical tests:

Kolmogorov-Smirvov Tests

The Kolmogorov-Smirnov test (K-S test) is a non-parametric test of the equality of one-dimensional probability distributions, that can be used to compare a sample with a probability distribution (one-sample) or to compare two samples (two-sample). The test provides insight into whether a sample has been drawn from the provided probability distribution or that two samples have been drawn from the same, unknown probability distribution.

In the K-S test, the test statistic, D, is given by:

$$D = \sup_x \left| F_{X_n}(x) - F_X(x) \right|$$

This test statistic is then compared to the critical values of the Kolmogorov distribution to determine the p-value of the test, which maintains its usual interpretation.

Following on from the diagnostic plots of $F_X(x)$ and its expected distribution, we implement a two-sample Kolmogorov-Smirvov test:

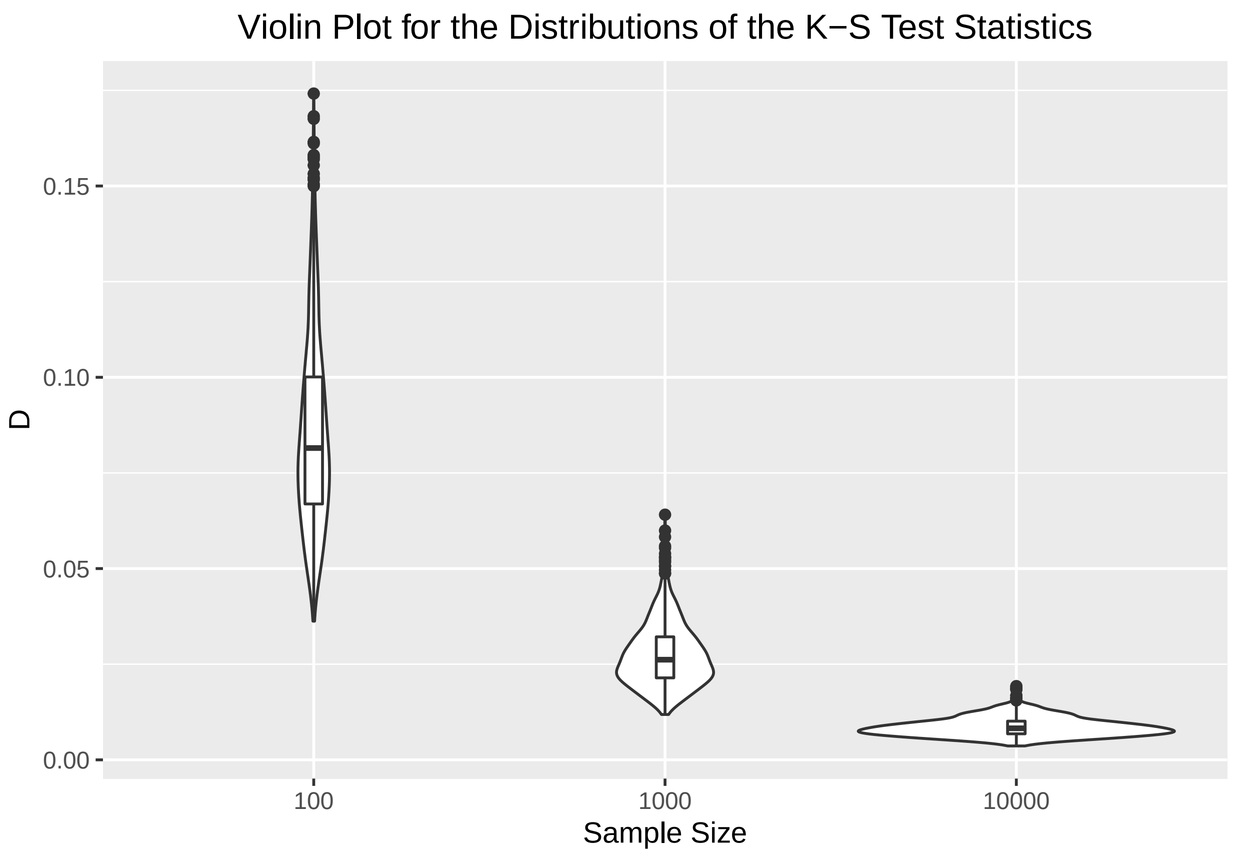
```
runif_sample_1000 <- runif(1000)
unif_ks_test <- ks.test(integral_vector, runif_sample_1000)
unif_ks_test
```

```
##
##  Two-sample Kolmogorov-Smirnov test
##
## data:  integral_vector and runif_sample_1000
## D = 0.044, p-value = 0.2877
## alternative hypothesis: two-sided
```

A p-value of 0.2877 suggests that the two samples (`integral_vector` and `runif_sample_1000`) have been drawn from the same distribution. Trivially, this implies have both been drawn from a standard uniform distribution, as we would expect, since we know that $F_X(x) \sim U(0,1)$

