StatR 101: Fall 2012
Homework 4 - Solutions
Eli Gurarie, October 23, 2012
Due Sunday, October 21, 6:00 pm

1. From the book, perform exercises 1, 2 and 3 on page 55. The text of the exercises is repeated below:

(a) *Rewrite the Fibonacci* `while()` *loop with the update of* `Fib1` *based just on the current values of* `Fib1` *and* `Fib2`
One solution is to recreate `Fib1` with subtraction within the loop.

```
Fib1 <- 1
Fib2 <- 1
Fibonacci <- Fib2
while (Fib2 < 300) {
  Fibonacci <- c(Fibonacci, Fib2)
  Fib2 <- Fib1 + Fib2
  Fib1 <- Fib2 - Fib1
}
```

(b) *In fact,* `Fib1` *and* `Fib` *aren't necessary either. Rewrite the Fibonacci* `while()` *loop without using any variables except* `Fibonacci`*.*

Here is a strictly one variable method:

```
Fibonacci <- c(1,1)
while (max(Fibonacci) < 300)
  Fibonacci <- c(Fibonacci, sum(Fibonacci[c(length(Fibonacci) - 1, length(Fibonacci))]))
```

giving:

```
[1] 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

It has two problems: First, it is ungainly (note the sequence of parentheses and brackets at the end!) Second, it is hard to stop *before* the sequence hits 300. The following code introduces an extra variable (`i`), that is not strictly necessary but is easier to read. Note that the `while()` loop has no "built in" stopping mechanism, it just waits for the new command `break`.

```
Fibonacci <- c(1,1)
while(1)
{
  i <- length(Fibonacci)
  if( (Fibonacci[i-1] + Fibonacci[i]) < 300)
    Fibonacci <- c(Fibonacci, Fibonacci[i-1] + Fibonacci[i])
```

```
  else(break)
}
```

(c) *Determine the number of Fibonacci numbers less than 1 000 000.*

Any of the loops above will work, as would writing a convenient wrapper function. Note that this function does not return any value, it merely prints a statement answering your question:

```
FibLength <- function(F.max)
{
  Fibonacci <- c(1,1)
  while(1)
  {
    i <- length(Fibonacci)
    if( (Fibonacci[i-1] + Fibonacci[i]) < F.max)
      Fibonacci <- c(Fibonacci, Fibonacci[i-1] + Fibonacci[i])
    else(break)
  }
  print(paste("The number of Fibonacci numbers less than", F.max, "is", length(Fibonacci)))
}
```

Note the implementation:

```
>   FibLength(1e6)
[1] "The number of Fibonacci numbers less than 1e+06 is 30"
> # note the insistence on scientific notation:
>   FibLength(1000000)
[1] "The number of Fibonacci numbers less than 1e+06 is 30"
> # but not here:
>   FibLength(1000001)
[1] "The number of Fibonacci numbers less than 1000001 is 30"
> # note the limit (at least on my computer) of this computation:
>   FibLength(1e308)
[1] "The number of Fibonacci numbers less than 1e+308 is 1475"
>   FibLength(1e309)
[1] "The number of Fibonacci numbers less than Inf is 1476"
```

2. **The Taylor Expansion:**

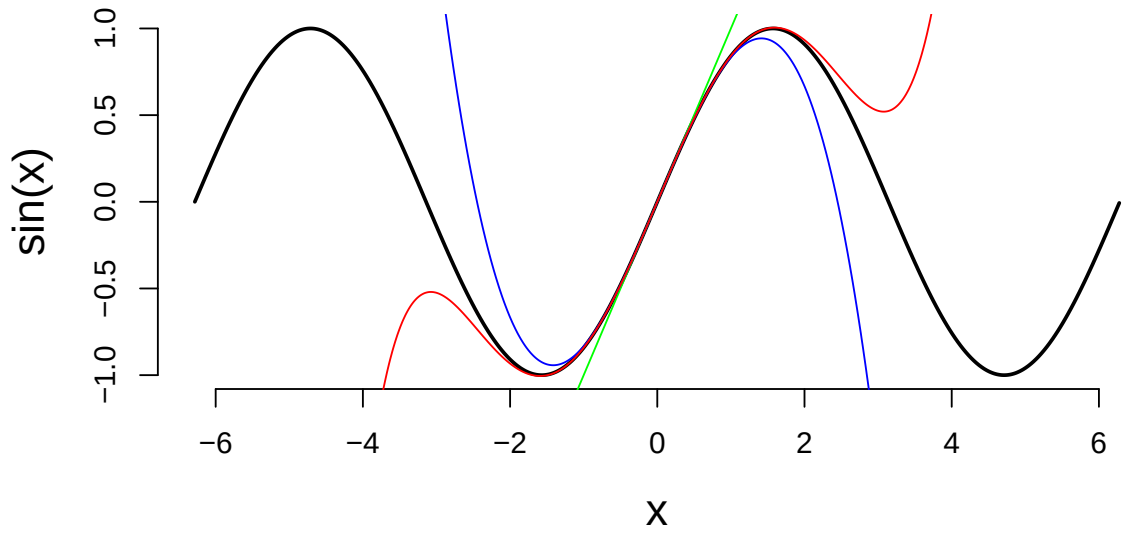
(a) *Building on the code above, illustrate the sine function and its Taylor expansion around 0 up to the 5th term.*

