1 Functions

In R *functions* are a symbolic representation of an operation to be carried out on variables known as *inputs*. Functions are useful when you plan to apply a certain operation to a range of inputs which would be cumbersome to declare beforehand. The syntax for declaring a function can be best understood with an example:

Suppose we don't know about the `dnorm()` function and want to calculate the standard normal density,

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

for a grid of x values in the interval $(-3, 3)$. Without using a function this can be accomplished by:

```
x <- seq(-3, 3, length=1000)
phi <- rep(1/sqrt(2*pi), 1000)
for(i in 1:1000)
{

  phi[i] <- phi[i]*exp(-(x[i]^2)/2 )

}
```

Now here is the same task performed by using a function:

```
f <- function(x)
{
  return( exp(-(x^2)/2) )
}

x <- seq(-3, 3, length=1000)
phi <- rep(1/sqrt(2*pi), 1000)

for(i in 1:1000)
{

  phi[i] <- phi[i]*f(x[i])

}
```

There is a way to avoid the use of a loop altogether with the function-based approach here by using the `apply` command, which I will mention later. The `return` statement within a function is only used when you want some value returned by the function. Sometimes you just want a function to do something, but not return a value–such functions are used in the examples below.

One important thing to note about functions is that the inputs are *local* variables only accessible within a function call. If there is another variable defined outside the function with the same name as a function input, it can not be recovered within the function. Similarly, variables defined within the function take precedence over the global declaration. However, if there is no variable defined within the function with that same name, then the global value is still recognized:

```
# global variable
x <- 5

# x is also the input name
f <- function(x)
{
  print(x)
}

f(2)
[1] 2

# x is global at first, then local
f <- function(y)
{

  # x hasn't been declared within the function yet
  print(x)

  # x is now local
  x <- y
  print(x)

}

f(2)
[1] 5
[2] 2
```

Looking track of what is local and what is global is a common source of computing errors. Suppose later in the R session above we made a slight error:

```
f <- function(z,y)
{
  return( x + y )
}

# shouldn't this be 2+3=5?
f(2,3)
[1] 8
```

A general way to avoid any confusion is to delete global variables if you're no longer using the `rm` function, and never use local variables that have the same name as global variables. One final remark about scope is that if you alter a global variable within a function, it will only be changed within that function; once you leave that function the value reverts to what it was before:

```
x <- 10

f <- function(y)
{
 print(x)
 x = x + y
 print(x)
}

f(5)
[1] 10
[1] 15

print(x)
[1] 10
```

Suppose we want a function that takes the coefficients a, b, c of a quadratic polynomial

$$ax^2 + bx + c$$

and returns the real roots (if any) of the polynomial. Recall the roots of a quadratic polynomial are given by

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

so the roots are real if and only if the discriminant, $b^2 - 4ac$, is non-negative. The following R function returns the real roots, or a null vector if there are none:

```
get.root <- function(a, b, c)
{

  ## calculate the discriminant
  d <- b^2 - 4*a*c

  if( d < 0 )
  {

    roots <- c()

  } else
  {

    roots <- c( (-b - sqrt(d))/(2*a), (-b + sqrt(d))/(2*a) )

  }
  return(roots)

}
```

As two examples we will calcuate the roots of

$$x^2 + x - 2$$

(which can be factored as $(x-1)(x+2)$, so the roots are 1, -2) and

$$x^2 + 1$$

which only has the roots $\pm\, i$, which are not real.

```
# x^2 + x - 2
get.root(1, 1, -2)
[1] -2  1

# x^2 + 1
get.root(1, 0, 1)
NULL
```

2 The apply command

Frequently you will want to call the same function across a grid of values for the input. One way to do this would be to use a for loop, but in many cases the `apply` command offers a more computationally efficient alternative. The basic syntax is `apply(A, m, f)` where `A` is a matrix or data frame containing the grid of values for your inputs, `f` is the function to be applied, and `m` is 1 if you are applying `f` to the the rows and 2 if columns.

In the following example we have a data matrix that is 200 rows each containing 10 i.i.d. Binomial(50, .6) random variables:

```
A = matrix( rbinom(2000, 50, .6), 200, 10)
```

The following code obtains the third smallest entry in each of the 100000 rows both using a loop and using `apply`:

```
# f to calculate the 3rd smallest entry of a vector x
f <- function(x) sort(x)[3]

# Storage for the function output
OUT <- rep(0, 200)

# Using a loop
for(i in 1:200)
{

   OUT[i] <- f( A[i,] )

}
```

```
# Using apply
OUT <- apply(A, 1, f)
```

In this case apply provides a negligible, if any, improvement in efficiency when compared to the for loop.