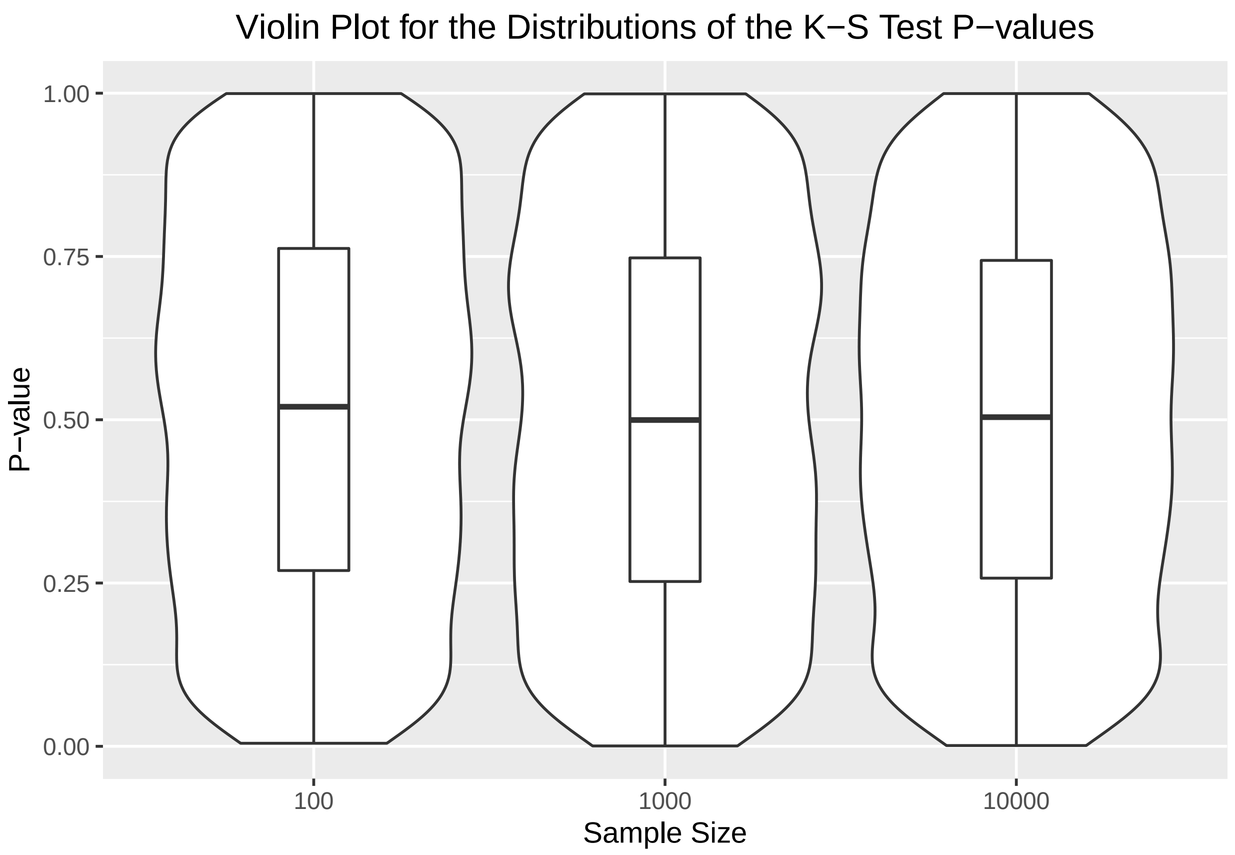
```r
n_vals = c(100, 1000, 10000)
m = 1000

# Approximate function for the cdf, using fx and x_vals
# from before.
approx_CDF <- approxfun(x_vals, cumsum(fx(x_vals) * dx))

ks_test_f <- function(x) {
    kstestx = ks.test(x, approx_CDF)
    return(c(kstestx$p, kstestx$statistic))
}
ks_results = data.frame()

for (n in 1:3) {
    n1 = n_vals[n]
    x <- matrix(0, nrow = n1, ncol = m)
    x1 = sampling_function(n1 * m)
    for (i in 1:m) {
        # Split into matrix to use the apply function.
        x[, i] <- x1[((i - 1) * n1 + 1):(i * n1)]
    }
    ks_testx = apply(x, 2, ks_test_f)
    ks_results = rbind(ks_results, data.frame(p_value = ks_testx[1,
        ], D = ks_testx[2, ], N = rep(n1, m)))
}

ks_results = na.omit(ks_results)
```

```r
ggplot(ks_results, aes(factor(N), D)) + geom_violin() + geom_boxplot(width = 0.05) +
    labs(title = "Violin Plot for the Distributions of the K-S Test Statistics",
        x = "Sample Size", y = "D")
```



Given the definition of the test statistic, $D = \sup_x \left|F_{X_n}(x) - F_X(x)\right|$, we would expect $D \xrightarrow{n\to\infty} 0$. The above violin plot implies this and suggests that $Var(D) \to 0$ as well (since the probability density function of D becomes thinner as $n \to \infty$).

