STA601 Lab 1: R and Monte Carlo Review"

your name here

September 4, 2021

A very very gentle introduction to Monte Carlo methods via R

In this lab we will briefly cover Monte Carlo integration methods. This will be a hopefully useful introduction to R.

Let $y_1, y_2, \ldots, y_n \overset{iid}{\sim} q$, and let $h : \mathbb{R} \to \mathbb{R}$ be a function. Then

$$\frac{1}{n}\sum_{i=1}^{n} h(y_i) \to \mathbb{E}_q[h(Y)] = \int_{-\infty}^{\infty} h(y)q(y)dy$$

This method is super useful for the computation of the moments of a distribution. The steps are easy:

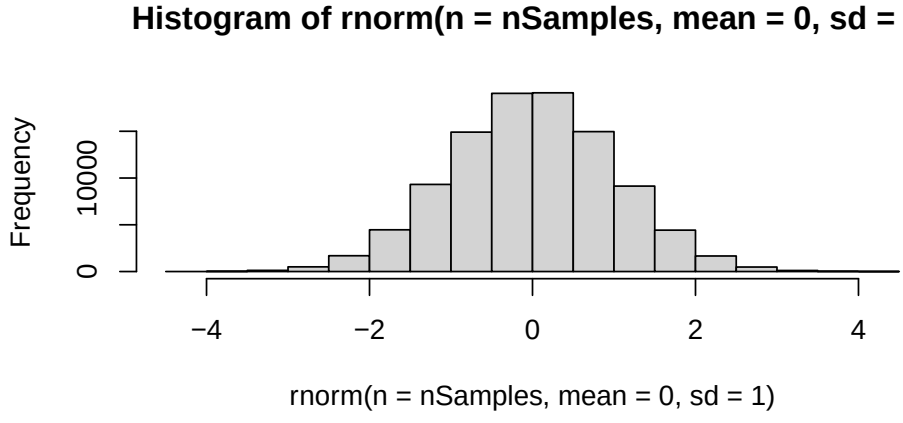
1. Draw n iid samples from a chosen distribution (say, a standard normal)
2. For every sample, compute $h(y_i)$
3. Take the mean of $h(y_1), \ldots, h(y_n)$

Step 1 - random number generation

A a first step, we need to understand how to generate random samples. Let's start with the normal.

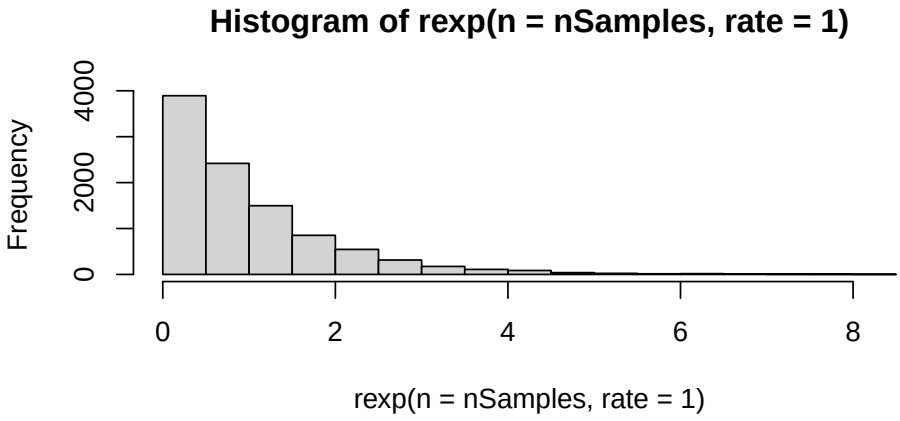
First, let's generate 1000 samples from a normal distribution with mean 0 and sd 1

```
> nSamples = 100000
> hist(rnorm(n = nSamples, mean = 0, sd = 1))
```



This reasoning applies to every distribution:

```
> nSamples = 10000
> hist(rexp(n = nSamples, rate = 1))
```