Examples where apply provides a noticable improvement are usually ones that would involve nested loops. Suppose we want to calculate $x^2 + y^2$ for order pairs (x, y) in the space $(0, 1) \times (2, 3)$ for a mesh size of .01 in each direction. The following code does this by using nested for loops:

```
f <- function(x) x[1]^2 + x[2]^2
x1 <- seq(0,1,by=.01)
x2 <- seq(1,2,by=.01)
q <- NULL
k <- 0
for(i in 1:101)
{
 for(j in 1:101)
 {
  k=k+1
  q[k] <- f( c(x1[i],x2[j]) )
 }
}

# Q[i,j] now equals f( c(x1[i], x2[j]) )
Q <- t( matrix(q, 101, 101) )
```

The code above takes about 5 seconds to run on most computers, which is about 6 times slower than the same process which uses apply. For this to work we need to first set up a data frame so that operations on the rows are equivalent to the above loop:

```
# V[100*(i-1) + j,] = c(x1[i], x2[j])
n <- NULL
for(i in 1:101) n <- c(n, rep((i-1)/100, 101))
v <- cbind(n, seq(1,2,length=101))

# f defined the same as above
z <- apply(v,1,f)
Z <- t( matrix(z, 101, 101) )
```

The difference in efficiency (about .9 seconds vs. 5 seconds) is not amazing but would be considerably larger if, for example, we used a smaller mesh size and thereby increased the number of calculations required.

3 Disease process example

Consider an extremely simple model for a potentially terminal disease such that there are three states: healthy, infected, and dead. The following rules probabilities govern this disease process:

- If you are healthy you become infected with probability p_i and stay healthy with probability $1 - p_i$.
- If you are infected you die with probability p_d and are equally likely to either remain infected, or become healthy again.
- Once you enter the "dead" state you remain there with probability 1.

It is useful to represent this situation by the *transition matrix*,

$$P = \begin{pmatrix} 1-p_i & p_i & 0 \\ \frac{1-p_d}{2} & \frac{1-p_d}{2} & p_d \\ 0 & 0 & 1 \end{pmatrix}$$

In this matrix 1 represents healthy, 2 is infected, and 3 is dead. Then P_{ij} is the probability of moving from state i to state j. In this example we will write code to simulate from this process and use it to estimate:

- The probability that a randomly selected subject dies by time T
- The expected number of timepoints until death