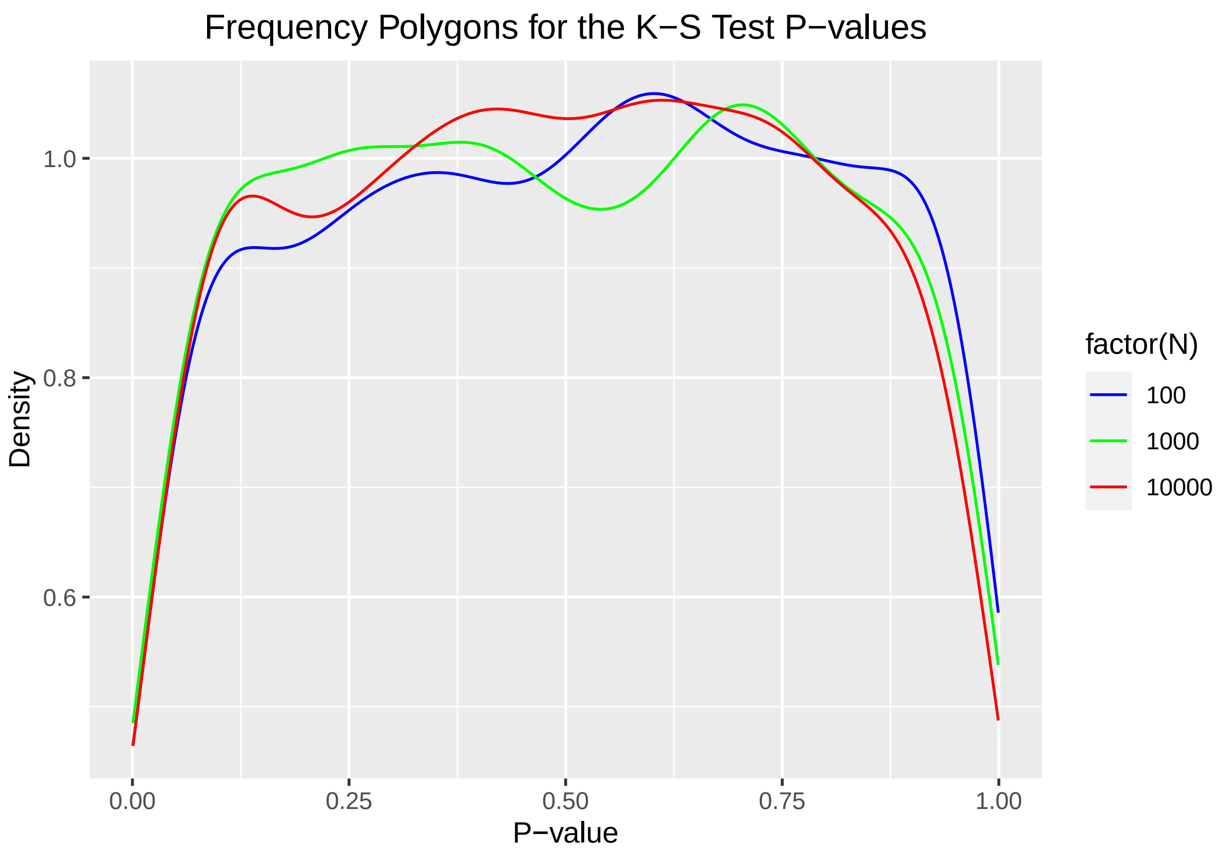
```
ggplot(ks_results, aes(factor(N), p_value)) + geom_violin() +
    geom_boxplot(width = 0.2) + labs(title = "Violin Plot for the Distributions of the K-S Test P-value
    x = "Sample Size", y = "P-value")
```



There are two key takeaways from the above plot:

1. The mean of the p-value is consistently ≈ 0.5. This implies that we do not have sufficient evidence to reject the null hypothesis (that our sample has been drawn from $F_X(x)$) at any reasonable significance level.
2. The violin plots are of a consistent width. This implies that the p-values are distributed uniformly, which we will visualise in the following plot.

```
ggplot(ks_results, aes(p_value, colour = factor(N))) + scale_colour_manual(values = c("blue",
    "green", "red")) + geom_freqpoly(breaks = seq(0, 1, 0.1),
    stat = "density") + labs(title = "Frequency Polygons for the K-S Test P-values",
    x = "P-value", y = "Density")
```



We confirm that the distribution of p-values is roughly uniform, as expected under the null hypothesis. For larger sample sizes we would expect the p-value's density function to approach a horizontal line at density = 1, and this can be easily verified with slight modifications to our code.

```
{
    ks_table <- ks_results %>%
        group_by(N) %>%
        summarise(`Mean P-value` = round(mean(p_value), digits = 3),
            `Standard Deviation (P-Value)` = round(sqrt(var(p_value)),
                3), `Mean D` = round(mean(D), digits = 3), `Standard Deviation (D)` = round(sqrt(var(D)
                3))

    kable(ks_table, align = "c")
}
```

N	Mean P-value	Standard Deviation (P-Value)	Mean D	Standard Deviation (D)
100	0.515	0.289	0.085	0.025
1000	0.503	0.289	0.027	0.008
10000	0.503	0.284	0.009	0.003

This table confirms that, as the sample size increases, the expectation and standard deviation of the p-value remains constant (as we would expect from a uniform distribution) and the expectation and standard deviation of D tends to 0 (as we would expect, since, by the Law of Large Numbers, a larger sample provides a more accurate representation of the distribution that it is sampled from). Since $D \to 0$, we can be confident that our data is sampled from $F_X(x)$.

