Trapezoid Rule and Monte Carlo Estimation

Michael Rose

October 29, 2018

Trapezoidal Rule with 10 Partitions

Consider $\int_0^1 x^2 dx$. Using evaluation at 10 points, we will approximate it using the trapezoid rule in **R**.

The composite trapezoidal rule divides the integral into n subintervals. The trapezoid rule is then performed on each of those n subintervals.

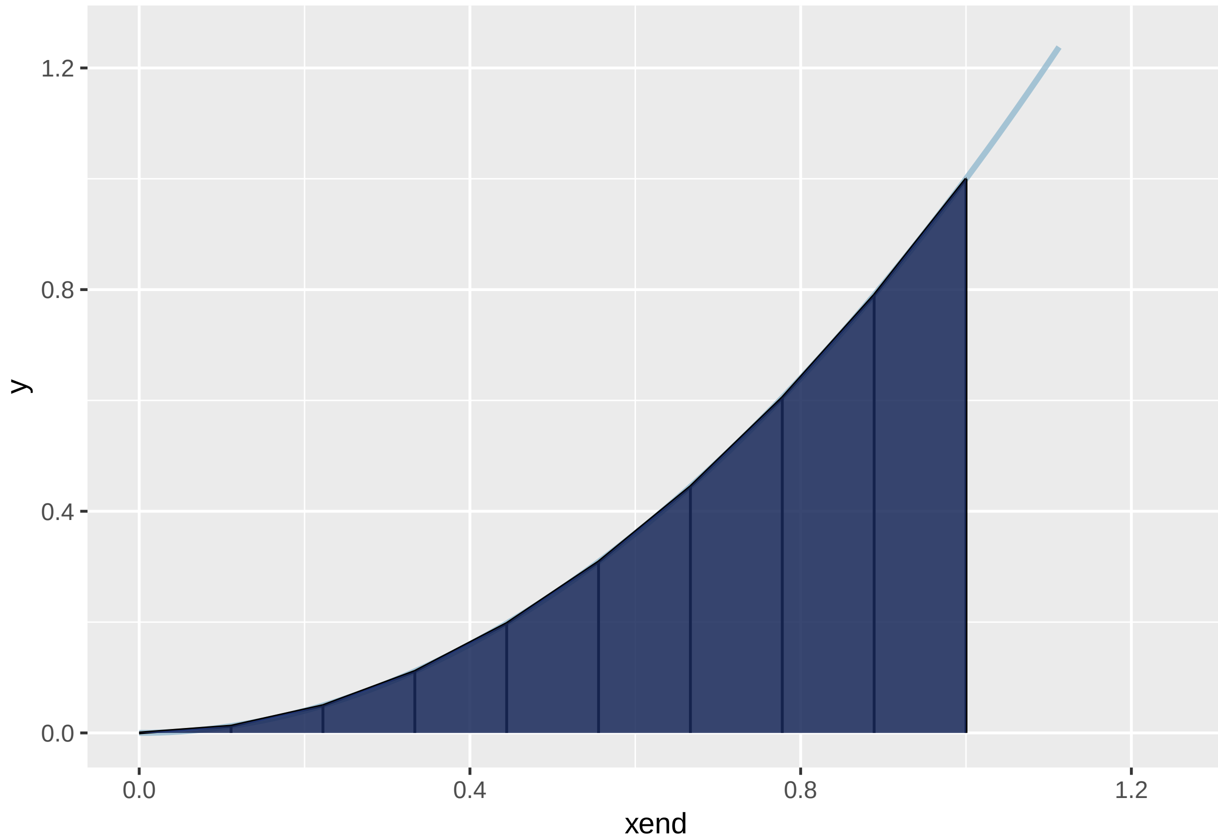
Let our function f be twice differentiable in the interval $[a, b]$. Also let $h = \frac{(b-a)}{n}$ and $x_j = a + jh$ for each $j = 0, 1, ..., n$. Then the composite trapezoidal rule, with its error term is defined as the following:

$\int_a^b f(x)dx = \frac{h}{2}[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)] - \frac{b-a}{12}h^2 f''(\mu)$

