custom functions

`This section covers the following graphing functions:`

1. rStdnNorm()
2. rChisq()
3. annuityAmt()
4. Eratosthenes()
5. smoother()
6. doublesmoother()
7. mcrnGenerator()

---

1. `rStdNorm()`

An **random standard normal simulator** can be implemented as a function in R as

```
rStdNorm <- function(n) {
        Z<-0
        for (jin 1:12) {
                U <- runif(n,min=-.5,max =.5)
                Z<-Z+U
        }
        return(Z)
}
```

💡 Example Use Case

A trial with 3 values is executed as follows:

```
rStdNorm(3)
## [1] 0.5258927-1.1469172-0.8239253
```

---

2. `rChisq()`

A function which calculates chi-squared random variables on ***k*** degrees of freedom.

```
rChisq <- function(n,k) {
        X<-0
        for (iin 1:k) {
                Z<-rStdNorm(n)
                X<-X + Z^2
        }
        return(X)
}
```

💡 Example Use Case

Atrial with k = 17 degrees of freedom, and 2 values is executed as follows:

```
rChisq(2, 17)
## [1] 26.85696 19.04116
```

---

3. `annuityAmt()`

Suppose payments of ***R*** dollars are deposited annually into a bank account which earns constant interest ***i*** per year.

What is the accumulated value of the account at the end of ***n*** years, supposing deposits are made at the end of each year?

The total amount at the end of ***n*** years is

$$R(1+i)^{n-1} + \cdots + R(1+i) + R$$
$$= R \cdot \frac{(1+i)^n - 1}{i}$$

An R function to calculate the amount of an annuity is

```
#return single value
annuityAmt <- function(n, R, i) {
        R*((1 + i)^n- 1) / i
}
```

```
#return list of values
annuityValues <- function(n,R,i) {
        amount<-R*((1+i)^n-1)/i
        PV<-amount * (1+i)^(-n)
        list(amount=amount,PV=PV)
}
```

💡 Example Use Cases

If $400 is deposited annually for 10 years into an account bearing 5% annual interest, we can calculate the accumulated amount using

```
annuityAmt(10, 400, 0.05)

## [1] 5031.157
```

```
annuityValues(10,400,0.05)
## $amount
## [1] 5031.157
##
## $PV
## [1] 3088.694
```

---

4. `Eratosthenes()`

An R function to implement Eratosthenes sieve for finding prime numbers is

```
Eratosthenes <- function(n) {
# Print all prime numbers up to n (based on the sieve of Eratosthenes)
if (n >= 2) {

        noMultiples <- function(j) sieve[(sieve %% j) != 0]

        sieve <- seq(2, n)

        primes <- c()

        for (i in seq(2, n)) {
                if (any(sieve == i)) {
                        primes <- c(primes, i)
                        sieve <- c(noMultiples(i), i)
                }
        }
        return(primes)
}

else{
        stop("Input value of n should be at least 2.") }
}
```

💡 Example Use Case

```
Eratosthenes(30)
## [1]  2  3  5  7 11 13 17 19 23 29
```

---

5. `smoother()`

A simple way to make predictions from such data is to smooth the scatterplot of the ***y*** values that are plotted against the ***x*** values.

One way to do this is to use moving averages.

In other words, just take averages of ***y*** values that are near each other according to their x values.

Join these averages together to form a curve

smoother() outputs a new data frame consisting of a column of equally spaced x values and a column of corresponding local averages, taking the following arguments:
- **x**: the vector of x values
- **y**: the vector of y values
- **x.min**: a constant which specifies the left boundary of the plotted curve
- **x.max**: a constant which specifies the right boundary of the plotted curve
- **window**: a constant giving the range of x values used to calculate the moving averages

An R function that implements smoother is

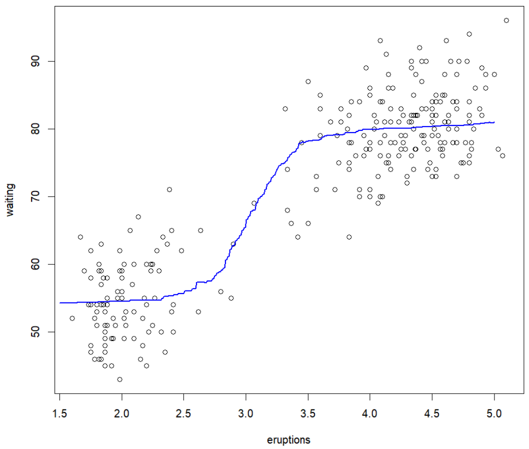
```
smoother <- function(x, y, x.min, x.max, window=1) {
        xpoints <- seq(x.min, x.max, len=401)
        yaverages <- numeric(401)
        for (i in 1:length(xpoints)) {
                indices <- which(abs(x- xpoints[i]) < window)
                if (length(indices) < 1) {
                        stop("Your choice of window width is too small.")
                        # The stop() function provides such a message and aborts
execution of the function.
```

```
                } else {
                        yaverages[i] <- mean(y[indices])
                }
        }

        data.frame(x = xpoints, y = yaverages)
}
```

💡 Example Use Case

```
head(faithful)
##   eruptions waiting
## 1     3.600      79
## 2     1.800      54
## 3     3.333      74
## 4     2.283      62
## 5     4.533      85
## 6     2.883      55
```

```
plot(faithful)

lines(smoother(faithful$eruptions,faithful$waiting, 1.5,5.0,1), col="blue", lwd=2)
```



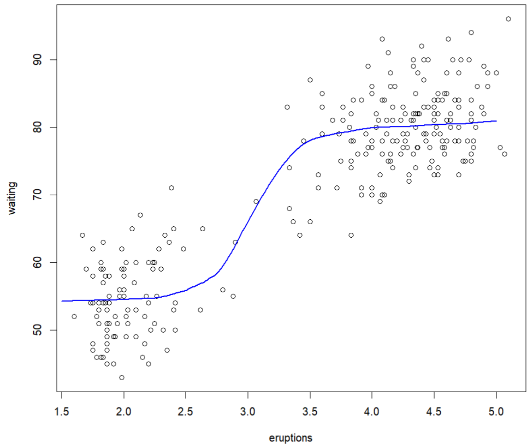
---

6. `doublesmoother()`

```
doublesmoother <- function(x, y, x.min, x.max, window) {
        output1 <- smoother(x, y, x.min, x.max, window[1])
        output2 <- smoother(output1$x, output1$y, x.min, x.max, window[2])

        output2
        }
```

💡 Example Use Case

```
plot(faithful)

ines(doublesmoother(faithful$eruptions,faithful$waiting, 1.5, 5.0,
c(1,0.1)),col="blue",lwd=2)
```



---

7. mcrnGenerator()

An R function that implements a multiplicative congruential (pseudo)random number generator:

```
mcrnGenerator <- function(seed = 6) {
        u <- numeric(1000)
        x <- seed
        for(i in 1:1000) {
            x <- (171 * x) %% 30269
            u[i] <- x/30269
```

```
        }

        return u
        #returns a numeric vector of values between 0 and 1 but not 1.
}
```

💡 Example Use Case

```
> mcrnGenerator()[1]
## [1] 0.03389607
```