Student's T-test

The t-test is a statistical hypothesis test to analyse how significant the difference is between the mean of a (normally distributed) sample and the theoretical expectation of the distribution it has supposedly been drawn from. Whilst $f_X(x)$ is clearly not normally distributed, the Central Limit Theorem assures that when independent random variables are summed up, their normalized sum ($\frac{1}{n}\sum_{i=1}^{n} X_i$) tends towards a normal distribution regardless of the initial distribution of the X_i. This asymptotic normality of the mean and our relatively large sample sizes (i.e. >50) justifies the use of Student's t-test.

```r
# Create a function to round all numeric values in a data frame.
round_df <- function(x, digits) {
    numeric_columns <- sapply(x, mode) == "numeric"
    x[numeric_columns] <- round(x[numeric_columns], digits)
    x
}
```

```r
# Find the theoretical mean of the distribution f.
find_mean <- function(f, ..., from = 0, to = 1) {
    integrate(function(x) x * f(x, ...), from, to)
}
theoretical_mean <- find_mean(f)[[1]]
cat("Theoretical expected value = ", format(find_mean(f)[1]),
    "with absolute error <", format(find_mean(f)[2]), "\n")
```

```
## Theoretical expected value =  0.5935632 with absolute error < 1.833209e-05
```

```r
t_test_100 <- t.test(sample_100, mu = theoretical_mean)
t_test_1000 <- t.test(sample_1000, mu = theoretical_mean)
t_test_10000 <- t.test(sample_10000, mu = theoretical_mean)

t_test <- matrix(c(theoretical_mean, t_test_100$estimate, abs(theoretical_mean -
    t_test_100$estimate), t_test_100$p.value, t_test_100$conf.int[[1]],
    t_test_100$conf.int[[2]], theoretical_mean, t_test_1000$estimate,
    abs(theoretical_mean - t_test_1000$estimate), t_test_1000$p.value,
    t_test_1000$conf.int[[1]], t_test_1000$conf.int[[2]], theoretical_mean,
    t_test_10000$estimate, abs(theoretical_mean - t_test_10000$estimate),
    t_test_10000$p.value, t_test_10000$conf.int[[1]], t_test_10000$conf.int[[2]]),
    ncol = 6, byrow = TRUE)

# Notation: CI LB = Confidence Interval Lower Bound' etc.
colnames(t_test) <- c("Theoretical Mean", "Sample Estimate",
    "Absolute Difference", "P-value", "CI LB", "CI UB")
rownames(t_test) <- c("Sample 100", "Sample 1000", "Sample 10000")
t_test <- as.table(round_df(t_test, 4))
kable(t_test, align = "c")
```

	Theoretical Mean	Sample Estimate	Absolute Difference	P-value	CI LB	CI UB
Sample 100	0.5936	0.6347	0.0412	0.1520	0.5781	0.6913
Sample 1000	0.5936	0.6000	0.0065	0.4886	0.5817	0.6184
Sample 10000	0.5936	0.5873	0.0062	0.0352	0.5815	0.5931

We see from the above table that the theoretical mean lies inside the 95% confidence interval for all three samples and it is implied that $|\overline{X} - \mu| \xrightarrow{n \to \infty} 0$. This, alongside the large p-value for each of our three samples, is further verification our ratio-of-uniforms scheme has been implemented correctly.