Highly recommended: write your functions first!

```
> # A very very simple example on how to write your homework correctly.
> # Goal: write a function that computes the mean of the normal distribution
> compute_mean_norm_MC = function(nSamples){
+   samples = rnorm(n = nSamples, mean = 0, sd = 1)
+   MC_estimate = mean(samples)
+   return(MC_estimate)
+ }
```

and test it!

```
> nSamples = 200000
> compute_mean_norm_MC(nSamples)
```

```
[1] 0.001095772
```

Task:

Write your own function to compute the standard deviation of the normal distribution!

```
> compute_sd_norm_MC = function(nSamples){
+   samples = rnorm(n = nSamples, mean = 0, sd = 1)
+   MC_estimate = sd(samples)
+   return(MC_estimate)
+ }
```

```
> nSamples = 200000
> compute_sd_norm_MC(nSamples)
```

```
[1] 1.001108
```

Higher Moments

What if we want to compute more than one moment at a time? We can even be more creative. Let's compute the first 10 moments of the normal distribution

```
> compute_moments_norm_MC = function(nSamples=10000, mean=0, sd=1, nMoms=1){
+
+   # Step 1 -  Draw the random samples
+   samples = rnorm(n = nSamples, mean = mean, sd = sd)
+
+   # Step 2 -  Inizialize the empty vector
+   moments = rep(NA, nMoms)
+
+   # Step 3 -  Compute the moments
+   for(i in 1:nMoms){
+     moments[i] = mean(samples^i)
+   }
+
+   return(moments)
+ }
```

and now run the function

```
> # compute the first 10 moments of a standard normal.
> round(compute_moments_norm_MC(nSamples=100000,
+                               nMoms = 10),4)

 [1]   0.0004   0.9980   0.0061   2.9873   0.0460  14.8229   0.4504 102.0315
 [9]   6.5827 891.3376
```

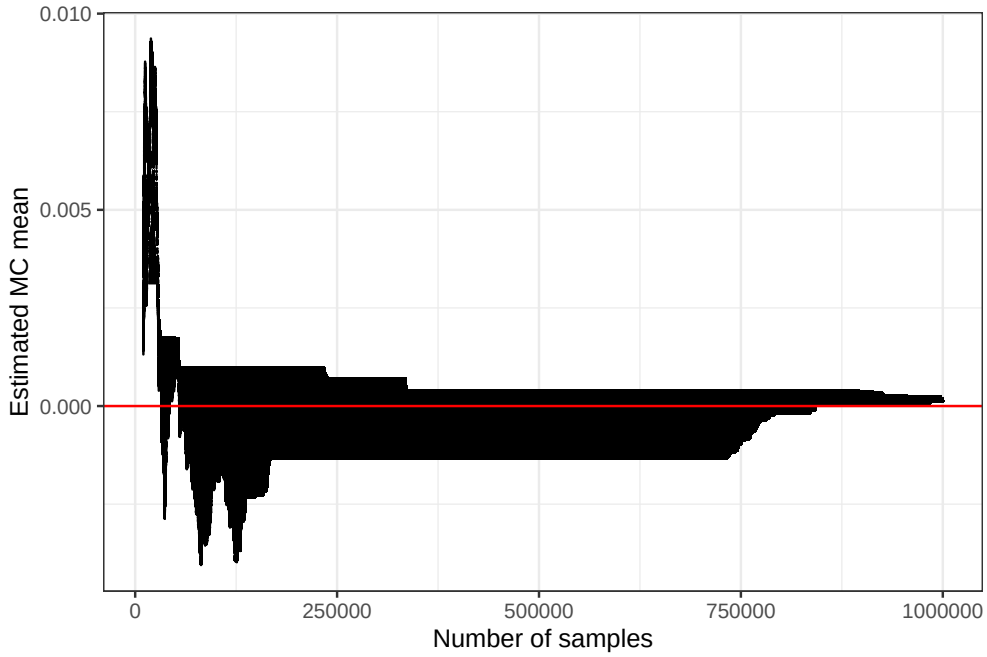
How does the number of samples influcences the estimate?