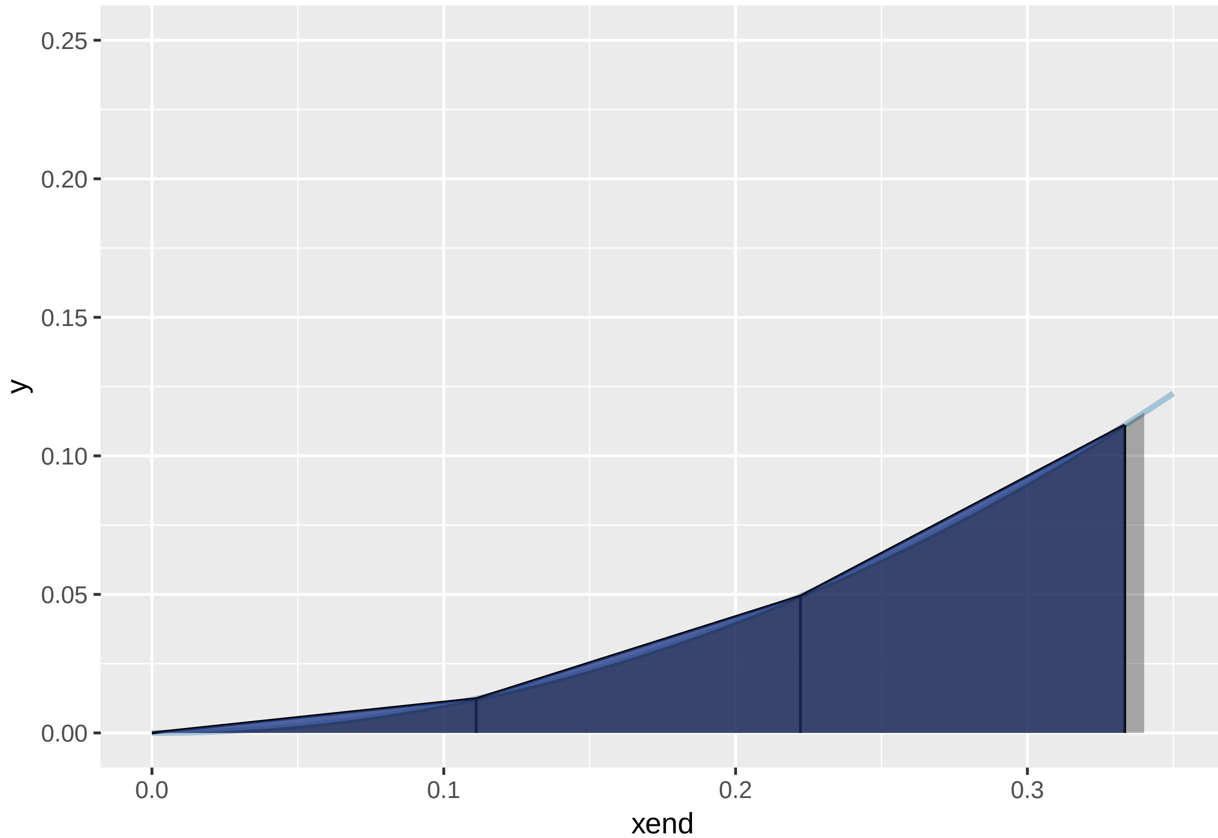
where, since there exists a number μ between $[a, b]$:

error $= -\frac{b-a}{12n^2}f''(\mu)$.

```
# define our function
given_function <- function(x){
  return(x * x)
}

# break the interval into 10 subintervals
subintervals <- seq.int(0, 1, length.out = 10)

# create a vector for functions outputs
fx <- vector(length = length(subintervals))

# for each subinterval, calculate the function
for (i in 1:length(subintervals)){
  fx[i] <- given_function(subintervals[i])
}

# collect our points into a data frame
needed_points <- tibble(xend = subintervals,
                        y = rep(0, 10),
                        yend = fx,
                        yend1 = c(fx[2:10], fx[10]),
                        xend1 = c(subintervals[2:10], subintervals[10])
                        )

# plot the function and its approximation
ggplot(data = needed_points) +
  stat_function(fun = given_function, size = 1.05, alpha = 0.75, color = 'lightskyblue3') +
  geom_segment(aes(x = xend, y = y, xend = xend, yend = yend)) +
  geom_segment(aes(x = xend, y= yend, xend = xend1, yend = yend1)) +
  geom_ribbon(aes(x = xend, ymin = y, ymax = yend), fill = 'royalblue4', alpha = 0.8) +
  geom_area(stat = 'function', fun = given_function, fill = 'black', alpha = 0.3, xlim = c(0, 1)) +
  xlim(c(0, 1.25)) + ylim(c(0, 1.25))
```




```r
# zooming in
ggplot(data = needed_points) +
  stat_function(fun = given_function, size = 1.05, alpha = 0.75, color = 'lightskyblue3') +
  geom_segment(aes(x = xend, y = y, xend = xend, yend = yend)) +
  geom_segment(aes(x = xend, y= yend, xend = xend1, yend = yend1)) +
  geom_ribbon(aes(x = xend, ymin = y, ymax = yend), fill = 'royalblue4', alpha = 0.8) +
  geom_area(stat = 'function', fun = given_function, fill = 'black', alpha = 0.3, xlim = c(0, 1)) +
  xlim(c(0, 0.35)) + ylim(c(0, 0.25))
```