Chi-squared test

Since we know that the sample mean is asymptotically normally distributed, we can use the chi-squared test for the variance. Like the Student's t-test, the chi-squared test is used to determine whether there is a statistically significant difference between the expected frequencies and the observed frequencies. It is applicable, since $\overline{X}$ is asymptotically normal and therefore $var(\overline{X}) \sim \chi^2_{n-1}$.

```r
# Find the theoretical variance of the distribution f.
find_variance <- function(f, ..., from = 0, to = 1) {
    integrate(function(x) (x - theoretical_mean)^2 * f(x, ...),
        from, to)
}
theoretical_variance <- find_variance(f)[[1]]
cat("Theoretical variance = ", format(find_variance(f)[1]), "with absolute error <",
    format(find_variance(f)[2]), "\n")
```

```
## Theoretical variance =  0.08833495 with absolute error < 8.295023e-05
```

```r
# varTest performs a one-sample chi-squared test. It is
# from the package 'EnvStats'.
chi_test_100 <- varTest(sample_100, sigma.squared = theoretical_variance)
chi_test_1000 <- varTest(sample_1000, sigma.squared = theoretical_variance)
chi_test_10000 <- varTest(sample_10000, sigma.squared = theoretical_variance)

chi_test <- matrix(c(theoretical_variance, chi_test_100$estimate,
    abs(theoretical_variance - chi_test_100$estimate), chi_test_100$p.value,
    chi_test_100$conf.int[[1]], chi_test_100$conf.int[[2]], theoretical_variance,
    chi_test_1000$estimate, abs(theoretical_variance - chi_test_1000$estimate),
    chi_test_1000$p.value, chi_test_1000$conf.int[[1]], chi_test_1000$conf.int[[2]],
    theoretical_variance, chi_test_10000$estimate, abs(theoretical_variance -
        chi_test_10000$estimate), chi_test_10000$p.value, chi_test_10000$conf.int[[1]],
    chi_test_10000$conf.int[[2]]), ncol = 6, byrow = TRUE)

colnames(chi_test) <- c("Theoretical Variance", "Sample Estimate",
    "Absolute Difference", "P-value", "CI LB", "CI UB")
rownames(chi_test) <- c("Sample 100", "Sample 1000", "Sample 10000")
chi_test <- as.table(round_df(chi_test, 4))
kable(chi_test, align = "c")
```

	Theoretical Variance	Sample Estimate	Absolute Difference	P-value	CI LB	CI UB
Sample 100	0.0883	0.0814	7e-03	0.6006	0.0627	0.1098
Sample 1000	0.0883	0.0875	9e-04	0.8375	0.0803	0.0957
Sample 10000	0.0883	0.0873	1e-03	0.4315	0.0850	0.0898

Once again, we have that our theoretical variance is inside the 95% confidence interval for the variance for all three samples and that we have no statistically significant evidence to reject the null hypothesis (since p-value > 0.05 for all three samples). In addition, the sample variance is clearly converging to the theoretical variance.

It is clear from the above diagnostic plots and statistical tests that we have indeed implemented our ratio-of-uniforms scheme correctly and that our `sampling_function` is successful in independantly sampling from $f_X(x)$. This concludes the 'Verification' section of the notebook.

Monte Carlo

In our final section, we implement three Monte Carlo procedures to estimate the normalising constant associated with $f_X(x)$. This is equivalent to finding $\int_0^1 h(x)\ dx$.

Throughout this section we denote:

$$\theta = \int_0^1 h(x)\ dx$$

The two Monte Carlo procedures are as follows:

1. Crude Monte Carlo

Suppose we we write

$$\theta = \int_0^1 \phi(x) f(x)\ dx$$

such that $f(\cdot)$ is a probability density function and $\phi(\cdot) = \frac{h}{f}$.

In this case, we have $\theta = E_f[\phi(X)]$ which motivates a probabilistic approach:

Assuming we can generate $X_1, X_2, ..., X_n \overset{iid}{\sim} f(\cdot)$, we can estimate θ using

$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n} \phi(X_i)$$

This estimator has the following properties:

$$E_f[\hat{\theta}] = E_f\left[\frac{1}{n}\sum_{i=1}^{n}\phi(X_i)\right] = \frac{1}{n}\sum_{i=1}^{n} E_f[\phi(X_i)] = \frac{1}{n} n E_f[\phi(X_i)] = \int \phi(x) f(x)\ dx = \theta$$

$$\text{var}(\hat{\theta}) = \text{var}\left(\sum_{i=1}^{n}\phi(X_i)\right) = \frac{1}{n^2} n \text{var}(\phi(X_i)) = \frac{1}{n} E[[\phi(X) - E[\phi(X)]^2] = \frac{1}{n}\int_0^1 [\phi(x) - \theta]^2 f(x)\ dx = \frac{k}{n}$$

Hence, we see that our estimator is unbiased and large samples will lead to more accurate estimators. Clearly we want to make the variance as small as possible, which motivates the next Monte Carlo method:

2. Control Variates Monte Carlo

We once again consider the problem of estimating

$$\theta = E_f[\phi(X)] = E(Z)$$

via Monte Carlo integration. For notational convenience we write $Z = \phi(X)$.

Suppose $\exists W = \psi(X)$ such that $E(W)$ is known and W is correlated with Z. Then, given a sample $X_1, X_2, ..., X_n \overset{iid}{\sim} f(\cdot)$, we construct the following estimator:

$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n}\Big[Z_i - \beta(W_i - E(W))\Big]$$

where β is an arbitrary variable we can optimise. For a general β the estimator has the following properties:

$$E_f[\hat{\theta}] = \frac{1}{n}\sum_{i=1}^{n}\Big[E_f[Z_i] - \beta[E_f[W_i] - E[W]]\Big] = \frac{1}{n}nE_f[Z_i] = \int \phi(x)f(x)\ dx = \theta$$

$$\text{var}(\hat{\theta}) = \frac{1}{n^2}\sum_{i=1}^{n}\text{var}(Z_i - \beta(W_i - E(W_i))) = \frac{1}{n^2}\sum_{i=1}^{n}\text{var}(Z_i) + \text{var}(\beta(W_i - E(W_i))) - 2\text{cov}(Z_i, \beta(W_i - E(W_i)))$$

$$= \frac{1}{n}\Big(\text{var}(Z) + \beta^2\text{var}(W) - 2\beta\text{cov}(Z, W)\Big)$$

