STAT 33B Lab 4

Gunnar Mayer (3034535154)

Edit this file, knit to PDF, and:

- Submit the Rmd file on bCourses.
- Submit the PDF file on Gradescope.

If you think you'll need help with submission, please ask during the lab.

Answer all 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.

As you work, you may find it helpful to be able to run your code. You can run a single line of code by pressing Ctrl + Enter. You can run an entire code chunk by clicking on the green arrow in the upper right corner of the code chunk.

Knit the document from time to time to make sure that your code runs without errors from top to bottom in a fresh R environment.

Monte Carlo Integration

In this lab, you'll practice writing code that uses loops, and learn how to time or "benchmark" your code in order to compare the performance of different implementations.

Monte Carlo integration is a statistical technique for computing the area (or integral) of an arbitrary shape. The idea of the technique is to draw a square around the shape and randomly, uniformly sample points inside the square. Then the ratio of total points inside the shape to total points inside the square (that is, all points) is the same as the ratio of the shape's area to the square's area. This makes it possible to compute the area of the shape.

For example, suppose we want to use Monte Carlo integration to find the area of the unit circle centered at the origin. We can enclose the circle with a square that has opposite corners (-1, -1) and (1, 1). The area of this square is 4. If we sample `n` points, then we can compute the area of the circle as:

```
4 * (number of points in circle) / n
```

Since the technique is random, it will produce a different result with each run. However, the larger the total number of sampled points `n` is, the closer the result will be to the true area of the shape.

In R, you can use the `runif()` function to generate random uniform samples. For instance, to sample one value between -2 and 10, you would write `runif(1, -2, 10)`. You can use `runif()` to sample points for Monte Carlo integration by sampling the x and y coordinates separately.

Exercise 1

Implement Monte Carlo integration for the unit circle example using `runif()` and a for-loop. Specifically, each iteration of your loop should:

1. Sample an x coordinate.
2. Sample a y coordinate.
3. Test whether the sampled point falls inside the unit circle (use the equation for a circle).
4. Update a running total of the number of points that fell inside the unit circle.

After the loop, use the formula above to compute the area of the circle. Test your code for 1 million points, which should be sufficient to compute pi to at least 2 digits.

```
# Your code goes here.
n = 1000000
inside_tot = 0
for(i in 1:n) {
  x = runif(1, -1, 1)
  y = runif(1, -1, 1)
  if(sqrt(x**2 + y**2) < 1) {
    inside_tot = inside_tot + 1
  }
}
pi = 4 * inside_tot / n
pi
```

```
## [1] 3.141232
```

Exercise 2

In Exercise 1, you may have noticed that it takes seconds or even minutes (depending on your code and computer) to compute the result using 1 million points. Vectorization is generally faster than for-loops, and all of the steps in the Monte Carlo integration can be vectorized.

Implement Monte Carlo integration for the unit circle example using vectorized operations (without a for-loop). Note that:

- The `runif()` function can sample more than one value at a time.
- Testing whether a point falls inside the unit circle is arithmetic. All of R's arithmetic operators are vectorized.
- The `sum()` function can compute the number of `TRUE` values in a vector.

Once again, test your code for 1 million points.

```
n = 100000

x = runif(n, -1, 1)
y = runif(n, -1, 1)
inside_pt = sqrt(x**2 + y**2) < 1

pie = 4 * sum(inside_pt) / n
pie
```

```
## [1] 3.13924
```

Benchmarking Code

When measuring the performance of a snippet of code, it's important to time the code more than once. Averaging timings from multiple runs ensures that the timings are not biased by other processes running on your computer.

The microbenchmark package provides a function `microbenchmark` that collects multiple timings (with sub-millisecond accuracy) of code automatically. You can install the package with the code:

```
install.packages("microbenchmark")
```

Once the microbenchmark package is installed, you can time an expression by passing it to the `microbenchmark` function:

```
library(microbenchmark)

# How long does it take to sample 100 observations from standard normal
# distribution?
microbenchmark(rnorm(100))
```

```
## Unit: microseconds
##        expr   min     lq    mean median     uq    max neval
##  rnorm(100) 6.934 7.2415 8.44003  7.467 8.0075 75.608   100
```

You can also pass multiple expressions for the `microbenchmark` function to time:

```
# Which is faster, sampling 100 observations from standard normal or from
# standard uniform?
microbenchmark(rnorm(100), runif(100))
```

```
## Unit: microseconds
##        expr   min    lq    mean median    uq    max neval
##  rnorm(100) 6.972 7.488 8.31833  7.904 8.500 24.940   100
##  runif(100) 3.816 4.040 4.75740  4.344 4.791 20.235   100
```

Finally, you can time multiple lines of code by surrounding them in curly braces `{ }`. If you're timing more than one snippet of code, make sure to surround all of them with curly braces `{ }`, since the curly braces themselves add a small amount of extra time to the computation. For example:

```
# Is it slower to sample 100 strings than it is to sample 100 integers?
microbenchmark(
  { # Case 1
    options = c("a", "b", "c")
    sample(options, 100, replace = TRUE)
  },
  { # Case 2
    options = 1:3
    sample(options, 100, replace = TRUE)
  }
)
```

```
## Unit: microseconds
##                                                                                 expr
##  {     options = c("a", "b", "c")     sample(options, 100, replace = TRUE) }
##               {     options = 1:3     sample(options, 100, replace = TRUE) }
##    min     lq    mean median    uq    max neval
##  6.889 7.3340 9.33641 7.7840 8.614 47.459   100
##  6.453 7.1855 8.88220 7.6145 8.475 26.805   100
```

Read the documentation in `?microbenchmark` for more details about using the function.

Exercise 3

Use the `microbenchmark` function to benchmark your code from Exercise 1 and Exercise 2. You can reduce the number of sampled points to 10,000 in the benchmarked code.

Discuss your result in 1-3 sentences. On your computer, how much slower is the for-loop implementation compared to the vectorized implementation?

Make sure your benchmarks are displayed in the notebook you submit.

```
# Your code goes here.
benchmark = microbenchmark(
  {
n = 10000
inside_tot = 0
for(i in 1:n) {
  x = runif(1, -1, 1)
  y = runif(1, -1, 1)
  if(sqrt(x**2 + y**2) < 1) {
    inside_tot = inside_tot + 1
  }
}
pi = 4 * inside_tot / n
pi
},
{
n = 10000
x = runif(n, -1, 1)
y = runif(n, -1, 1)
inside_pt = sqrt(x**2 + y**2) < 1

pie = 4 * sum(inside_pt) / n
pie
})

benchmark
```

```
## Unit: microseconds
##
##  {     n = 10000     inside_tot = 0     for (i in 1:n) {         x = runif(1, -1, 1)         y = runi
##                                                                                                      {
##        min         lq       mean    median        uq       max neval
##  33046.192 39984.5155 41854.4957 41845.498 43701.625 83854.914   100
##    528.204   558.6575   605.2877   582.269   648.017   816.215   100
```

YOUR WRITTEN ANSWER GOES HERE:

On my computer the difference between the mean of the for loop and vectorized operations is 42856.06 microseconds. Vector operations are substantially faster than looping. On average the for loop is 30x slower than the vector implementation, 44315.812 / 1459.752 ~= 30.