Assignment Two

Calder Evans

13 October 2023

Chapter Ten

Exercise One

Write a function that calculates the density function of a Uniform continuous variable on the interval (a, b).The function is defined as

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

a) Write your function without regard for it working with vectors of data. Demonstrate that it works by calling the function with a three times, once where $x < a$, once where $a < x < b$, and finally once where $b < x$.

```
duniform <- function(x, a, b)
  {
    if( a <= x & x <= b)
      {
        fx <- (1 / (b-a))
      }
    else
      {
        fx <- 0
      }
    return(fx)
}
"x < a"
```

```
## [1] "x < a"
```

```
  duniform(1,2,4)
```

```
## [1] 0
```

```
"a < x < b"
```

```
## [1] "a < x < b"
```

```r
duniform(1,0,4)
```

```
## [1] 0.25
```

```r
"b < x"
```

```
## [1] "b < x"
```

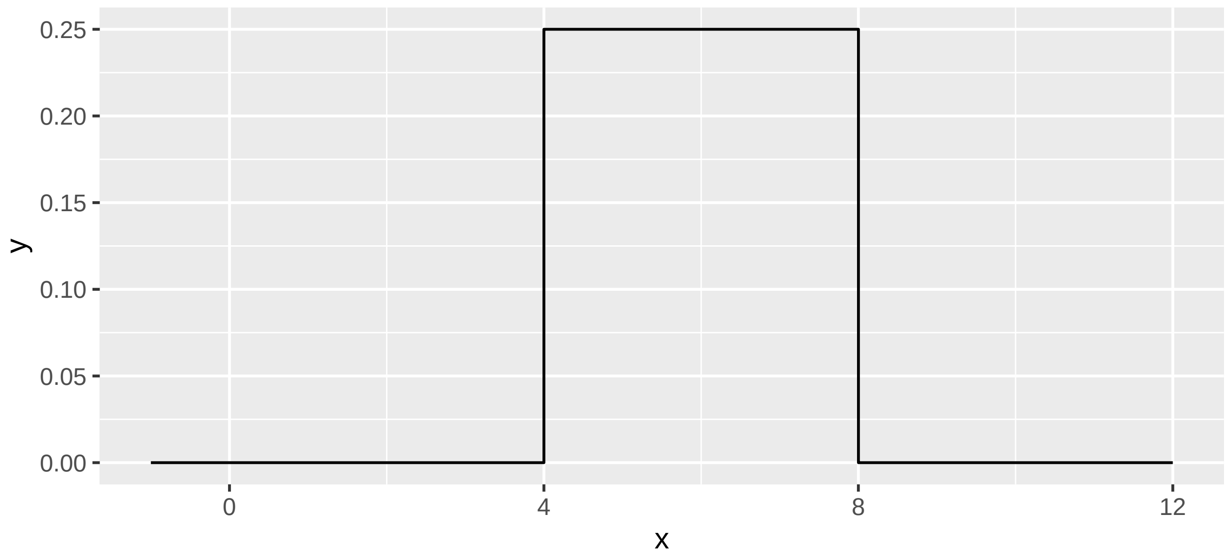
```r
duniform(5,0,4)
```

```
## [1] 0
```

b) Next we force our function to work correctly for a vector of `x` values. Modify your function in part (a) so that the core logic is inside a `for` statement and the loop moves through each element of `x` in succession.

```r
duniform <- function(x, a, b)
  {
    fx <- NULL
    for( i in 1:length(x) )
      {
        if(a <= x[i] & x[i] <= b )
          {
            fx[i] <- (1 / (b-a))
          }
        else
          {
            fx[i] <- 0
          }
      }
    return(fx)
  }
```