Once again, our estimator is unbiased, but we now have the ability to minimize the variance through an optimal choice of :

$$\frac{\text{dvar}(\hat{\theta})}{\text{d}\beta} = 0 \implies \frac{1}{n}\Big[-2\text{cov}(Z, W) + 2\hat{\beta}\text{var}(W)\Big] = 0 \implies \hat{\beta} = \frac{\text{cov}(Z, W)}{\text{var}(W)}$$

giving

$$\text{var}(\hat{\theta}) = \frac{1}{n}\left(\text{var}(Z) - \frac{\text{cov}^2(Z, W)}{\text{var}(W)}\right)$$

In practice, however, it is often unlikely we would know $\text{cov}(Z, W)$ and we therefore have to use the sample optimisation of β:

$$\hat{\beta} = \frac{\widehat{\text{cov}(Z, W)}}{\widehat{\text{var}(W)}}$$

We will explore both methods of control variates Monte Carlo integration.

Hit-or-Miss Monte Carlo

In addition, we have Hit-or-Miss Monte Carlo which we know is guaranteed to have a higher variance than crude Monte Carlo integration (see Emma McCoy's Stochastic Simulation lecture notes) and is therefore not favourable. Whilst we will not outline the procedure in detail, it involves bounding the function h with a rectangle and generating points uniformly inside this region. We then look at the proportion of points that were randomly generated inside (in the case of a probability density function, underneath) the function h.

We then have

$$\tilde{\theta} = \text{area of rectangle} \ \cdot \ \text{frequency of points under h}$$

For our Monte Carlo integration we will be using

$$f(x) = 1\ , \qquad \phi(x) = h(x)$$

where $h(x)$ is the function we have used throughout this notebook and $f(x)$ is the probability density function of a standard uniform distribution.