for given values of p_i, p_d. Once this is done we will make plots to summarize the estimated probabilities.

```
# Simulate the lifespan of a n people starting
# from state 1, given p.i and p.d
rproc <- function(n, p.i, p.d)
{

   ## the transition matrix
   P <- matrix(0, 3, 3)
   P[1,] <- c(1-p.i, p.i, 0)
   P[2,] <- c(.5*(1-p.d), .5*(1-p.d), p.d)
   P[3,] <- c(0, 0, 1)

   ## storage for each person’s lifespan
   X <- list()

   for(j in 1:n)
   {

      x <- c(1)
      k <- 1

      ## while the subject is *not* dead, continue to  simulate their lifespan
      while( x[k] != 3 )
      {

 ## their "random draw" for this timepoint
          u <- runif(1)
          ## the transition probabiilities corresponding to their current sate
          p <- P[x[k],]

          ## while probability p[1] go to state 1
          if( u < p[1] )
          {

             x <- c(x,1)

          ## Since you know they didn't go to state 1
          ## their u must be greater than p[1], so go
          ## to state 2 with probability p[2]
          } else if( u < sum(p[1:2]) )
          {

             x <- c(x,2)

          ## Since they didn't go to state 1 or 2, they
          ## must be in state 3
          } else
          {

             x <- c(x,3)

          }

          k <- k + 1

       }

       ## Store person j's lifespan
       X[[j]] <- x

    }

   return(X)

}

## estimate the probability that a random
## subject dies by time T, and the mean lifetime,
## for given p.i, p.d and sample size n
## Q = [n, p.i, p.d, T]
char.proc <- function(Q)
{

   n <- Q[1]; p.i <- Q[2]; p.d <- Q[3]; T <- Q[4];
   Y <- rproc(n, p.i, p.d)
   L <- rep(0, n)
   for(i in 1:n)
   {

      L[i] <- length( Y[[i]] )

   }

   return( c( mean(L <= T), mean(L) ) )

}

## Apply for fixed sample size n = 200,
## and death time T = 5, for p.i and p.d
## ranging from .1, .3, ..., .9

# Set up the data frame
# G will contain (.1, .3, ..., .9) cross (.1, .3, ..., .9)
pi <- seq(.1, .9, by=.2)
pd <- seq(.1, .9, by=.2)
G <- NULL
for(i in 1:5) G <- c(G, rep(pi[i], 5))
G <- cbind(G, pd)
G <- cbind(rep(200,25), G, rep(5, 25) )


OUT <- apply(G, 1, char.proc)

## Plot probability against p.i for
## each value of p.d on the same plot
# indices of the p.i values for pd = .1, ..., .9
ii1 <- seq(1, 25, by=5)
ii2 <- seq(2, 25, by=5)
ii3 <- seq(3, 25, by=5)
ii4 <- seq(4, 25, by=5)
ii5 <- seq(5, 25, by=5)

# plot prob vs. p.i. for p.d = .1
plot( G[ii1,2], OUT[1,ii1], col=2, ylim=c(0,1), xlab="p.i",
   ylab="Probability of death by time 5", axes=F,
   main="Prob. of death by time 5 vs. p.i for each p.d")
axis(1, seq(.1, .9, length=5), seq(.1, .9, length=5))
axis(2, seq(0, 1, length=6), seq(0, 1, length=6))
box()
lines( G[ii1,2], OUT[1,ii1], col=2)

#plot prob vs. p.i. for p.d = .3
```

```
lines( G[ii2,2], OUT[1,ii2], col=3)
points( G[ii2,2], OUT[1,ii2], col=3)

#plot prob vs. p.i. for p.d = .5
points( G[ii3,2], OUT[1,ii3], col=4)
lines( G[ii3,2], OUT[1,ii3], col=4)

#plot prob vs. p.i. for p.d = .7
points( G[ii4,2], OUT[1,ii4], col=5)
lines( G[ii4,2], OUT[1,ii4], col=5)

#plot prob vs. p.i. for p.d = .9
points( G[ii5,2], OUT[1,ii5], col=6)
lines( G[ii5,2], OUT[1,ii5], col=6)

## Plot mean lifespan against p.i for
## each value of p.d on the same plot
plot( G[ii1,2], OUT[2,ii1], col=2, ylim=c(0,80), xlab="p.i",
   ylab="Expected time until death", axes=F,
   main="Expected time until death vs. p.i for each p.d")
axis(1, seq(.1, .9, length=5), seq(.1, .9, length=5))
axis(2, seq(0, 80, length=5), seq(0, 80, length=5))
box()
lines( G[ii1,2], OUT[2,ii1], col=2)
lines( G[ii2,2], OUT[2,ii2], col=3)
points( G[ii2,2], OUT[2,ii2], col=3)
points( G[ii3,2], OUT[2,ii3], col=4)
lines( G[ii3,2], OUT[2,ii3], col=4)
points( G[ii4,2], OUT[2,ii4], col=5)
lines( G[ii4,2], OUT[2,ii4], col=5)
points( G[ii5,2], OUT[2,ii5], col=6)
lines( G[ii5,2], OUT[2,ii5], col=6)
```

From the plots below it is clear that the probability of death by time 5 increases as a function of both p_i, and p_d, and the mean lifespan decreases.

4 Gambling example

Suppose a person starts with D dollars and bets 1 dollar repeatedly, and they bet until they either go broke, or double their money. We will use simulation to estimate the amount of time a person will play before stopping, and the probability of going broke as a function of p, the probability of a win in a given round. When p is close to 0 or 1, the probability of ruin is basically 0 and 1, respectively, so we only look at $p \in (.4, .6)$.

```
## Simulate a single person's betting history under example 1.
## as a function of the win probability and initial bankroll, D.
## I = c(p, D)
bet.round <- function(I)
```

```
{

   p <- I[1]; D <- I[2];

   # amount you started with
   startval <- D

   # storage for their bankroll at each timepoint
   bank <- NULL

   # counter variable
   k <- 0

   ## break out out when they have no money left
   while( (D > 0) & (D < (2*startval)) )
   {

      ## indicator of whether they won this game
      ind <- rbinom(1, 1, p)

      ## if they won, increase their bankroll by 1
      ## else, decrease it by 1
      if( ind == 1)
      {

         D = D + 1

      } else
      {

         D = D - 1

      }

      k <- k + 1
      bank = c(bank, D)

   }

   ## k is the now the number of turns before they
   ## went broke. bank is their entire history

   ## this will be uncommented later
   ## return both the amount of time until they stopped
   ## and an indicator of whether or not they went broke
   # return(c(k, (D==0)))

   return(bank)
}

# plot an example trajectory with p = .4, D = 50
plot( bet.round( c(.4, 50) ), xlab="Time", ylab="Bankroll",
   main="Random bettor's trajectory", type="l")
```