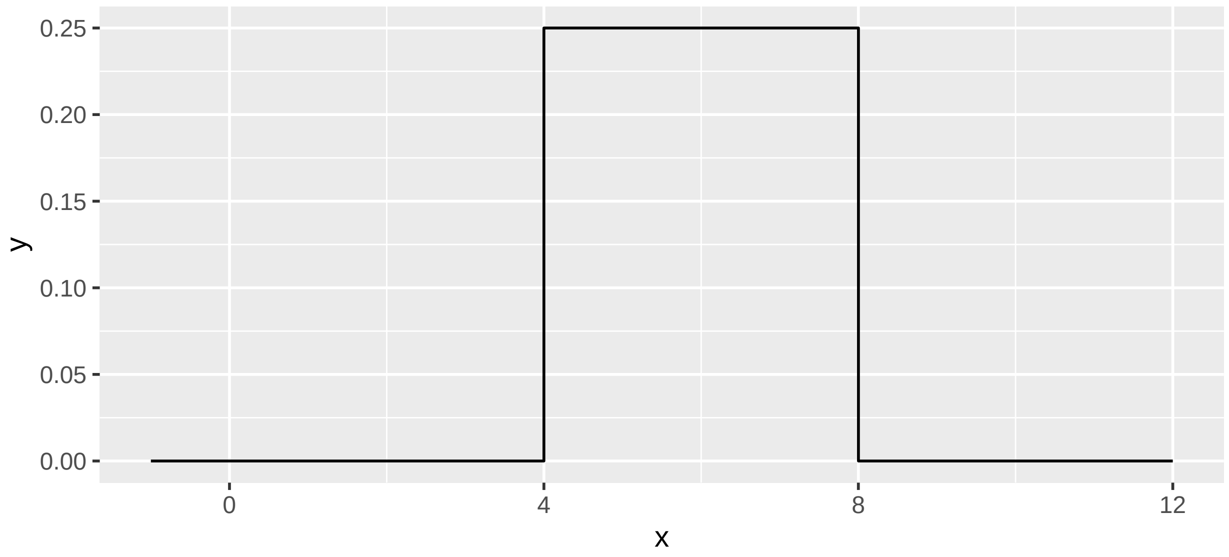
```r
data.frame( x=seq(-1, 12, by=.001) ) %>%
  mutate( y = duniform(x, 4, 8) ) %>%
  ggplot( aes(x=x, y=y) ) +
  geom_step()
```


0.25
0.20
0.15
0.10
0.05
0.00
y
0
4
8
12
x

c) Install the R package `microbenchmark`. We will use this to discover the average duration your function takes.

```
duniform <- function(x, a, b)
  {
    fx <- NULL
    for( i in 1:length(x) )
      {
        if(a <= x[i] & x[i] <= b )
          {
            fx[i] <- (1 / (b-a))
          }
        else
          {
            fx[i] <- 0
          }
      }
    return(fx)
}

data.frame( x=seq(-1, 12, by=.001) ) %>%
  mutate( y = duniform(x, 4, 8) ) %>%
  ggplot( aes(x=x, y=y) ) +
  geom_step()
```



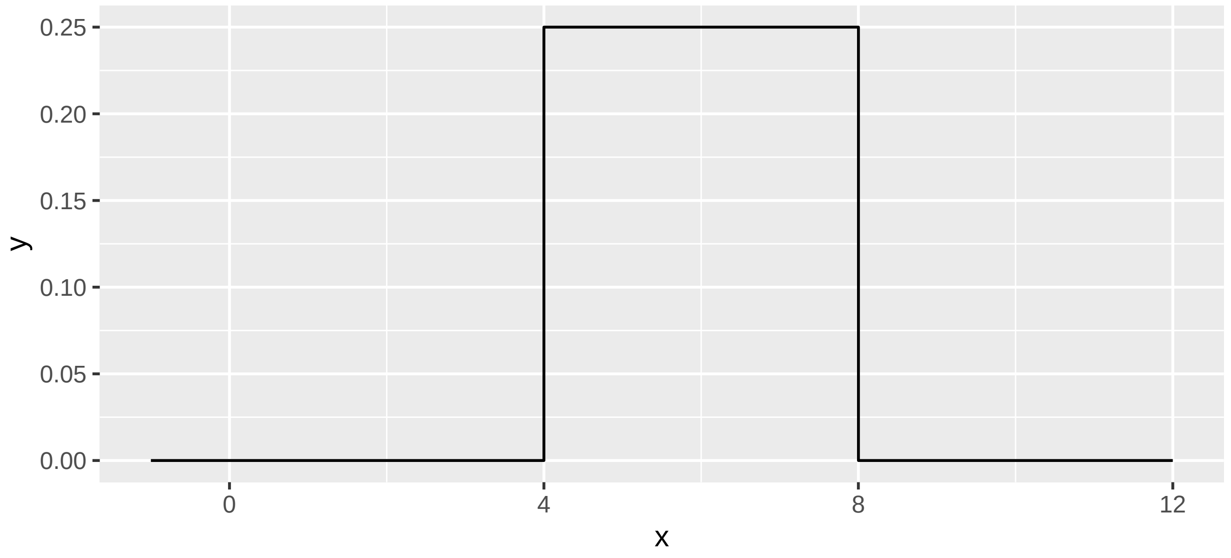
```
microbenchmark::microbenchmark(
  duniform( seq(-4,12,by=.0001), 4, 8), times=100)
```

```
## Unit: milliseconds
##                                      expr     min       lq     mean   median
##  duniform(seq(-4, 12, by = 1e-04), 4, 8) 34.6098 37.35715 40.61706 39.03785
##        uq      max neval
##  41.19925 106.7802   100
```