We can clearly see a dependence between the number of samples drawn, and the precision of the Monte Carlo estimate.

```
> compute_MC_mean_normal = function(nSamples){
+   means = rep(NA, nSamples)
+   S_samples = rep(NA, nSamples)
+   for (n in 1:nSamples){
+     # Step 1 - sample from a distribution
+     S = rnorm(1)
+     S_samples[n] = S
+     # Step 2 - update the estimate
+     if(n == 1){
+       means[n] = S
+     } else {
+       means[n] = ((n-1)*means[n-1] + S)/n
+     }
+   }
+
+
+   df_means = data.frame("y_bar" = means,
+                         "iter" = c(1:nSamples))
+   return(df_means)
+ }
```

Run the function

```
> nSamples  = 1000000
> df_means = compute_MC_mean_normal(nSamples)
> # Plot the result via ggplot
> ggplot(data = df_means[c(10000:nSamples),])+
+   geom_line(aes(x = iter, y = y_bar))+
+   xlab("Number of samples")+
+   ylab("Estimated MC mean")+
+   geom_hline(yintercept = 0, color = "red")
>
```



Task:

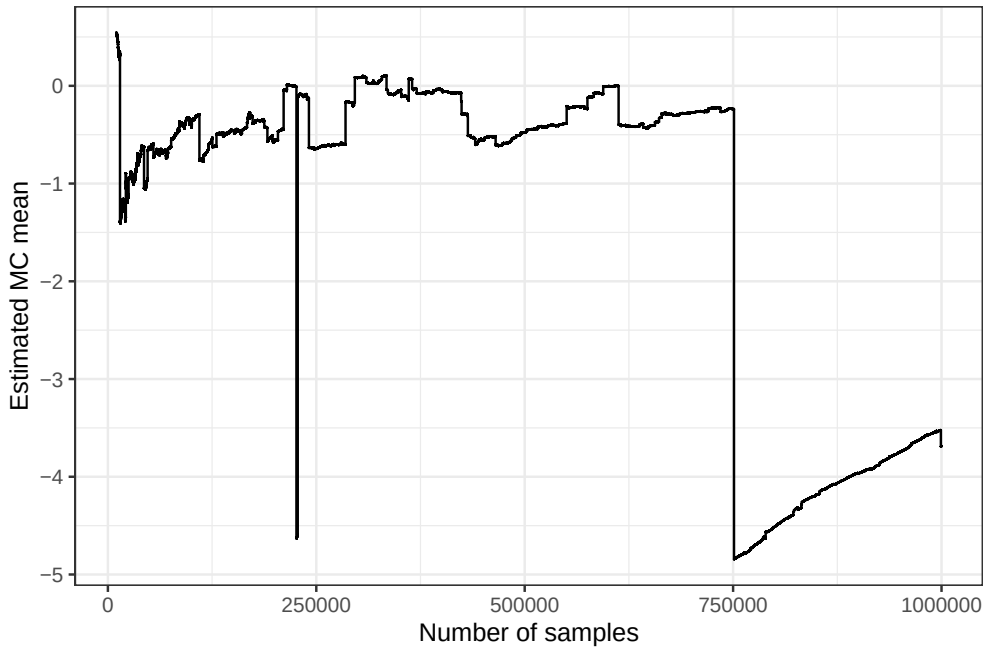
What happens if we sample from a Cauchy? Write a similar function, but this time, sample from a Cauchy distribution.

```
> compute_MC_mean_cauchy = function(nSamples){
+    means = rep(NA, nSamples)
+   S_samples = rep(NA, nSamples)
+   for (n in 1:nSamples){
+     # Step 1 - sample from a distribution
+     S = rcauchy(1)
+     S_samples[n] = S
+     # Step 2 - update the estimate
+     if(n == 1){
+       means[n] = S
+     } else {
+       means[n] = ((n-1)*means[n-1] + S)/n
+     }
+   }
+
```

```
+   df_means = data.frame("y_bar" = means, "iter" = c(1:nSamples))
+   return(df_means)
+ }
```