To estimate the amount of time it takes for the better to go broke **uncomment** `return(c(k, (D==0))` **and comment out** `return(bank)` **in the example code above**. Now the function `bet.round` will just return the amount of time before a person goes broke or doubles their initial bankroll, along with an indicator of whether or not they went broke. We can use this to estimate the probability of ruin and the mean time before you stop betting as a function of p when you start with $D = 10$ dollars:

```
## A function with estimates mean time to going broke based on
## nreps replications. J = c(p, D, nreps)
meantime <- function(J)
{

   p <- J[1]; D <- J[2]; nreps <- J[3];

   ## storage for each output
   m <- matrix(0, nreps, 2)

   ## get nreps samples of the ruin problem
   for(j in 1:nreps) m[j,] <- bet.round( c(p, D) )

   ## return the sample mean of time until
   ## stopping, and the sample proportion
   ## of people who "won"
   return( c(mean(m[,1]), mean(m[,2]) ) )

}


## define a grid of values of prob as store them in
## a 1-by-9 matrix
probs <- as.matrix( seq(.4, .6, length=10), 1, 10)
inputs <- cbind( probs, rep(10, 10), rep(1000, 10) )

## apply go.broke to each value in probs, with D=50, nreps=1000.
times <- apply(inputs, 1, meantime)

## plot the probability of ruin vs. p
plot(probs, times[2,], xlab="Probability of winning a single round",
   ylab="Probability of ruin",
   main="Probability of ruin vs. Probability of winning a single round", col=3)
lines(probs, times[2,], col=3)
```

```
## plot the expected time to ruin/win vs. p
plot(probs, times[1,], xlab="Probability of winning a single round",
   ylab="Expected time until ending",
   main="Probability of ruin vs. Expected time until you quit playing", col=4)
lines(probs, times[1,], col=4)
```

In the probability plot we can see that you are basically guaranteed to go broke if you play a game with win probability $p = .4$. Also, the expected time until you quit playing is maximized when you are playing a fair game.

5 Stopping time example

Without formally defining a stopping time we give an example. Let τ_k be a random variable defined as the first time a sum of $N(0,1)$ random variables exceeds k. That is,

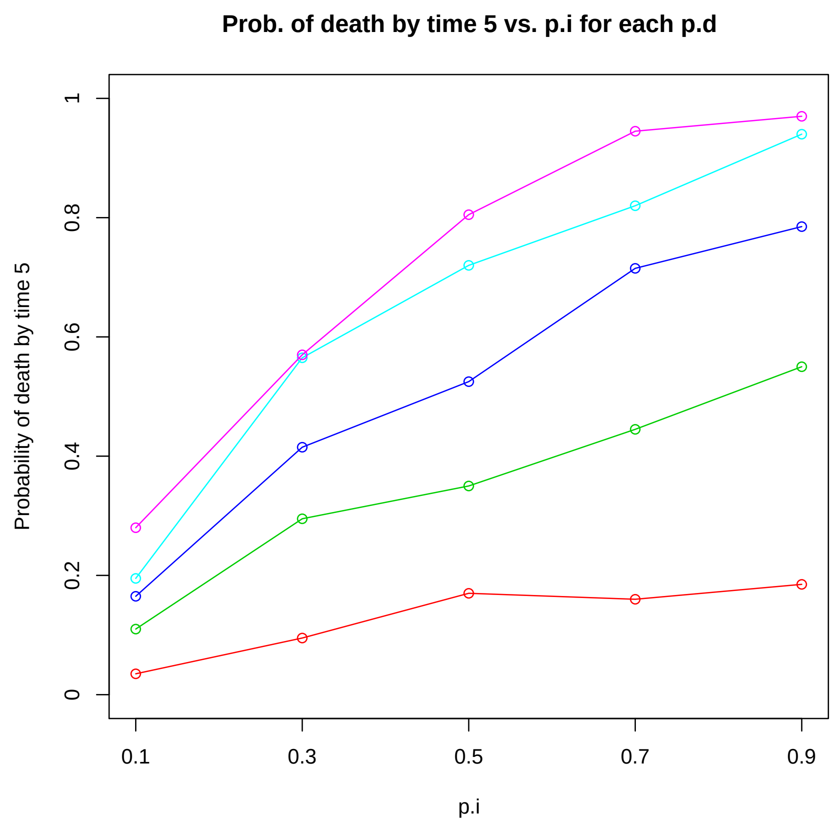
$$\tau_k = \min_i \left(\sum_{j=1}^{i} |X_j| \geq k \right)$$