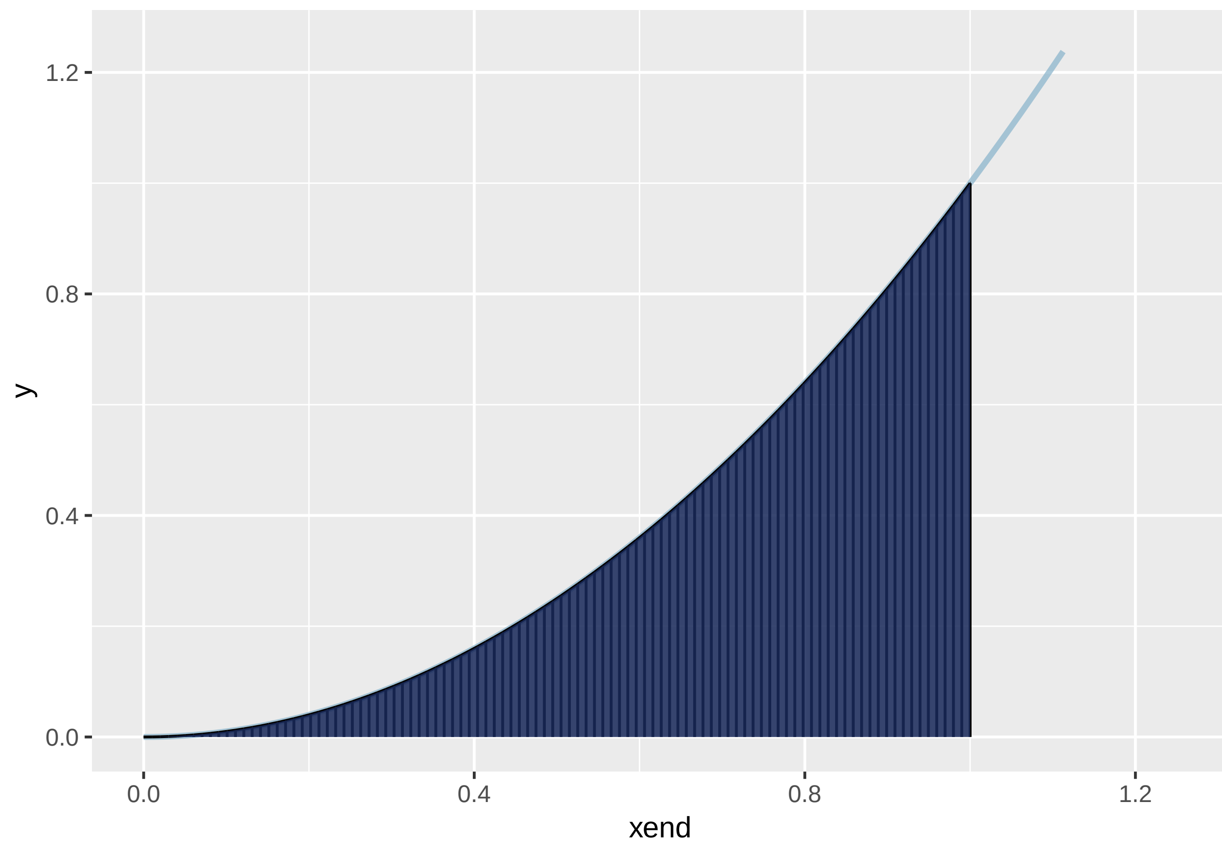
Numerical Solution

```
# composite trapezoid function
comp_trapezoid <- function(f, a, b, n){
  # check to make sure the function f is valid
  if (is.function(f) == FALSE){
    stop('f must be a valid function with one parameter.')
  }
  # implementation
  h <- (b-a)/n
  j <- 1:n - 1
  xj <- a + j*h
  approx <- (h/2) * (f(a) + 2 * sum(f(xj)) + f(b))

  return(approx)
}

# calculation with 10 partitions
comp_trapezoid(given_function, 0, 1, 10)
```

