STAT 33B Homework 1

Ming Fong (3035619833)

Sep 10, 2020

This homework is due **Sep 10, 2020** by 11:59pm PT.

Homeworks are graded for correctness.

As you work, write your answers in this notebook. Answer questions with complete sentences, and put code in code chunks. You can make as many new code chunks as you like.

Please do not delete the exercises already in this notebook, because it may interfere with our grading tools.

You need to submit your work in two places:

- Submit this Rmd file with your edits on bCourses.
- Knit and submit the generated PDF file on Gradescope.

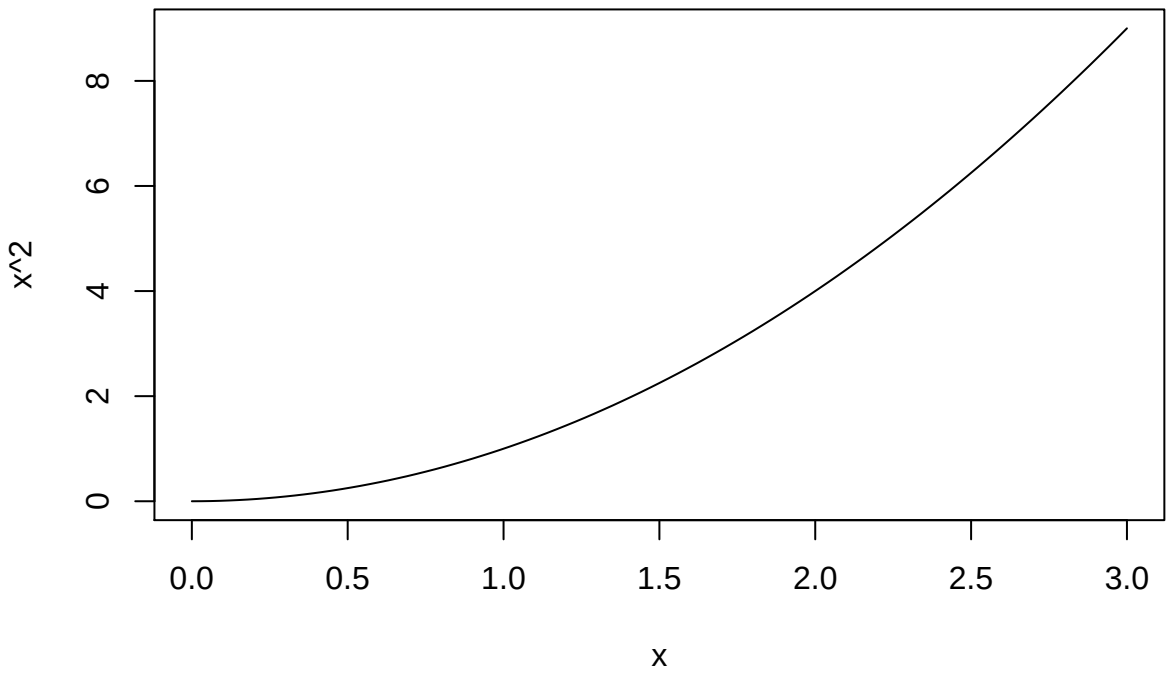
Exercise 1

Note: In R Markdown formatting, a dollar sign `$` marks the beginning of a (LaTeX) mathematical expression. For example, to format a linear equation nicely, you can write $ax + b = 0$. You don't need to know LaTeX for this class, so if you don't understand something written inside `$`, knit the PDF and read that instead.

The `curve()` function plots a curve based on an expression. The basic syntax is `curve(expr, from, to)` where `from` and `to` are the limits of the x-axis.

For example, to plot x^2 between 0 and 3:

```
curve(x^2, 0, 3)
```



The `curve()` function makes it possible to use R as a graphing calculator.

For each expression below:

1. Plot the expression so that all points where it is 0 are visible. Experiment to find appropriate limits for the x-axis.
2. At which point(s) is the expression zero? Estimate (within 1 unit) based on your plot rather than computing these mathematically.