```
x <- seq(-2*pi, 2*pi, .01)
y1 <- x
y3 <- y0 - 1/factorial(3) * x^3
y5 <- y1 +  1/factorial(5) * x^5
```

and use code

```
plot(x, sin(x), type="l", lwd=2)
lines(x, y1, col="green")
lines(x, y3, col="blue")
lines(x, y5, col="red")
```

To produce this plot:



(b) *Write a function* `TaylorSine` *which takes as its arguments values for x and number k and returns the Taylor approximation of sine around 0 to the kth term.*

The trick here is to have the terms calculated only for odd terms, while alternating signs. One way is to first test to see if term `i` is odd, and then flip the sign by taking -1 to the power of a count:

```
TaylorSine <- function(x, k){
  y <- x*0
```

```
  for(i in 1:k)
  if(i%%2 == 1)
    y <- y + (-1)^((i-1)/2) * x^i/factorial(i)
  return(y)}
```

You can also patiently and explicitly change signs with each step

```
TaylorSine <- function(x, k){
  y <- x*0
  sign <- 1
  for(i in 1:k)
  if(i%%2 == 1){
    y <- y + sign * x^i/factorial(i)
    sign <- -sign
  }
  return(y)}
```

Or, for fun, you can cycle through 1,0,-1,0 by obtaining the imaginary component of a complex number that rotates in increments of $\pi/2$ without any `if` statements:

```
TaylorSine <- function(x, k){
  y <- x*0
  for(i in 1:k)
    y <- y + Im(complex(mod=1, arg=pi/2*i)) * x^i/factorial(i)
  return(y)}
```

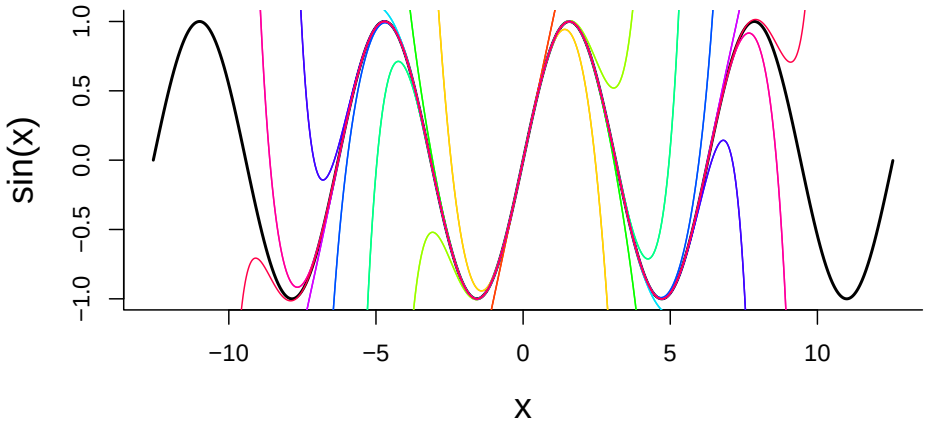
There are other nice solutions (e.g. some of you looped only over odd terms, some built recursive functions that called themselves), but I suspect the one above is most computationally efficient since it contains only one vector and no logical tests.

(c) *Use a loop to illustrate in a single plot the sine function (choosing a larger interval than in part (b)), and the Taylor approximations up to the 20th term, using different colors for each line.*

This code:

```
colors <- rainbow(21)
x <- seq(-4*pi, 4*pi, .01)
plot(x, sin(x), type="l", lwd=2)
for(i in 1:21)
  lines(x, TaylorSine(x, i), col=colors[i])
```

Produces this plot:



3. **Random variables I**

(a) *Simulate 1000 numbers from the standard uniform distribution:* $X \sim Unif(0, 1)$*, and report the sample standard deviation* (s_x) *of these numbers.*

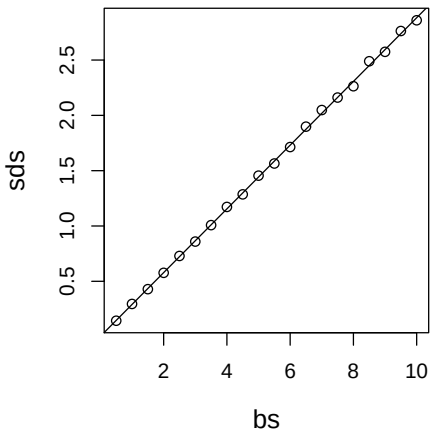
`sd(runif(1000))`. You should get something close to 0.29.

(b) *Write a function that returns the* s_x *of* n *draws from a* $Unif(0, b)$ *distribution, where* b *is the upper limit.*

```
UnifSD <- function(b, n) sd(runif(n, 0, b))
```

(c) *Obtain the values* s_x *against values of* b *ranging from 1 to 10 (using any large value of* n *of your choosing). Is there a consistent pattern? Can you propose a model that predicts* s_x *with upper limit* b*?*

```
# values of "b"
  bs <- seq(0.5, 10, .5)
# by far the most efficient way to do this is using "apply"
  sds <- apply(matrix(bs), 1, UnifSD, n=1000)
# but you can also use a loop
  sds <- bs*0
  for(i in 1:length(bs))
    sds[i] <- UnifSD(bs[i], n=1000)
# plot this function
  plot(bs, sds)
# Since Week 5, you can perform a linear model
  lm(sds~bs)
# Which gives us a slope coefficient around 0.287
# Draw the regression line (because you can)
  abline(lm(sds~bs))
```



Clearly, the relationship is linear, and the slope is around 0.287. The actual exact value is $\frac{1}{\sqrt{12}}$, for reasons that we will encounter later in the quarter when we introduce the mathematical definition of variance. In the meantime, it should begin to be intuitively clear that the standard deviation of a random variable is directly proportional to the scaling of the random variable.

4. **Random variables II**

(a) *Generate two vectors* `X1` *and* `X2`*, each of which represents 1000 draws from the Unif*$(0, 1)$ *distribution. Obtain the pairwise sum of these vectors (i.e.* `Y2 = X1+X2`*) and plot the density histogram of* `Y`*.*