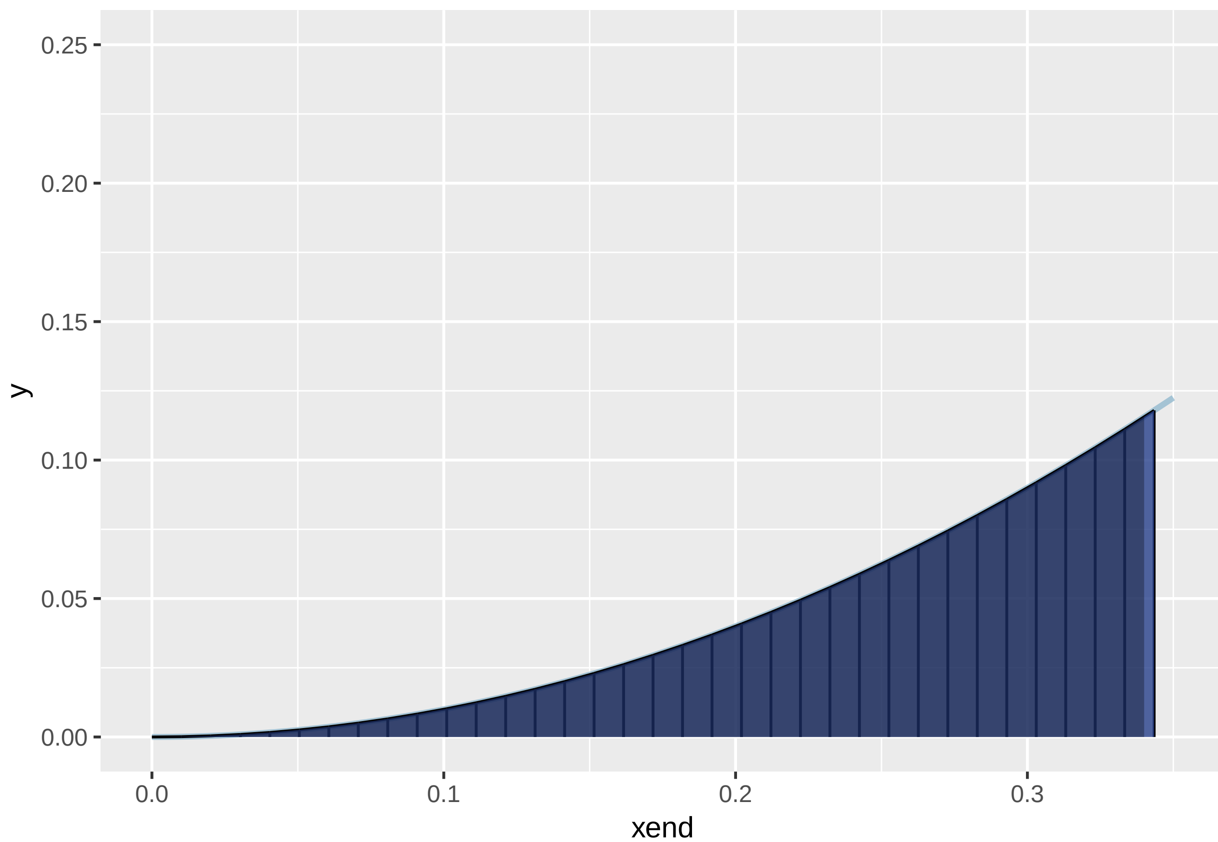
```
## [1] 0.335
```

Trapezoidal Rule with 100 Partitions

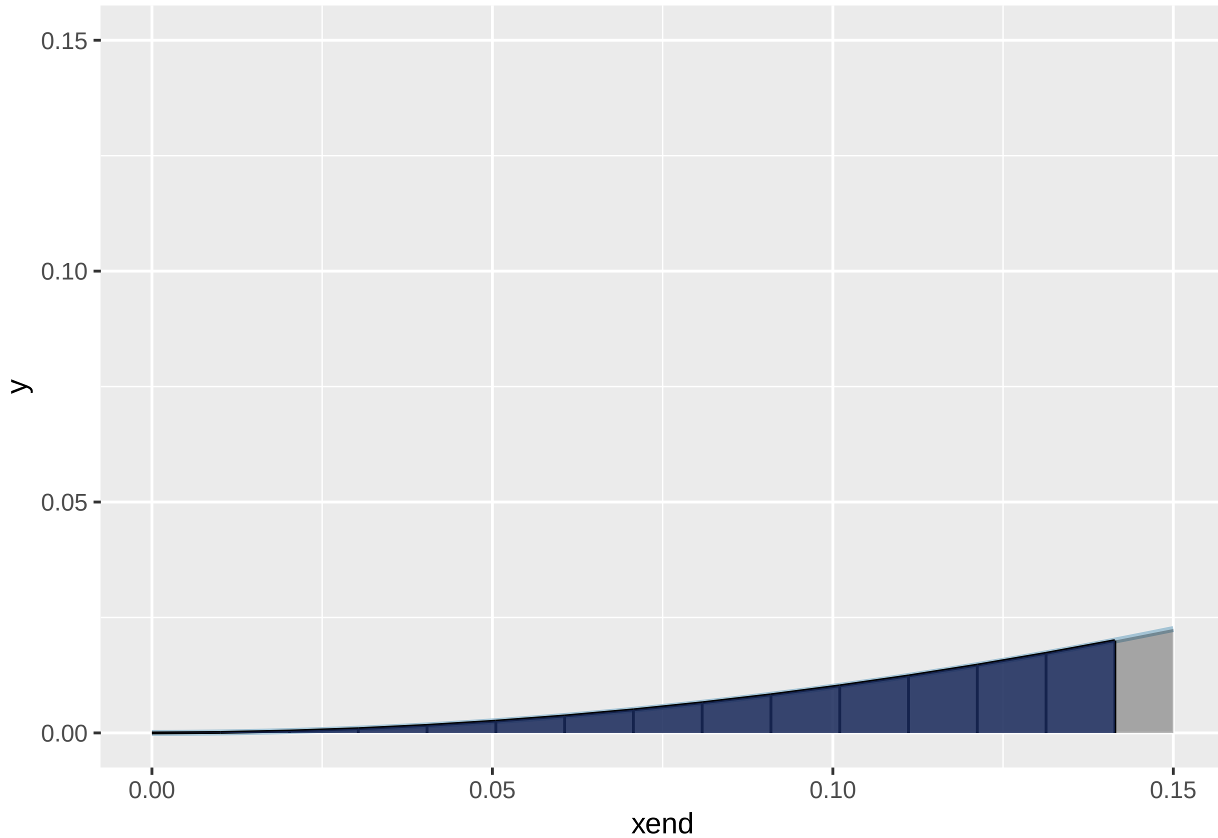
```r
# break the interval into 100 subintervals
subintervals <- seq.int(0, 1, length.out = 100)

# create a vector for functions outputs
fx <- vector(length = length(subintervals))

# for each subinterval, calculate the function
for (i in 1:length(subintervals)){
  fx[i] <- given_function(subintervals[i])
}

# collect our points into a data frame
needed_points <- tibble(xend = subintervals,
                        y = rep(0, 100),
                        yend = fx,
                        yend1 = c(fx[2:100], fx[100]),
                        xend1 = c(subintervals[2:100], subintervals[100])
                        )

# plot the function and its approximation
ggplot(data = needed_points) +
  stat_function(fun = given_function, size = 1.05, alpha = 0.75, color = 'lightskyblue3') +
  geom_segment(aes(x = xend, y = y, xend = xend, yend = yend)) +
  geom_segment(aes(x = xend, y= yend, xend = xend1, yend = yend1)) +
  geom_ribbon(aes(x = xend, ymin = y, ymax = yend), fill = 'royalblue4', alpha = 0.8) +
  geom_area(stat = 'function', fun = given_function, fill = 'black', alpha = 0.3, xlim = c(0, 1)) +