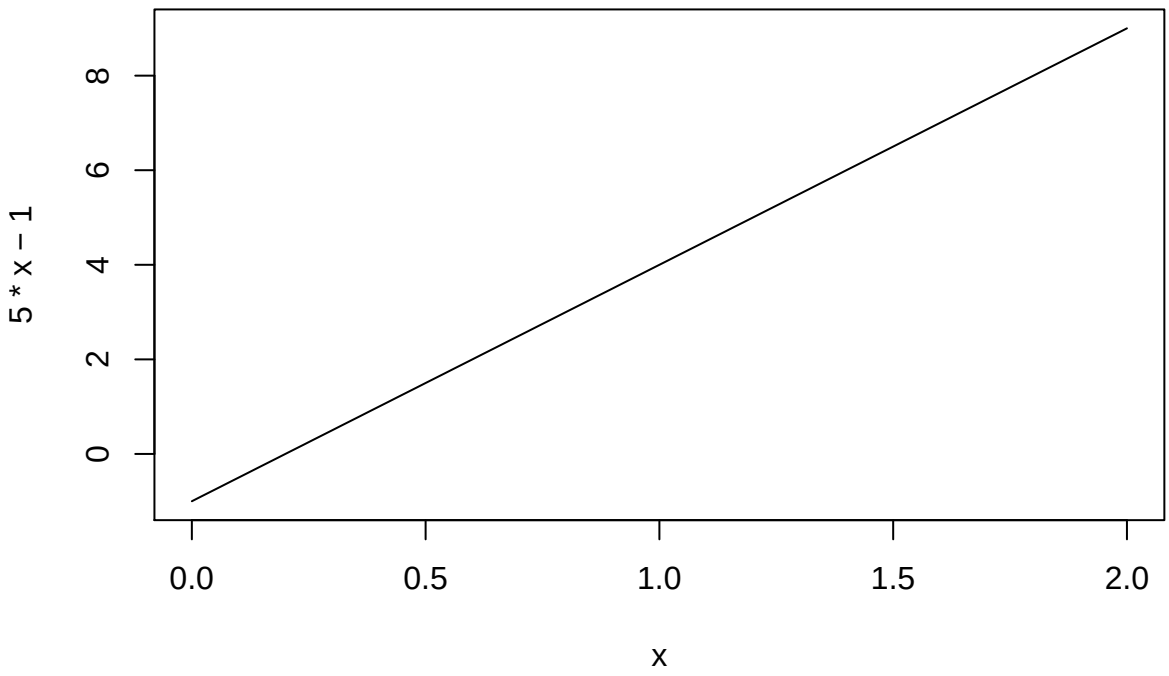
Here are the expressions:

1. $5x - 1$
2. $3x^2 - 2x - 8$
3. $\sqrt{3x} - 4$

YOUR ANSWER GOES HERE:

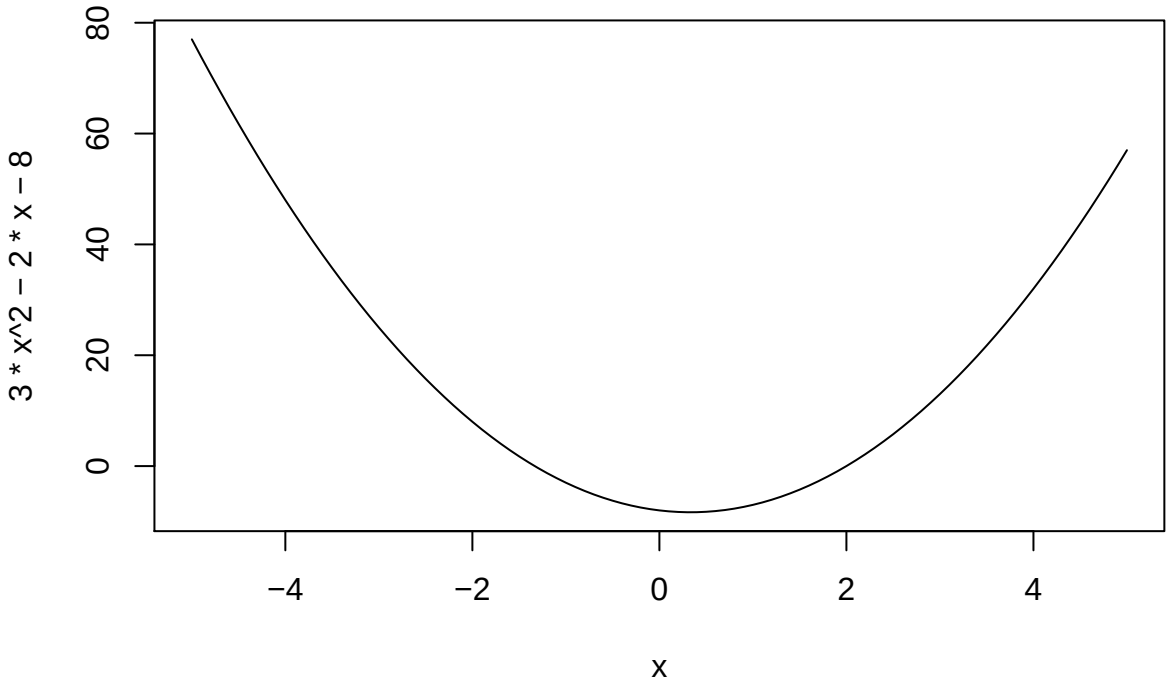
1. Zero at approximately $x = 0.2$

```
curve(5*x - 1, 0, 2)
```