And make the plot!

```
> # Run the function a
> df_means_cauchy = compute_MC_mean_cauchy(nSamples=1000000)
> # Plot the result via ggplot
> ggplot(data = df_means_cauchy[c(10000:nSamples),])+
+   geom_line(aes(x = iter, y = y_bar))+
+   xlab("Number of samples")+
+   ylab("Estimated MC mean")
```



What is happening? The Cauchy distribution has an infinite expected value! If $X \sim Cauchy(0, 1)$, the associated density function is

$$p(x) = \frac{1/\pi}{1 + x^2}$$

and it follows easily that

$$\mathbb{E}[X] = +\infty$$

Another cool thing ... while loops

We have introduced `for` loops. But there is another interesting type of loop that can be very useful. Suppose that we want our Monte Carlo estimate to be "sufficiently precise". Ideally, we would like to keep on sampling until our estimate differs very little from the one at the point before. We thus need to set a degree of tolerance, and use a `while` loop.

```
> compute_mean_norm_MC = function(tolerance = 1e-8, maxIter = 1e5){
+
+   # Step 1 - inizialize
+   MC_estimate = rnorm(n = 1, mean = 0, sd = 1)
```

```
+   diff = 1
+   n = 1
+
+   # Step 2 - start the loop
+   while(diff>tolerance & n < maxIter){
+
+     new_value = rnorm(n = 1, mean = 0, sd = 1)
+     MC_estimate_updated = (n*MC_estimate + new_value)/(n+1)
+     n = n+1
+
+     # Update the difference
+     diff = abs(MC_estimate_updated - MC_estimate/MC_estimate)
+
+     # Update the MC estimate
+     MC_estimate = MC_estimate_updated
+   }
+
+   if(diff > tolerance & n >= maxIter) {
+     print ("convergence criterion has not been met")
+   }
+   else {
+     return(list("MC_estimate" = MC_estimate, "iterations" = n, "diff" = diff))
+   }
+   return(list("MC_estimate" = MC_estimate, "iterations" = n, "diff" = diff))
+ }
>
```

Try it out!

```
> compute_mean_norm_MC()

[1] "convergence criterion has not been met"
$MC_estimate
[1] -0.00258958

$iterations
[1] 1e+05

$diff
[1] 1.00259
```

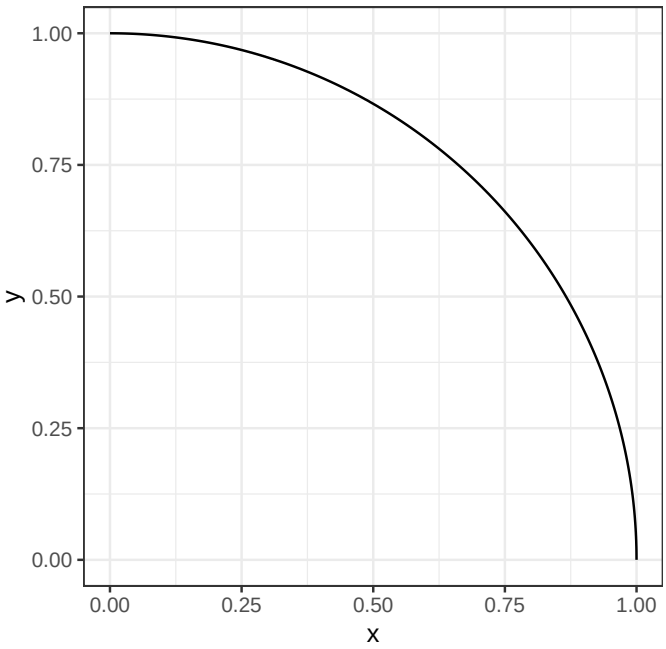
Task:

In the function above we don't know if the `while` loop exited because we reached convergence or we reached the maximum number of iterations. Update the function to provide a warning message if the convergence criterion has not been met.