  xlim(c(0, 1.25)) + ylim(c(0, 1.25))
```




```
# zooming in
ggplot(data = needed_points) +
  stat_function(fun = given_function, size = 1.05, alpha = 0.75, color = 'lightskyblue3') +
  geom_segment(aes(x = xend, y = y, xend = xend, yend = yend)) +
  geom_segment(aes(x = xend, y= yend, xend = xend1, yend = yend1)) +
  geom_ribbon(aes(x = xend, ymin = y, ymax = yend), fill = 'royalblue4', alpha = 0.8) +
  geom_area(stat = 'function', fun = given_function, fill = 'black', alpha = 0.3, xlim = c(0, 1)) +
  xlim(c(0, 0.35)) + ylim(c(0, 0.25))
```




```r
# closer
ggplot(data = needed_points) +
  stat_function(fun = given_function, size = 1.05, alpha = 0.75, color = 'lightskyblue3') +
  geom_segment(aes(x = xend, y = y, xend = xend, yend = yend)) +
  geom_segment(aes(x = xend, y= yend, xend = xend1, yend = yend1)) +
  geom_ribbon(aes(x = xend, ymin = y, ymax = yend), fill = 'royalblue4', alpha = 0.8) +
  geom_area(stat = 'function', fun = given_function, fill = 'black', alpha = 0.3, xlim = c(0, 1)) +
  xlim(c(0, 0.15)) + ylim(c(0, 0.15))
```




```
# calculation with 100 partitions
comp_trapezoid(given_function, 0, 1, 100)
```

```
## [1] 0.33335
```

Monte Carlo Approximation

We wish to integrate $I(f) = \int_a^b f(x)dx$.