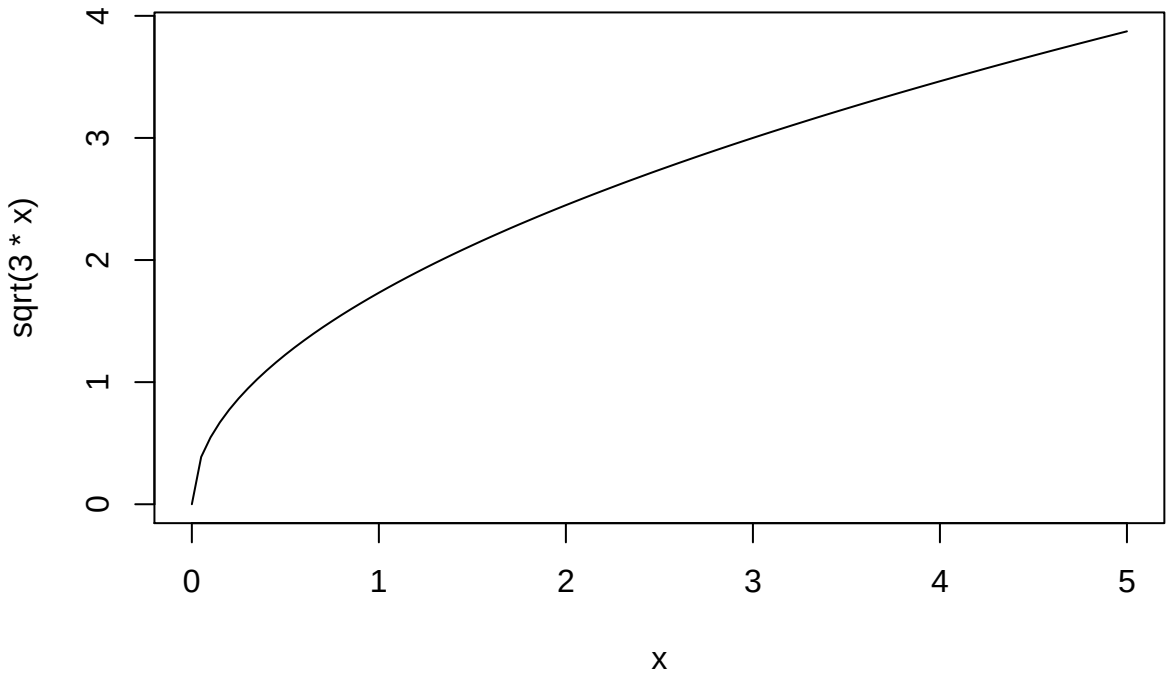
2. Zeros at approximately $x = -1.5, 2$

```
curve(3*x^2 - 2*x - 8, -5, 5)
```



3. Zero at approximately $x = 0$

```
curve(sqrt(3*x), 0, 5)
```



Exercise 2

A **discrete probability distribution** is a table of mutually exclusive outcomes and their associated probabilities.