d) Instead of using a `for` loop, it might have been easier to use an `ifelse()` command. Rewrite your function to avoid the `for` loop and just use an `ifelse()` command. Verify that your function works correctly by producing a plot, and also run the `microbenchmark()`. Which version of your function was easier to write? Which ran faster?

```
duniform <- function(x, a, b)
  {
    fx <- ifelse(a <= x & x <= b, 1/(b-a), 0 )
    return(fx)
}

data.frame( x=seq(-1, 12, by=.001) ) %>%
  mutate( y = duniform(x, 4, 8) ) %>%
  ggplot( aes(x=x, y=y) ) +
  geom_step()
```



```
microbenchmark::microbenchmark(
  duniform( seq(-4,12,by=.0001), 4, 8), times=100)
```

```
## Unit: milliseconds
##                                     expr    min     lq     mean  median      uq
##  duniform(seq(-4, 12, by = 1e-04), 4, 8) 2.5015 2.8116 4.748969 3.66835 5.00905
##      max neval
##  60.2605   100
```

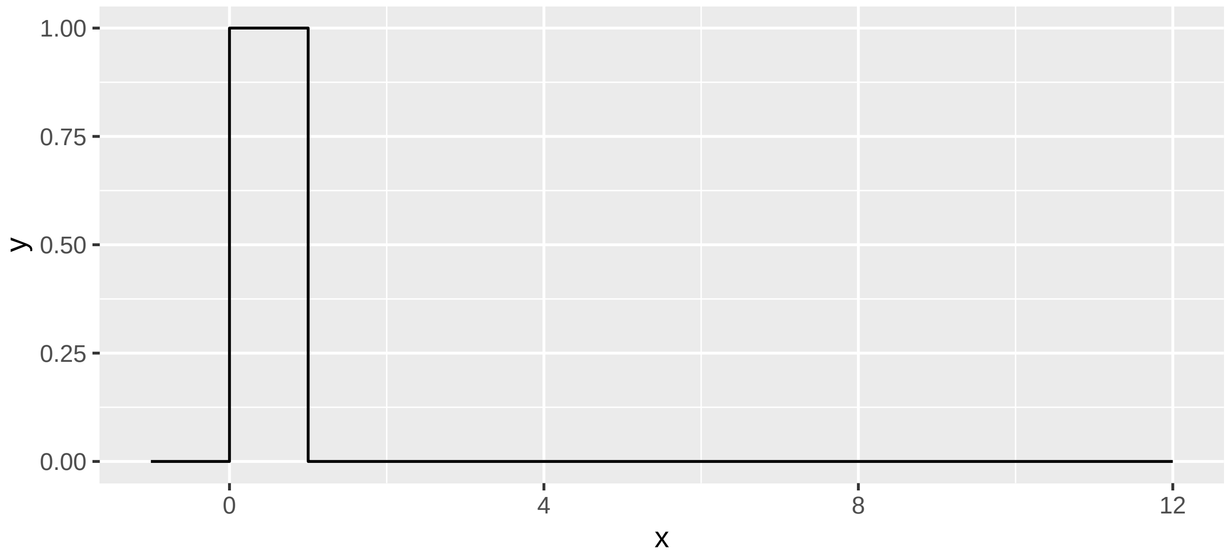
```
(a) The ifelse function was much easier to write and it ran much faster!
```

Exercise Two

I very often want to provide default values to a parameter that I pass to a function. For example, it is so common for me to use the `pnorm()` and `qnorm()` functions on the standard normal, that R will automatically use `mean=0` and `sd=1` parameters unless you tell R otherwise. To get that behavior, we just set the default parameter values in the definition. When the function is called, the user specified value is used, but if none is specified, the defaults are used. Look at the help page for the functions `dunif()`, and notice that there are a number of default parameters. For your `duniform()` function provide default values of `0` and `1` for `a` and `b`. Demonstrate that your function is appropriately using the given default values.

```
duniform <- function(x, a=0, b=1)
  {
    fx <- ifelse(a <= x & x <= b, 1/(b-a), 0 )
    return(fx)
}

data.frame( x=seq(-1, 12, by=.001) ) %>%
  mutate( y = duniform(x) ) %>%
  ggplot( aes(x=x, y=y) ) +
  geom_step()
```



```
microbenchmark::microbenchmark(
  duniform( seq(-4,12,by=.0001)), times=100)
```

```
## Unit: milliseconds
##                                expr  min      lq    mean median     uq     max
##  duniform(seq(-4, 12, by = 1e-04)) 2.52 2.81465 4.67639 3.6152 5.2659 54.0408
##  neval
##    100
```