- First we will choose some pdf we hope to sample from $g(x) \in [a, b]$.
- Then we can generate data $X_1, X_2, ..., X_n$ from $g(x)$.
- Finally, we estimate $I(f) = \sum_{i=1}^{n} \frac{f(g(x))}{n}$ and each iteration provides a new sample from $g(x)$.

```
# function for arbitrary f
# n is number iterations
# a is lower bound
# b is upper bound
# f is the function to be approximated
# since f in [a, b] we can use a uniform sampling function

mc_Integral_Unif <- function(n, a, b, f){
  # check to make sure functions are valid
  if (is.function(f) == FALSE){
    stop('f and g must be valid functions with one parameter.')
  }
```

```r
  # sampling function
  g <- runif(n, a, b)

  #
  y <- (f(g))/(1/(b-a))

  # int approximation
  Int <- sum(y)/n

  # standard error of int and 95% confidence interval
  y_squared <- y^2
  se <- sqrt((sum(y_squared)/n-Int^2)/n)
  ci_l <- Int - 1.96 * se
  ci_u <- Int + 1.96 * se
  list("Int" = Int, "SE" = se, "95% CI Lower Bound" = ci_l, "95% CI Upper Bound" = ci_u)
}
```

```r
# run for 10 iterations
mc_Integral_Unif(10, 0, 1, given_function)
```

```
## $Int
## [1] 0.277201
##
## $SE
## [1] 0.08671253
##
## $`95% CI Lower Bound`
## [1] 0.1072444
##
## $`95% CI Upper Bound`
## [1] 0.4471576
```

```r
# run for 100 iterations
mc_Integral_Unif(100, 0, 1, given_function)
```

```
## $Int
## [1] 0.3110383
##
## $SE
## [1] 0.02785532
##
## $`95% CI Lower Bound`
## [1] 0.2564419
##
## $`95% CI Upper Bound`
## [1] 0.3656347
```

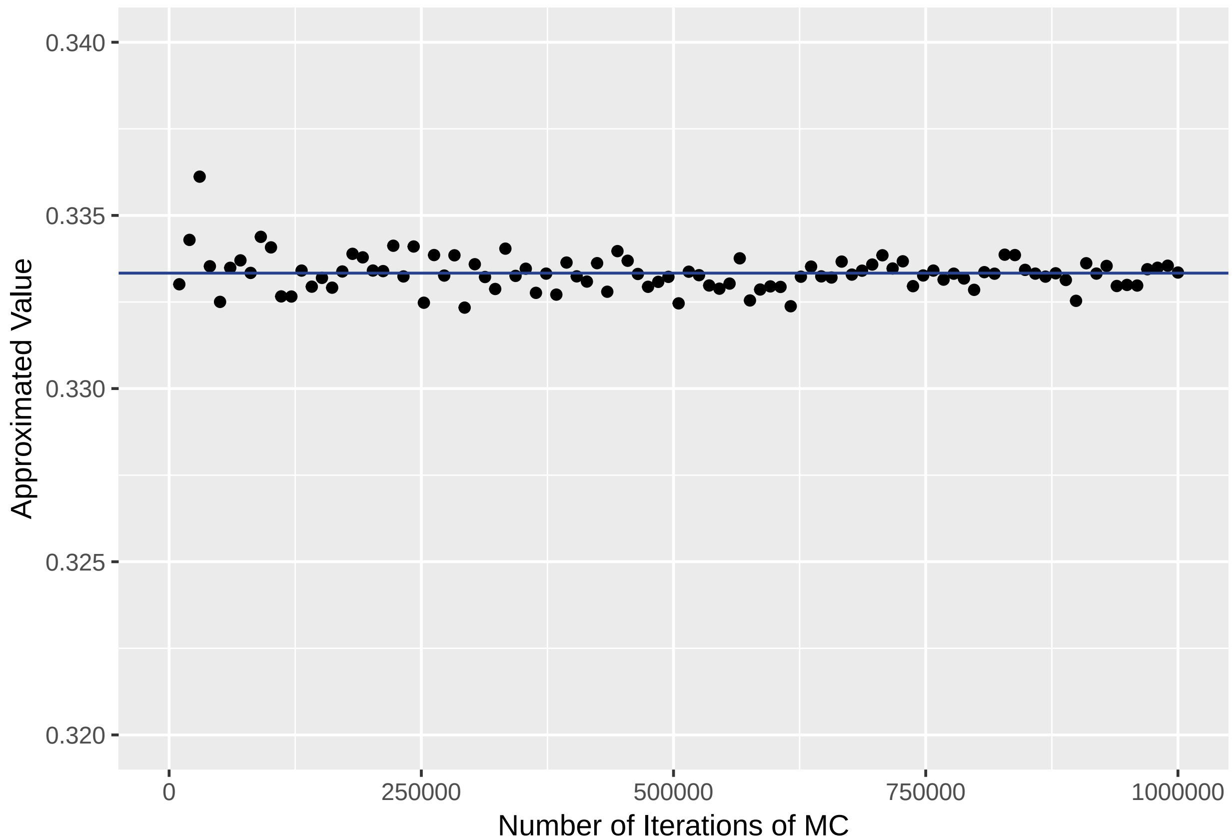
Higher Iterations of Monte Carlo

```r
# break the interval into 100 subintervals
subintervals <- seq.int(0, 1000000, length.out = 100)

# create a matrix for functions outputs
fx <- matrix(NA, nrow = length(subintervals), ncol = 4)
```