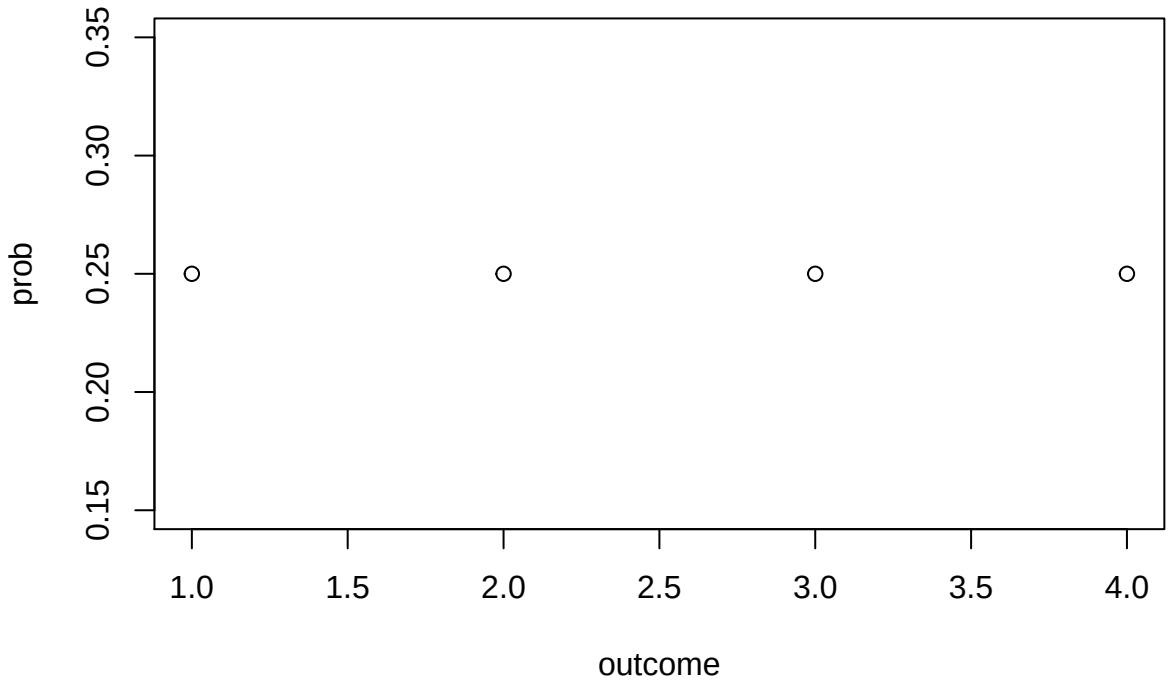
For example, the **discrete uniform distribution** assigns equal probability to each outcome. So the discrete uniform distribution on the integers 1-4 is:

Outcome	Probability
1	0.25
2	0.25
3	0.25
4	0.25

A fair coin toss is another instance of a discrete uniform distribution, where each of the two outcomes has probability 0.5.

You can plot the distribution above with the code:

```
outcome = seq(1, 4)
prob = rep(0.25, 4)
plot(outcome, prob)
```