Comparison of Crude Monte Carlo and Control Variates Monte Carlo

```
phi <- h
theta <- expectation_phi <- integrate(phi, 0, 1)[[1]]

# Generate random samples from f, which is standard
# uniformly distributed.
unif_sample_100 <- runif(100)
unif_sample_1000 <- runif(1000)
unif_sample_10000 <- runif(10000)

# Generate the corresponding values of phi(X_i) for the
# standard uniform samples.
phi_values_sample_100 <- sapply(unif_sample_100, phi)
phi_values_sample_1000 <- sapply(unif_sample_1000, phi)
phi_values_sample_10000 <- sapply(unif_sample_10000, phi)

# We refer to the Mean Value Estimator for crude Monte
# Carlo as MVE.

# Find the MVE for the crude Monte Carlo method (procedure
# 1. above).
MVE_100 <- sum(phi_values_sample_100)/100
MVE_1000 <- sum(phi_values_sample_1000)/1000
MVE_10000 <- sum(phi_values_sample_10000)/10000

# The function we integrate to find the variance in crude
# Monte Carlo integration.
phi_minus_theta_squared <- function(x) ((exp((-1/7) * (-0.6 +
    log(x/(1 - x)))^2))/(x * (1 - x)) - theta)^2

# Find the variance for the crude Monte Carlo method
# (procedure 1. above).
var_MVE_100 <- integrate(function(x) (h(x) - theta)^2, 0, 1)[[1]]/100
var_MVE_1000 <- var_MVE_100/10  # To avoid further unecessary calculations.
var_MVE_10000 <- var_MVE_1000/10
```

```
psi <- function(x) (1 + x + x^2)
expectation_psi <- integrate(psi, 0, 1)[[1]]  #expecation wrt f.
# Since the expectation is wrt f, a standard uniform
# distribution.
var_phi <- integrate(function(x) (h(x) - theta)^2, 0, 1)[[1]]
var_psi <- integrate(function(x) (psi(x) - expectation_psi)^2,
    0, 1)[[1]]
cov_phi_psi <- integrate(function(x) (phi(x) - theta) * (psi(x) -
    expectation_psi), 0, 1)[[1]]
theoretical_beta_hat <- cov_phi_psi/var_psi

# Generate the corresponding values of psi(X_i) for the
# standard uniform samples.
psi_values_sample_100 <- sapply(unif_sample_100, psi)
psi_values_sample_1000 <- sapply(unif_sample_1000, psi)
psi_values_sample_10000 <- sapply(unif_sample_10000, psi)
```

```r
# Use built-in R functions for variance and covariance to
# estimate beta hat.
beta_hat_100 <- cov(phi_values_sample_100, psi_values_sample_100)/var(psi_values_sample_100)
beta_hat_1000 <- cov(phi_values_sample_1000, psi_values_sample_1000)/var(psi_values_sample_1000)
beta_hat_10000 <- cov(phi_values_sample_10000, psi_values_sample_10000)/var(psi_values_sample_10000)

# We refer to the Mean Value Estimator for control variates
# Monte Carlo as MVE_CV.

# Find the MVE for the control variates Monte Carlo method
# (procedure 2. above) using the sample beta hat.
sample_MVE_CV_100 <- (sum(phi_values_sample_100) - beta_hat_100 *
    sum(psi_values_sample_100 - expectation_psi))/100
sample_MVE_CV_1000 <- (sum(phi_values_sample_1000) - beta_hat_1000 *
    sum(psi_values_sample_1000 - expectation_psi))/1000
sample_MVE_CV_10000 <- (sum(phi_values_sample_10000) - beta_hat_100 *
    sum(psi_values_sample_10000 - expectation_psi))/10000

# Find the MVE for the control variates Monte Carlo method
# (procedure 2. above) using the theoretical beta hat.
theoretical_MVE_CV_100 <- (sum(phi_values_sample_100) - theoretical_beta_hat *
    sum(psi_values_sample_100 - expectation_psi))/100
theoretical_MVE_CV_1000 <- (sum(phi_values_sample_1000) - theoretical_beta_hat *
    sum(psi_values_sample_1000 - expectation_psi))/1000
theoretical_MVE_CV_10000 <- (sum(phi_values_sample_10000) - theoretical_beta_hat *
    sum(psi_values_sample_10000 - expectation_psi))/10000

# Find the variance using the sample beta hat
sample_var_MVE_CV_100 <- (var_phi + var_psi * beta_hat_100^2 -
    2 * beta_hat_100 * cov_phi_psi)/100
sample_var_MVE_CV_1000 <- (var_phi + var_psi * beta_hat_1000^2 -
    2 * beta_hat_1000 * cov_phi_psi)/1000
sample_var_MVE_CV_10000 <- (var_phi + var_psi * beta_hat_10000^2 -
    2 * beta_hat_10000 * cov_phi_psi)/10000

# Find the variance using the theoretical beta hat
theoretical_var_MVE_CV_100 <- (var_phi - (cov_phi_psi^2)/var_psi)/100
theoretical_var_MVE_CV_1000 <- theoretical_var_MVE_CV_100/10
theoretical_var_MVE_CV_10000 <- theoretical_var_MVE_CV_1000/10
```

```r
crudeMC1 <- matrix(c(theta, MVE_100, abs(MVE_100 - theta), sample_MVE_CV_100,
    abs(sample_MVE_CV_100 - theta), theoretical_MVE_CV_100, abs(theoretical_MVE_CV_100 -
        theta), theta, MVE_1000, abs(MVE_1000 - theta), sample_MVE_CV_1000,
    abs(sample_MVE_CV_1000 - theta), theoretical_MVE_CV_1000,
    abs(theoretical_MVE_CV_1000 - theta), theta, MVE_10000, abs(MVE_10000 -
        theta), sample_MVE_CV_10000, abs(sample_MVE_CV_10000 -
        theta), theoretical_MVE_CV_10000, abs(theoretical_MVE_CV_10000 -
        theta)), ncol = 7, byrow = TRUE)

# As before, we use shorthand notation. C = Crude, CVt =
# Control Variates with Theoretical beta hat, CVs = Control
# Variates with Sample beta hat.  Where 'Abs' is used as a
# column header, it is assumed that this refers to the
# absolute difference between the value in the previous
# column and the value it is attempting to estimate.

colnames(crudeMC1) <- c("Theta", "MVE C", "Abs", "MVE CVs", "Abs",
    "MVE CVt", "Abs")
rownames(crudeMC1) <- c("Sample 100", "Sample 1000", "Sample 10000")
crudeMC1 <- as.table(round_df(crudeMC1, 4))
kable(crudeMC1, align = "c")
```

	Theta	MVE C	Abs	MVE CVs	Abs	MVE CVt	Abs
Sample 100	4.6895	4.6240	0.0655	4.5296	0.1598	4.4971	0.1923
Sample 1000	4.6895	4.7528	0.0633	4.7171	0.0277	4.7182	0.0287
Sample 10000	4.6895	4.6882	0.0013	4.6794	0.0101	4.6764	0.0131

```r
crudeMC2 <- matrix(c(var_MVE_100, sample_var_MVE_CV_100, theoretical_var_MVE_CV_100,
    var_MVE_1000, sample_var_MVE_CV_1000, theoretical_var_MVE_CV_1000,
    var_MVE_10000, sample_var_MVE_CV_10000, theoretical_var_MVE_CV_10000),
    ncol = 3, byrow = TRUE)
colnames(crudeMC2) <- c("Var(MVE)", "Var(MVE CVs)", "Var(MVE CVt)")
rownames(crudeMC2) <- c("Sample 100", "Sample 1000", "Sample 10000")
crudeMC2 <- as.table(round_df(crudeMC2, 4))
kable(crudeMC2, align = "c")
```

	Var(MVE)	Var(MVE CVs)	Var(MVE CVt)
Sample 100	0.0350	0.0105	0.0088
Sample 1000	0.0035	0.0009	0.0009
Sample 10000	0.0004	0.0001	0.0001

It is clear that all three of our Monte Carlo procedures (crude, control variates with sample $\hat{\beta}$ and control variates with theoretical $\hat{\beta}$) have been implemented correctly, given that each of their mean value estimators converge to the theoretical mean. We note that both of the control variates methods are substantially better at estimating θ and have a variance roughly 4x smaller than the crude Monte Carlo procedure, as we would expect given the choice of $\hat{\beta}$.