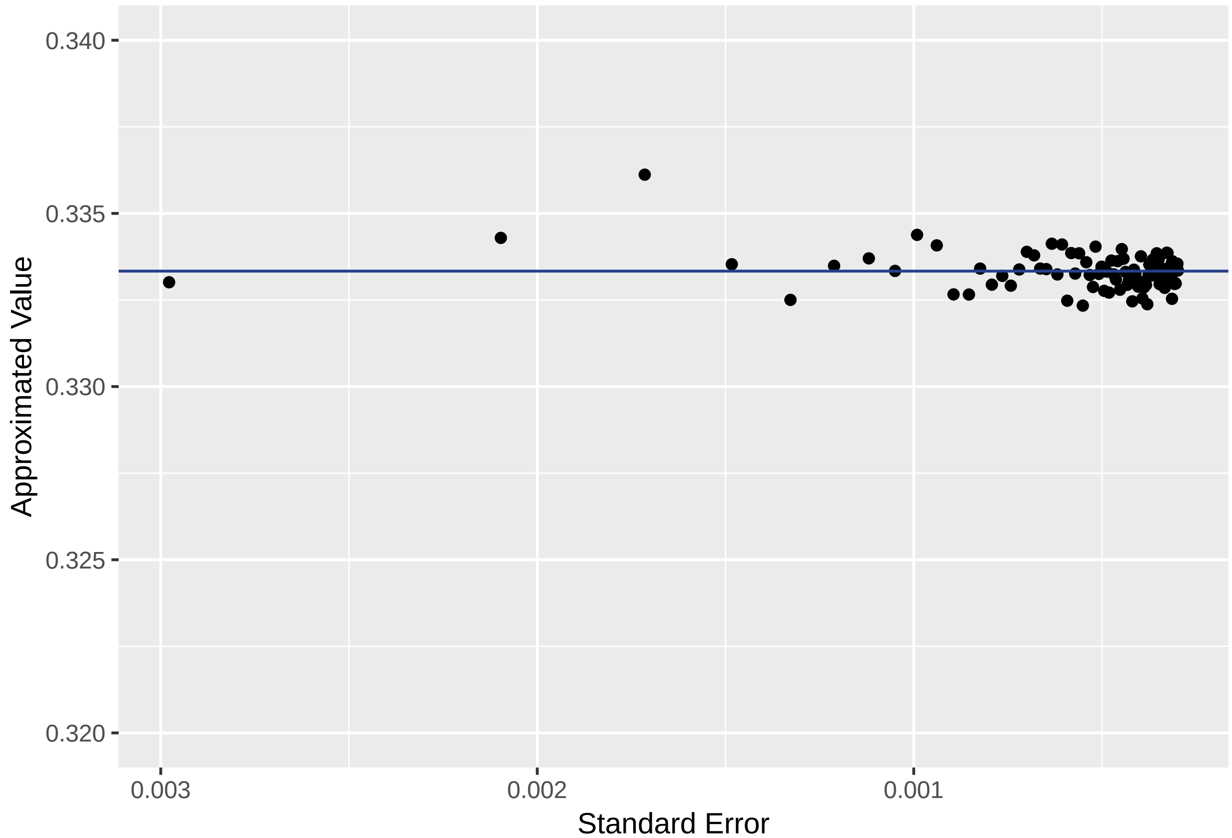
```r
# for each subinterval, calculate the function
for (i in 1:length(subintervals)){
  v <- mc_Integral_Unif(subintervals[i], 0, 1, given_function)
  fx[i, 1] <- v$Int
  fx[i, 2] <- v$SE
  fx[i, 3] <- v$`95% CI Lower Bound`
  fx[i, 4] <- v$`95% CI Upper Bound`
}

# collect our points into a data frame
needed_points <- tibble(
                     exp_num = 1:100,
                     num_iters = subintervals,
                     approx_val = fx[, 1],
                     standard_error = fx[, 2],
                     conf_int_lb = fx[, 3],
                     conf_int_ub = fx[,4]
                     )

needed_points
```

```
## # A tibble: 100 x 6
##    exp_num num_iters approx_val standard_error conf_int_lb conf_int_ub
##      <int>     <dbl>      <dbl>          <dbl>       <dbl>       <dbl>
##  1       1         0      NaN          NaN            NaN         NaN
##  2       2     10101.     0.333        0.00298        0.327       0.339
##  3       3     20202.     0.334        0.00210        0.330       0.338
##  4       4     30303.     0.336        0.00171        0.333       0.339
##  5       5     40404.     0.334        0.00148        0.331       0.336
##  6       6     50505.     0.333        0.00133        0.330       0.335
##  7       7     60606.     0.333        0.00121        0.331       0.336
##  8       8     70707.     0.334        0.00112        0.332       0.336
##  9       9     80808.     0.333        0.00105        0.331       0.335
## 10      10     90909.     0.334        0.000991       0.332       0.336
## # ... with 90 more rows
```