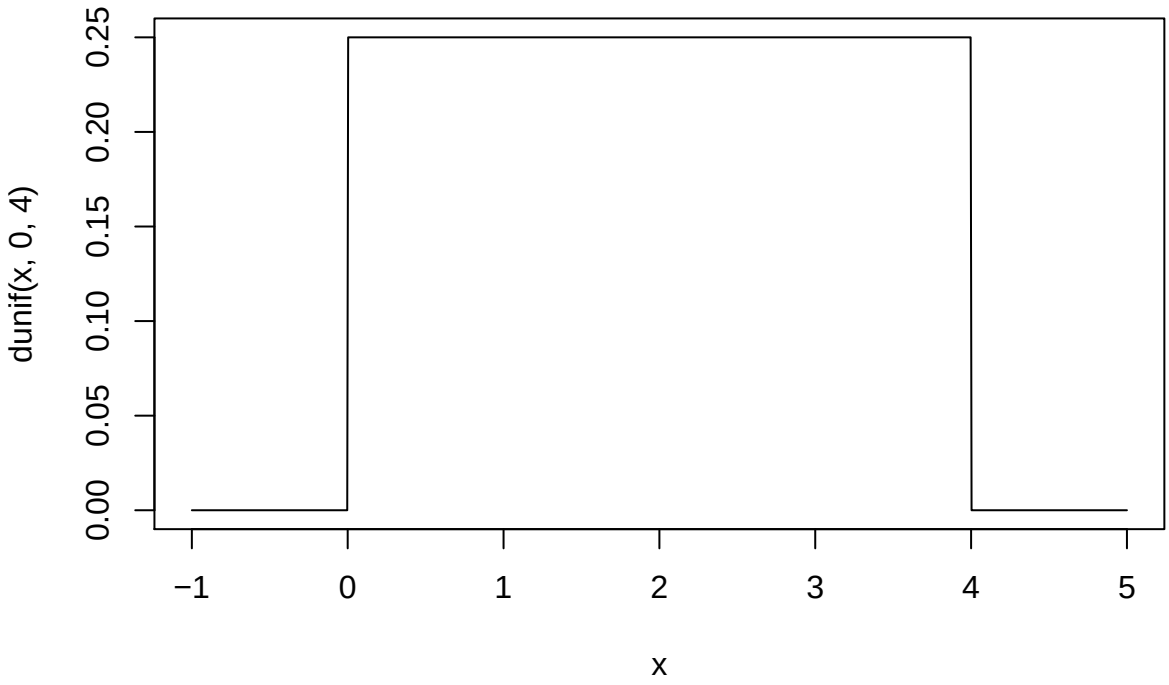
Because the probabilities are all equal, a plot of a uniform distribution is always flat, regardless of the number of outcomes.

The idea of a uniform distribution can also be extended to continuous values. For example, suppose we want to select a decimal value between 0 and 4 (inclusive), with equal probability for any value in the interval. This is a **countinuous uniform distribution**.

As with any uniform distribution, a plot of a continuous uniform distribution is flat. However, instead of discrete points, the distribution is a curve along an interval.

You can plot the continuous uniform distribution from 0 to 4 with the code:

```
curve(dunif(x, 0, 4), -1, 5, n = 1000)
```



A plot of a continuous distribution is called a **density plot**.

In a density plot, the y-axis no longer represents probability. Instead, the probability of points in any subinterval is the total area between the curve and the line $y = 0$ over the subinterval. For instance, in the distribution above, the probability of the outcome being a point between 1 and 2 (inclusive) is 0.25.

R provides functions to compute the density of various distributions. The code above uses the `dunif()` function, the **d**ensity plot function for the **unif**orm distribution. The function has parameters to control the interval for the distribution.

R also provides functions to sample values randomly from various distributions. For example, the `runif()` function samples **r**andomly from a continuous **unif**orm distribution. You can sample 10 values from the continuous uniform distribution from -1 to 1 with the code:

```
runif(10, -1, 1)
```

```
##  [1]  0.28408580 -0.40468391  0.69880883 -0.39815849  0.67531998 -0.05290717
##  [7] -0.83179158  0.23749433 -0.34283498 -0.47609900
```

Since these values are randomly sampled, they will be different each time you run the code.

The **Gaussian distribution** or normal distribution is another continuous probability distribution. The distribution plays an important role in mathematics, statistics, and the natural sciences. The density plot for this distribution is often described as a "bell-shaped curve". The Gaussian distribution is typically parameterized by its mean (center) and standard deviation (how spread out it is), rather than the interval it covers.

R provides functions `dnorm()` and `rnorm()` for the Gaussian distribution that are analogous to the `dunif()` and `runif()` functions described above.