where the X_j are i.i.d. $N(0,1)$. We will write an R program that generates from the distribution of τ_k, and will use that to estimate its mean and variance:

```
## generate a single realization of tau_k
single.tau <- function(k)
{
   X <- c(0)
   i <- 0

   ## quit the loop when the sum exceeds 1
   while( sum(abs(X)) < k )
   {
      i <- i + 1

      # add another term to X
      X<- c(X, rnorm(1))
   }

   #return the number of terms required to get sum(abs(X)) > 3.
   return(i)
}

## generate nreps replications of tau
rtau <- function(nreps, k)
{

   ## the data frame apply the function to
   B <- as.matrix( rep(k, nreps), 1, nreps )
   ## apply the single.tau
   V <- apply(B, 1, single.tau)

  return(V)

}
```

We can now use this to investigate the distribution of τ_k as a function of k based of 1000 replications for a grid of k values:

```
K <- seq(.5, 5, by=.5)

# storage for the means and std deviations
Q <- matrix(0, 10, 2)
for(j in 1:10)
{

   TAU <- rtau(1000, K[j])
   Q[j,] <- c( mean(TAU), sd(TAU) )

}

par(mfrow=c(2,1))
plot(K, Q[,1], xlab="The cut off point, k",
   ylab="Mean of Tau", main="Mean of tau vs. k", col=2)
lines(K, Q[,1], col=2)

plot(K, Q[,2], xlab="The cut off point, k",
   ylab="S.D. of Tau", main="S.D. of tau vs. k", col=3)
lines(K, Q[,2], col=3)
```



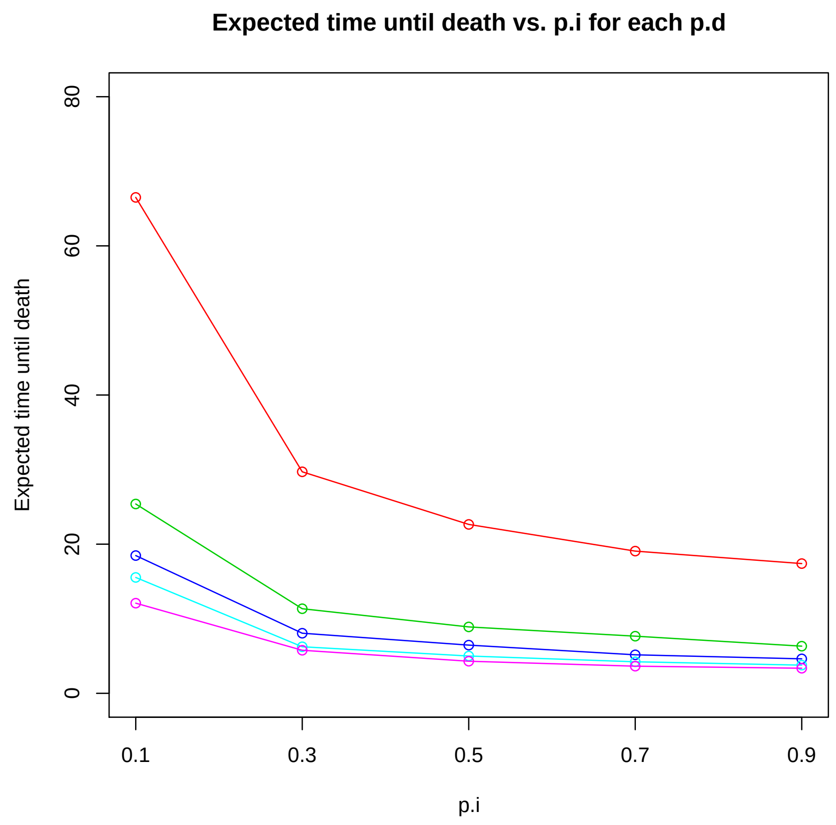
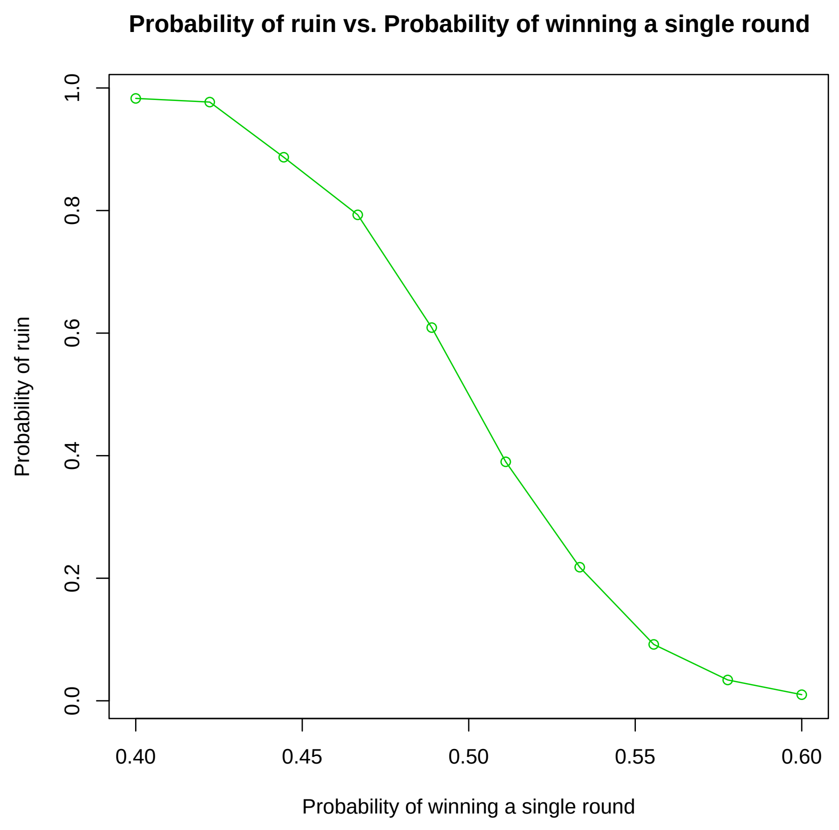