It is also interesting to note that $\hat{\beta}$ (theoretical) $= 2.7798 \approx \hat{\beta}$ (sample 10000) $= 2.7518$ and we would expect $\hat{\beta}$ (sample N) $\rightarrow \hat{\beta}$ (theoretical) almost surely.

Comparison of Control Variates Monte Carlo (theoretical $\hat{\beta}$) and Hit-or-Miss Monte Carlo

```r
h_max <- optimise(h, c(0, 1), maximum = TRUE)[[2]]

hitormiss_MC <- function(n) {
    u <- runif(n)
    v <- runif(n, min = 0, max = h_max)
    accepted <- v[v < h(u)]  # Values that are below the curve h.
    # Calculate an estimate for the integral, since
    # length/n is the proportion of points below the curve
    # and h_max = c = c * (1-0) = area of the rectangle.
    area <- h_max * length(accepted)/n
    return(area)
}

# Compare hit-or-miss integration with theoretical control
# variate integration (since this gave us the best estimate
# and lowest variance).

# Generate hit-or-miss integral estimations.
hitormiss_100 <- hitormiss_MC(100)
hitormiss_1000 <- hitormiss_MC(1000)
hitormiss_10000 <- hitormiss_MC(10000)

# Generate hit-or-miss theoretical integral variances (as
# discussed in lectures).
var_hitormiss_100 <- (theta * (h_max - theta))/100
var_hitormiss_1000 <- var_hitormiss_100/10
var_hitormiss_10000 <- var_hitormiss_1000/10
```

```r
hitormiss <- matrix(c(theta, theoretical_MVE_CV_100, abs(theta -
    theoretical_MVE_CV_100), theoretical_var_MVE_CV_100, hitormiss_100,
    abs(theta - hitormiss_100), var_hitormiss_100, theta, theoretical_MVE_CV_1000,
    abs(theta - theoretical_MVE_CV_1000), theoretical_var_MVE_CV_1000,
    hitormiss_1000, abs(theta - hitormiss_1000), var_hitormiss_1000,
    theta, theoretical_MVE_CV_10000, abs(theta - theoretical_MVE_CV_10000),
    theoretical_var_MVE_CV_10000, hitormiss_10000, abs(theta -
        hitormiss_10000), var_hitormiss_10000), ncol = 7, byrow = TRUE)

# As above, we use the notation HoM = Hit-or-Miss
colnames(hitormiss) <- c("Theta", "MVE CVt", "Abs", "Var(MVE CVt)",
    "MVE HoM", "Abs", "Var(MVE HoM)")
rownames(hitormiss) <- c("Sample 100", "Sample 1000", "Sample 10000")
hitormiss <- as.table(round_df(hitormiss, 4))
knitr::kable(hitormiss, align = "c", booktabs = TRUE)
```

	Theta	MVE CVt	Abs	Var(MVE CVt)	MVE HoM	Abs	Var(MVE HoM)
Sample 100	4.6895	4.4971	0.1923	0.0088	5.8638	1.1743	0.2893
Sample 1000	4.6895	4.7182	0.0287	0.0009	4.6367	0.0527	0.0289
Sample 10000	4.6895	4.6764	0.0131	0.0001	4.6834	0.0060	0.0029

Whilst the hit-or-miss Monte Carlo procedure performs adaquately (in that it will converge to the true normalising constant, θ, for large enough N), it is clearly an inferior method to the control variates Monte Carlo procedure: the hit-or-miss mean value estimator is less accurate and converges to θ much slower than the control variates mean value estimator. In addition, the variance is approximately 32.8067x larger, which is to be expected. It is clear that, whilst both methods have seen implemented correctly and will (with a sufficiently large enough data set) provide a decent estimate for θ, the control variates Monte Carlo procedure is clearly superior and verifies our `h_integral` variable defined in section 1 of the notebook.

Summary

In this notebook, we discussed, outlined and implemented a ratio-of-uniforms scheme to sample from a probability distribution $f_X(x)$ known only up to proportionality. Subsequently, we verified this scheme via multiple diagnostic plots and statistical tests. Our conclusion was that we had indeed implemented our scheme, and sampled from $f_X(x)$, succesfully. Finally, we implemented three Monte Carlo procedures to esimate the normalising constant associated with $f_X(x)$, and our results agreed with the value previously found via R's built-in `integrate()` function and exemplified how we should always prefer to use crude Monte Carlo methods over the hit-or-miss procedure.