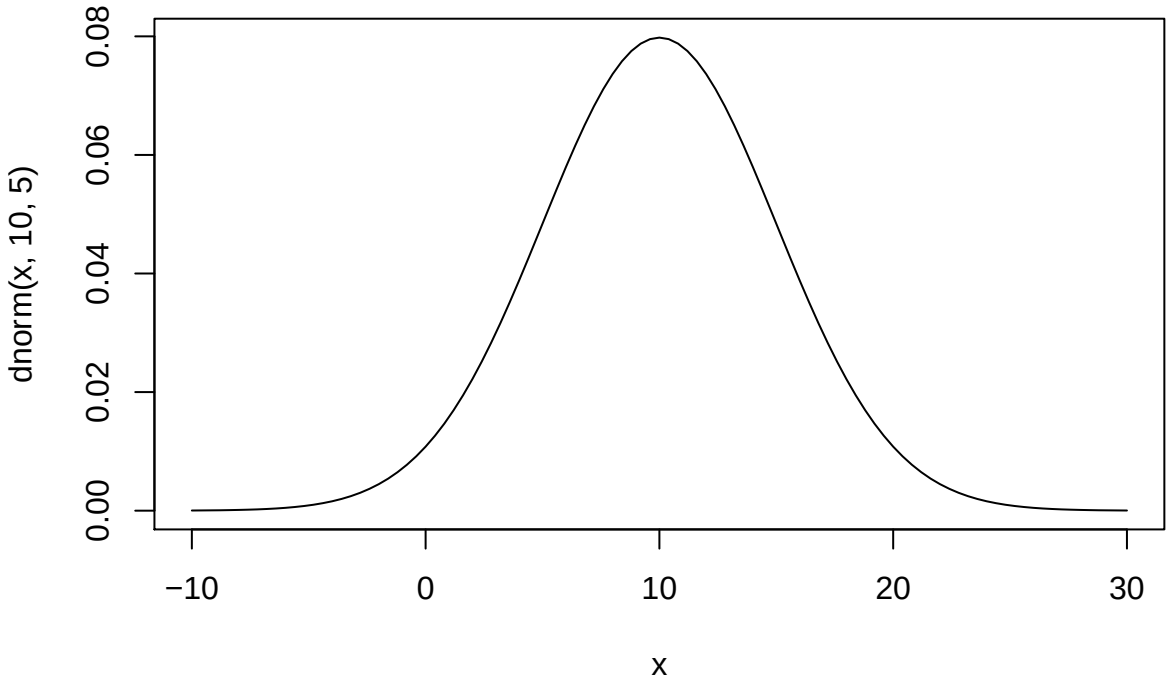
1. Use the `curve()` function to make a density plot of the Gaussian distribution with mean 10 and standard deviation 5. Set the x-axis to span from -10 to 30.
2. Skim the documentation for `rnorm()`. Then create a variable called `samp` that contains 10 random values from a normal distribution with mean 10 and standard deviation 5.
3. Compute the mean and standard deviation of `samp`. Comment briefly on how much these differ from the true mean 10 and true standard deviation 5.

   What happens if you run your code a second time? Are the sample mean and standard deviation the same? Are the sampled points the same?

YOUR ANSWER GOES HERE:

1.

```
curve(dnorm(x, 10, 5), -10, 30)
```



2.

```
mu = 10
sigma = 5
samp = rnorm(10, mu, sigma)
samp
```

```
##  [1]  3.698240  9.852319  8.311835 17.974670 15.403084 13.198894  8.346059
##  [8] 10.876628 15.345122 11.802462
```

3. Running the code many times will change the random sample and also change the sample mean and sd. The difference is computed below.

```r
mu2 = mean(samp)
mu2
```

```
## [1] 11.48093
```

```r
# The mean of the sample differs from the orignal mean by
mu2 - mu
```

```
## [1] 1.480931
```

```r
sigma2 = sd(samp)
sigma2
```

```
## [1] 4.203135
```

```r
# The standard deviation of the sample differs from the orignal standard deviation by
sigma2 - sigma
```

```
## [1] -0.7968648
```

Exercise 3

R's functions for random sampling use a pseudo-random number generator (PRNG). The numbers produced by a PRNG are not truly random, but satisfy conditions that make them close enough to random for most scientific computing tasks.

An advantage of PRNGs is that they are deterministic. Given the same initial parameter, called a **seed**, a PRNG will generate the same sequence of "random" numbers every time. When you start R, the seed is automatically assigned value from your computer that varies (e.g., the system time in milliseconds). This ensures that you get a different sequence of random numbers in each R session.

You can the PRNG seed with the `set.seed()` function. Setting the seed makes your results reproducible. That is, other people can run your code and get the same results even if the code generates random numbers.