Exercise Three

A common data processing step is to *standardize* numeric variables by subtracting the mean and dividing by the standard deviation. Mathematically, the standardized value is defined as

$$z = \frac{x - \bar{x}}{s}$$

where $\bar{x}$ is the mean and s is the standard deviation. Create a function that takes an input vector of numerical values and produces an output vector of the standardized values. We will then apply this function to each numeric column in a data frame using the `dplyr::across()` or the `dplyr::mutate_if()` commands. *This is often done in model algorithms that rely on numerical optimization methods to find a solution. By keeping the scales of different predictor covariates the same, the numerical optimization routines generally work better.*

```
standardize <- function(x)
  {
    z <- NULL
    for(i in 1:length(x))
    {
      z[i] <- (x[i] - mean(x)) / sd(x)
    }
  }

data( 'iris' )
# Graph the pre-transformed data.
ggplot(iris, aes(x=Sepal.Length, y=Sepal.Width, color=Species)) +
  geom_point() +
  labs(title='Pre-Transformation')

# Standardize all of the numeric columns
# across() selects columns and applies a function to them
# there column select requires a dplyr column select command such
# as starts_with(), contains(), or where().  The where() command
# allows us to use some logical function on the column to decide
# if the function should be applied or not.
iris.z <- iris %>% mutate( across( where(is.numeric), standardize) )

# Graph the post-transformed data.
ggplot(iris.z, aes(x=Sepal.Length, y=Sepal.Width, color=Species)) +
  geom_point() +
  labs(title='Post-Transformation')
```

(a) Code broke. You told me to just make the code not evaluate.

###Exercise Four In this example, we'll write a function that will output a vector of the first n terms in the child's game *Fizz Buzz*. The goal is to count as high as you can, but for any number evenly divisible by 3, substitute "Fizz" and any number evenly divisible by 5, substitute "Buzz", and if it is divisible by both, substitute "Fizz Buzz". So the sequence will look like 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, … *Hint: The `paste()` function will squish strings together, the remainder operator is `%%` where it is used as `9 %% 3 = 0`. This problem was inspired by a wonderful YouTube video that describes how to write an appropriate loop to do this in JavaScript, but it should be easy enough to interpret what to do in R. I encourage you to try to write your function first before watching the video.*

```
FizzBuzz <- function(n)
  {
    vector <- c(1:n)
    vector1 <- c(1:n)
    for(i in 1:length(vector))
    {
      if(vector[i] %% 15 == 0)
        {
          vector1[i] <- "FizzBuzz"
        }
      else if(vector[i] %% 3 == 0)
        {
          vector1[i] <- "Fizz"
        }
      else if(vector[i] %% 5 == 0)
        {
          vector1[i] <- "Buzz"
        }
    }
  return(vector1)
}
FizzBuzz(100)
```

```
##   [1] "1"        "2"        "Fizz"     "4"        "Buzz"     "Fizz"
##   [7] "7"        "8"        "Fizz"     "Buzz"     "11"       "Fizz"
##  [13] "13"       "14"       "FizzBuzz" "16"       "17"       "Fizz"
##  [19] "19"       "Buzz"     "Fizz"     "22"       "23"       "Fizz"
##  [25] "Buzz"     "26"       "Fizz"     "28"       "29"       "FizzBuzz"
##  [31] "31"       "32"       "Fizz"     "34"       "Buzz"     "Fizz"
##  [37] "37"       "38"       "Fizz"     "Buzz"     "41"       "Fizz"
##  [43] "43"       "44"       "FizzBuzz" "46"       "47"       "Fizz"
##  [49] "49"       "Buzz"     "Fizz"     "52"       "53"       "Fizz"
##  [55] "Buzz"     "56"       "Fizz"     "58"       "59"       "FizzBuzz"
##  [61] "61"       "62"       "Fizz"     "64"       "Buzz"     "Fizz"
##  [67] "67"       "68"       "Fizz"     "Buzz"     "71"       "Fizz"
##  [73] "73"       "74"       "FizzBuzz" "76"       "77"       "Fizz"
##  [79] "79"       "Buzz"     "Fizz"     "82"       "83"       "Fizz"
##  [85] "Buzz"     "86"       "Fizz"     "88"       "89"       "FizzBuzz"
##  [91] "91"       "92"       "Fizz"     "94"       "Buzz"     "Fizz"
##  [97] "97"       "98"       "Fizz"     "Buzz"
```

Exercise Five

The dplyr::fill() function takes a table column that has missing values and fills them with the most recent non-missing value. For this problem, we will create our own function to do the same.

```
test.vector <- c("A", NA, NA, "b", "c", NA, NA, NA)
myFill <- function(x)
{
  for(i in 1:length(test.vector))
  {
    dplyr::fill(test.vector)
  }
}
```