```
X1 <- runif(100000)
X2 <- runif(100000)
Y2 <- X1 + X2
```

(b) *Guess what the probability distribution function (p.d.f.) of* `Y` *might be, and illustrate it with a line over the histogram.*

The density distribution of the sum of two uniform random variables is a so-called "triangle distribution" that looks like this

$$f(x) = \begin{cases} x & \text{for } 0 < x \leq 1 \\ 2 - x & \text{for } 1 < x \leq 2 \\ 0 & \text{else} \end{cases} \tag{1}$$

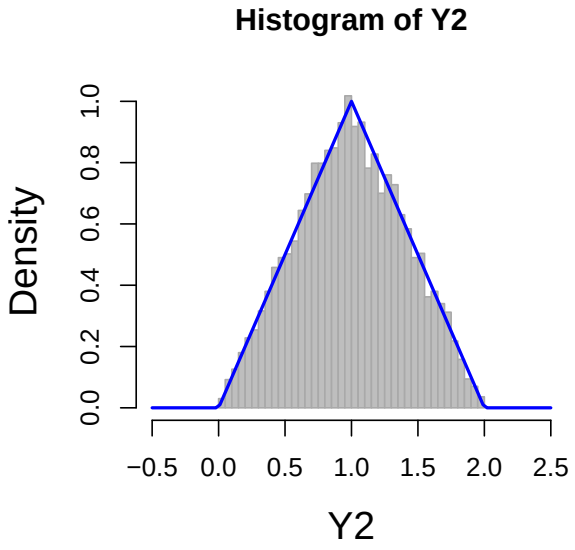
It is a bit tricky to show this formally, but it makes some intuitive sense. First, it is clear that two numbers that are constrained between 0 and 1 can not be less than 0 or greater than 2. Second, it seems like there are more ways to get numbers around 1 than around 0 or 2. In any case, in `R` a discrete case distribution like this one can be generated and illustrated as follows.

```
dtriangle <- function(x)
{
  d <- ifelse(x <= 0, 0,
        ifelse(x <= 1, x,
         ifelse(x <= 2, 2-x, 0)))
  return(d)
}
```

To see the triangleness of the distribution, it helps to illsutrate with a much bigger sample size that 1000. Thus:

```
Y2 <- runif(10000) + runif(10000)
hist(Y2, breaks=20, col="grey", freq=FALSE, xlim=c(-.5, 2.5), bor="darkgrey")
curve(dtriangle, add=TRUE, lwd=2, col=4)
```

Produces



(c) *Similarly obtain* `Y5 = X1 + X2 + ... + X5` *and* `Y10` *(using a loop). Plot on a single 4×2 figure the respective histograms and qqnorm plots of* `X1`, `Y2`, `Y5` *and* `Y10`. *What do you observe?*
This function will generate the sums k times:

```
UnifSums <- function(k, n=1000)
{
  X <- rep(0, n)
```

```
  for(i in 1:k)
    X <- X + runif(n)
  return(X)
}
```

You can use it to quickly compute the requested sums:

```
Y1 <- UnifSums(1)
Y2 <- UnifSums(2)
Y5 <- UnifSums(5)
Y10 <- UnifSums(10)
```

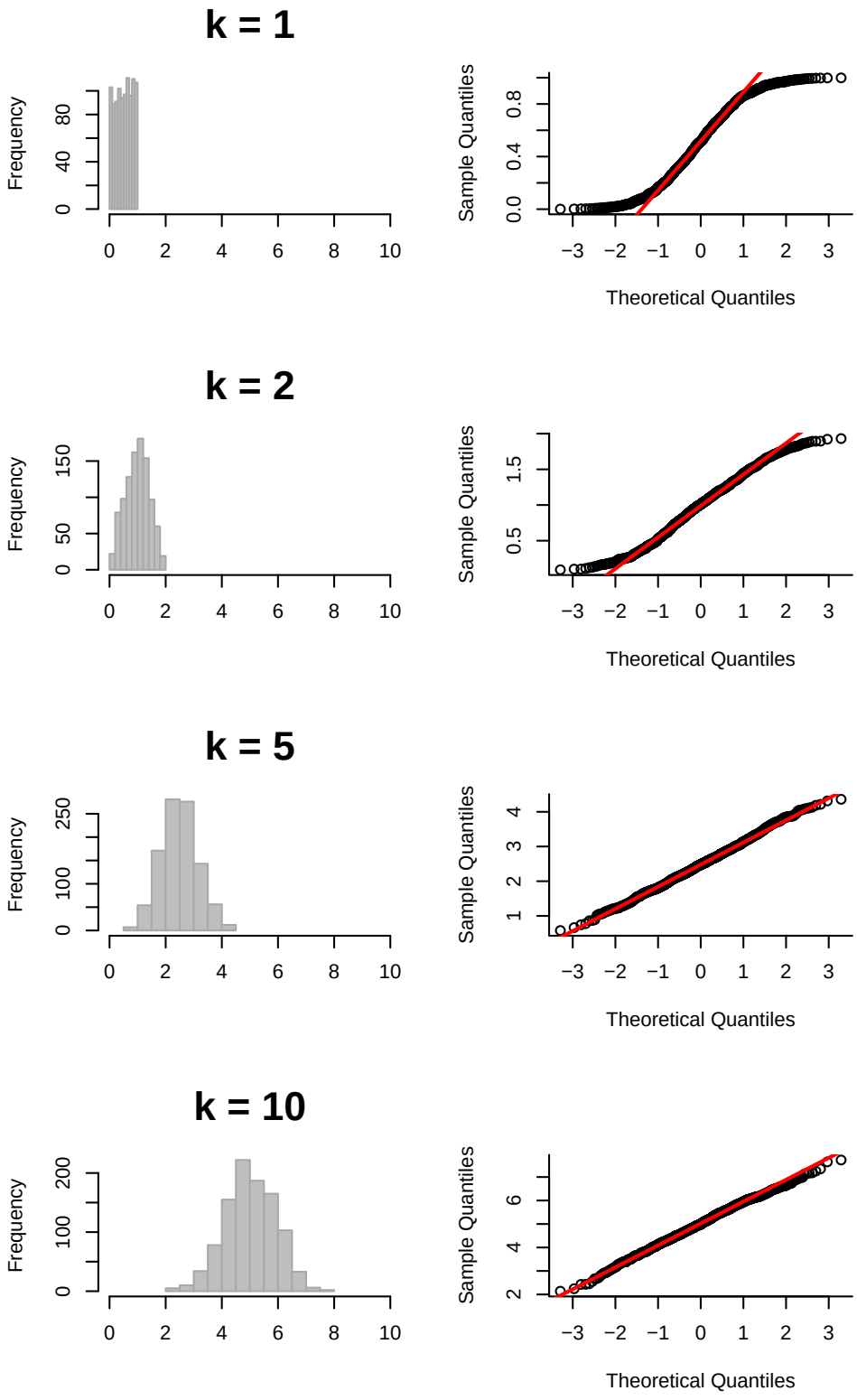
And plot them:

```
par(mfrow=c(4,2), bty="l")
hist(Y1, col="grey",  main="")
qqnorm(Y1, main=""); qqline(Y1, col=2, lwd=2)
hist(Y2, col="grey",  main="")
qqnorm(Y2, main=""); qqline(Y2, col=2, lwd=2)
hist(Y5, col="grey",  main="")
qqnorm(Y5, main=""); qqline(Y5, col=2, lwd=2)
hist(Y10, col="grey",  main="")
qqnorm(Y10, main=""); qqline(Y10, col=2, lwd=2)
```

Personally, even this amount of repetition ($4\times$) feels a bit tedious. If I want to change a graphic parameter, I would have to go change it in each command. The following is code that plots a sum of random uniform variables, and produces the plot. Note how I then plot them with a loop over the desired values:

```
PlotUnifSums <- function(k, n=10000)
{
  Y <- UnifSums(k, n)
  hist(Y, col="grey",  bor="darkgrey", main=paste("k =", k), xlab="", xlim=c(0,10))
  qqnorm(Y, main=""); qqline(Y, col=2, lwd=2, breaks=200)
}
for(k in c(1,2,5,10))
  PlotUnifSums(k)
```

The resulting figure is on the following page. As many of you noted, the more random variables you sum, the more Gaussian the distribution of the result. This is one of the most profound facts in probability theory, and one which will motivate much of the inference theory that we will cover later in the class.


k = 1
Frequency
0
40
80
0
2
4
6
8
10
Sample Quantiles
0.0
0.4
0.8
−3
−2
−1
0
1
2
3
Theoretical Quantiles
k = 2
Frequency
0
50
150
Sample Quantiles
0.5
1.5
k = 5
Frequency
0
100
250
Sample Quantiles
1
2
3
4
k = 10
Frequency
0
100
200
Sample Quantiles
2
4
6