The value of π

One application of Monte Carlo methods is the computation of the value for π. How? Let's look at the following plot:

```
> # Set the axis
> x = seq(from = 0,to = 1, by=0.0001)
> y = sqrt(1-x^2)
> df_circle = data.frame(x,y)
> # Plot the circle
> ggplot(data = df_circle)+
+   geom_line(aes(x, y))+
+   coord_fixed(ratio = 1)
```



The ratio of the two areas (the square and the circle) is

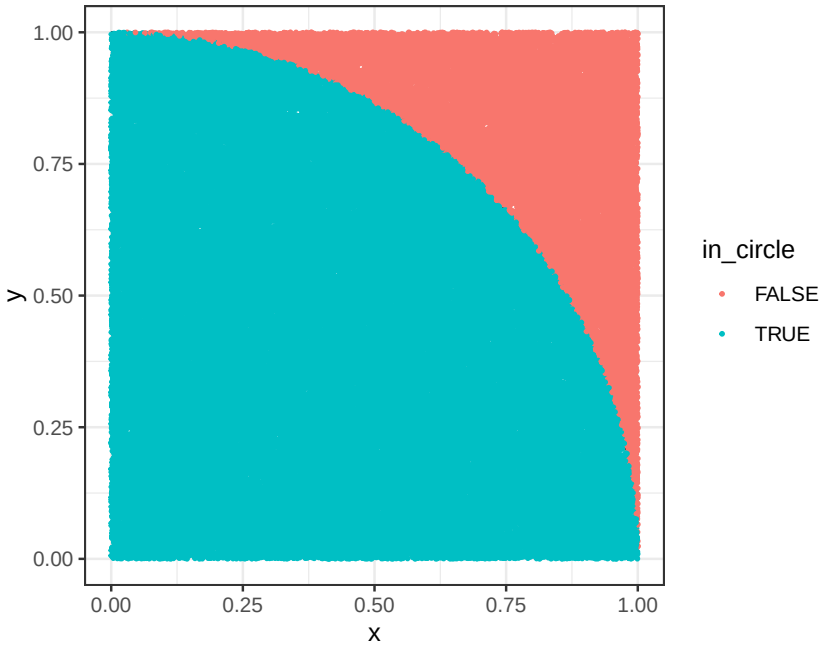
$$\frac{A_{circle}}{A_{square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

If we are able to compute the values for the two areas, then taking their ratio and multiplying it by 4 should give the value for π. How can we compute the areas? Monte Carlo!

```
> # Sample randomly from a bivariate uniform distribution.
> dims = 2
> nSamples = 100000
> uniform_samples = matrix(runif(dims*nSamples), ncol = dims, byrow=TRUE)
> df_points = data.frame(uniform_samples)
> colnames(df_points) = c("x", "y")
> # Plot the samples
> ggplot(data = df_points)+
+   geom_point(aes(x, y), size = 0.5)+
+   coord_fixed(ratio = 1)

> # see which points fall in the area of the circle
> df_points = df_points %>%
+   mutate("in_circle" = case_when(x^2 + y^2 <=1 ~ TRUE,
+                                  TRUE ~ FALSE))
```

```
> ggplot()+
+   geom_line(data = df_circle, aes(x=x, y = y))+
+   geom_point(data = df_points, aes(x=x, y = y, color = in_circle), size = 0.5) +
+   coord_fixed(ratio = 1)
>
```




```
> # Finally, get an estimate for pi!
> 4* sum(df_points$in_circle) / nSamples

[1] 3.15324
```