Generally, you should only set the seed once at the beginning of an R script or notebook, because it affects all subsequent calls that use the PRNG.

1. Set the seed to 93.
2. Create a variable called `samp` that contains 1000 random values from a normal distribution with mean 10 and standard deviation 5.
3. Compute the mean and standard deviation of `samp`.

   What happens if you run your code a second time? Are the sample mean and standard deviation the same? Are the sampled points the same? Explain how step 1 affects this.
4. Compute the number of values in `samp` that are greater than 8. *Hint: there is a fast, simple way to count `TRUE` values using implicit coercion.*

YOUR ANSWER GOES HERE:

1.

```r
set.seed(93)
```

2.

```r
samp = rnorm(1000, 10, 5)
```

3. If you set the seed and run the code again, the sampled points are always the same. `set.seed()` ensures the pseudo-random numbers used in `rnorm()` are the same each time.

```r
mean(samp)
```

```
## [1] 10.1505
```

```r
sd(samp)
```

```
## [1] 5.127733
```

4.

```r
sum(samp > 8)
```

```
## [1] 668
```

Exercise 4

A **Monte Carlo algorithm** is an algorithm that uses random number generation to approximate a non-random result.

One application of Monte Carlo algorithms is to approximate the area of a shape. In this exercise, you'll use a Monte Carlo algorithm to approximate the area of a circle, and thereby approximate pi.

Consider the circle centered at 0 with radius 1. Points on the circle satisfy $x^2 + y^2 = 1$. This is the "target shape" for which you'll approximate the area. In practice, the target shape is usually one whose area is difficult to compute by hand.

In order to approximate the area of the target shape, start by finding a square or rectangle that completely encloses the target shape. For instance, we can use the square with corners (-1, -1) and (1, 1) to enclose the circle.

Next, randomly sample x coordinates and y coordinates uniformly and independently over the square.

Now the proportion of sampled points inside the circle corresponds to the ratio between the area of the circle and the area of the square. Since you know the area of the square, you can use this relationship to compute the area of the circle.

1. Use R and the described algorithm to approximate the value of pi. Use a variable called `n` to control the sample size.

   Do not use if-statements or loops in your solution; they are not necessary. Try to take advantage of vectorization and implicit coercion. A typical solution will be 2-6 lines.

2. Test your code for different values of `n` from 10 to 1,000,000. How does changing `n` affect the accuracy of the approximation? How does it affect the time it takes to compute the approximation?

   Hint 1: R supports exponent notation, so you can write 1,000,000 in R as `1e6`.

   Hint 2: The `microbenchmark` package provides a simple function called `microbenchmark()` for timing expressions. You can use it to time an entire block of code by surrounding the code with curly braces `{ }`.

YOUR ANSWER GOES HERE:

1.

```r
monte_carlo = function(n = 1000) {
   x = runif(n, -1, 1)
   y = runif(n, -1, 1)
   (sum(x^2 + y^2 < 1) / n) * 4
}
monte_carlo()
```

```
## [1] 3.128
```

2. The higher the value of `n`, the more accurate the approximation. The runtime of the Monte Carlo function seems to scale linearly with `n`, but only for sufficiently large values of `n`.

```
library(microbenchmark)
monte_carlo(10)
```

```
## [1] 2.8
```

```
microbenchmark(monte_carlo(10))
```

```
## Unit: microseconds
##             expr   min lq    mean median    uq    max neval
##  monte_carlo(10) 7.401  8 8.91297  8.301 8.901 56.101   100
```

```
monte_carlo(100000)
```

```
## [1] 3.1378
```

```
microbenchmark(monte_carlo(100000))
```

```
## Unit: milliseconds
##                expr      min       lq     mean   median     uq     max neval
##  monte_carlo(1e+05) 7.985401 8.509902 9.181448 8.703351 8.9264 14.5616   100
```

```
monte_carlo(1e6)
```

```
## [1] 3.141948
```

```
microbenchmark(monte_carlo(1e6))
```

```
## Unit: milliseconds
##                expr     min       lq     mean   median      uq      max neval
##  monte_carlo(1e+06) 77.0869 79.24945 82.78038 82.81635 84.5489 116.6441   100
```