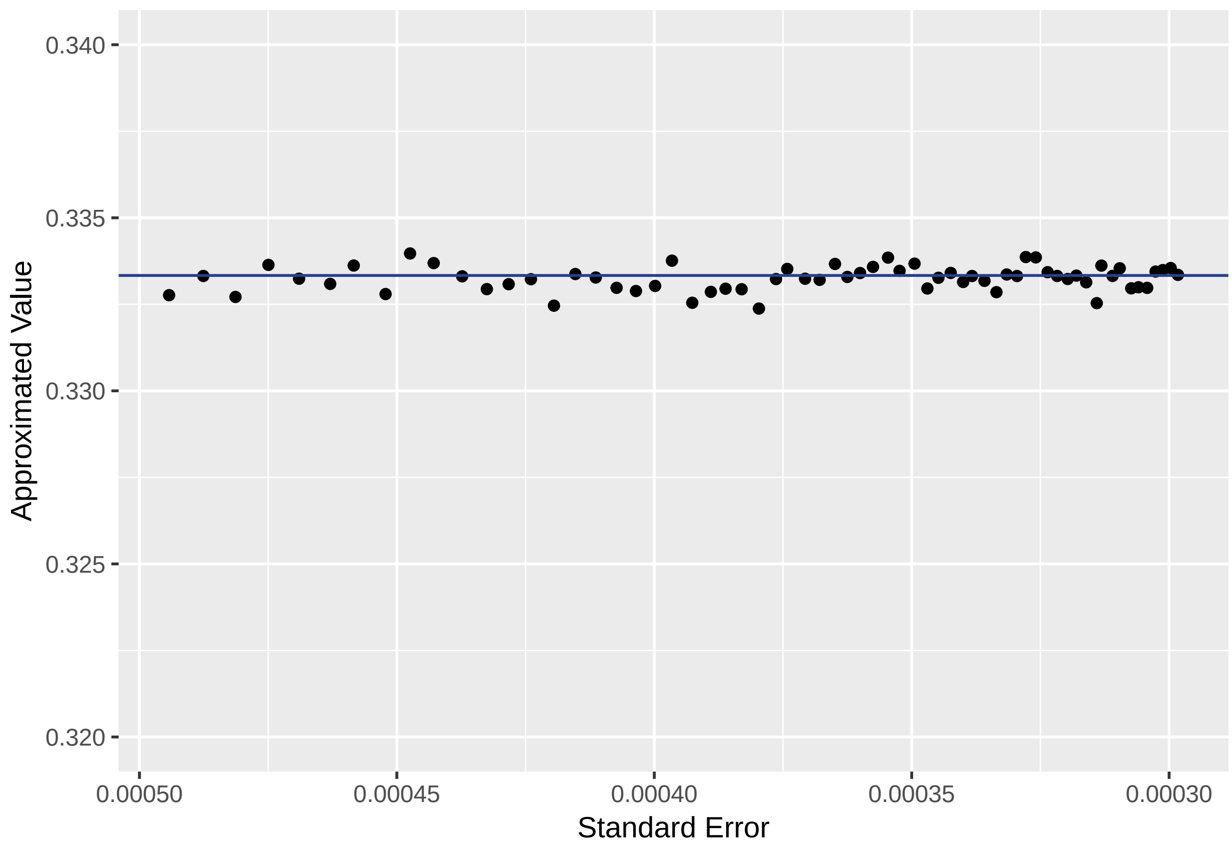
```r
# value vs num iters
ggplot(needed_points, aes(needed_points$num_iters, needed_points$approx_val)) +
  geom_point() +
  geom_hline(yintercept = 1/3, color = "royalblue4") +
  ylab("Approximated Value") + xlab("Number of Iterations of MC") +
  ylim(c(0.32, 0.34))
```




```r
# value vs standard error
ggplot(needed_points, aes(needed_points$standard_error, needed_points$approx_val)) +
  geom_point() +
  geom_hline(yintercept = 1/3, color = "royalblue4") +
  ylab("Approximated Value") + xlab("Standard Error") +
  ylim(c(0.32, 0.34)) + scale_x_reverse()
```




```r
# look at points with less than 5/10,000th error
se_0005 <- needed_points %>%
  filter(needed_points$standard_error < 0.0005)

ggplot(se_0005, aes(se_0005$standard_error, se_0005$approx_val)) +
  geom_point() +
  geom_hline(yintercept = 1/3, color = "royalblue4") +
  ylab("Approximated Value") + xlab("Standard Error") +
  ylim(c(0.32, 0.34)) + scale_x_reverse()
```



R's Given Value

```r
integrate(given_function, 0, 1)
```

```
## 0.3333333 with absolute error < 3.7e-15
```