Figure 1: $P(\text{deathtime} \leq 5)$ and expected time to death as a function of p_i, p_d



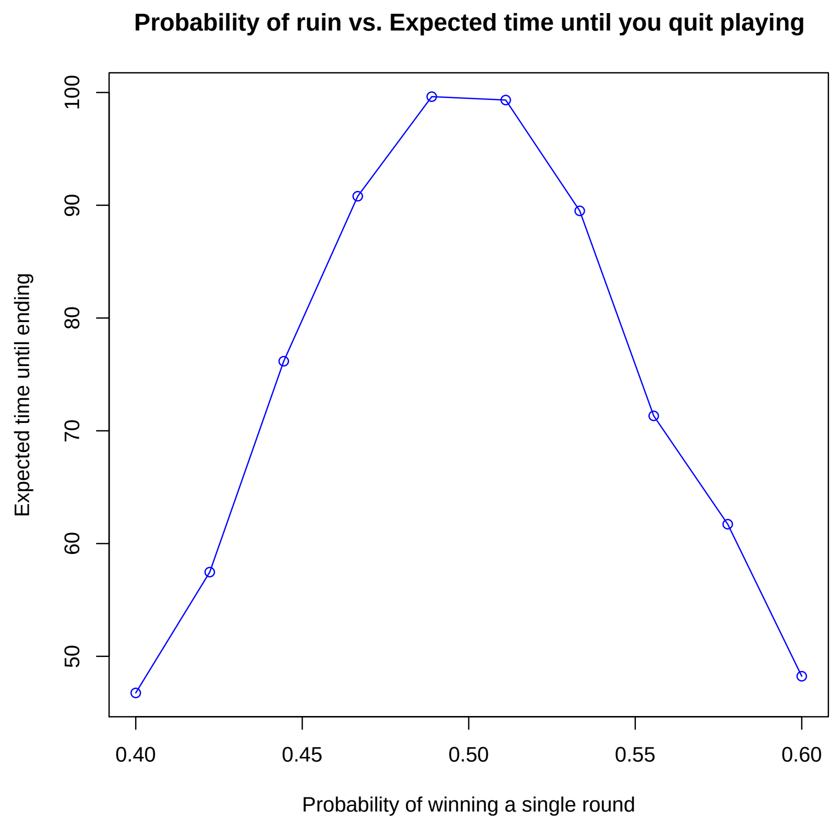


Figure 2: $P(\text{Ruin})$ and expected time to stop betting as a function of p

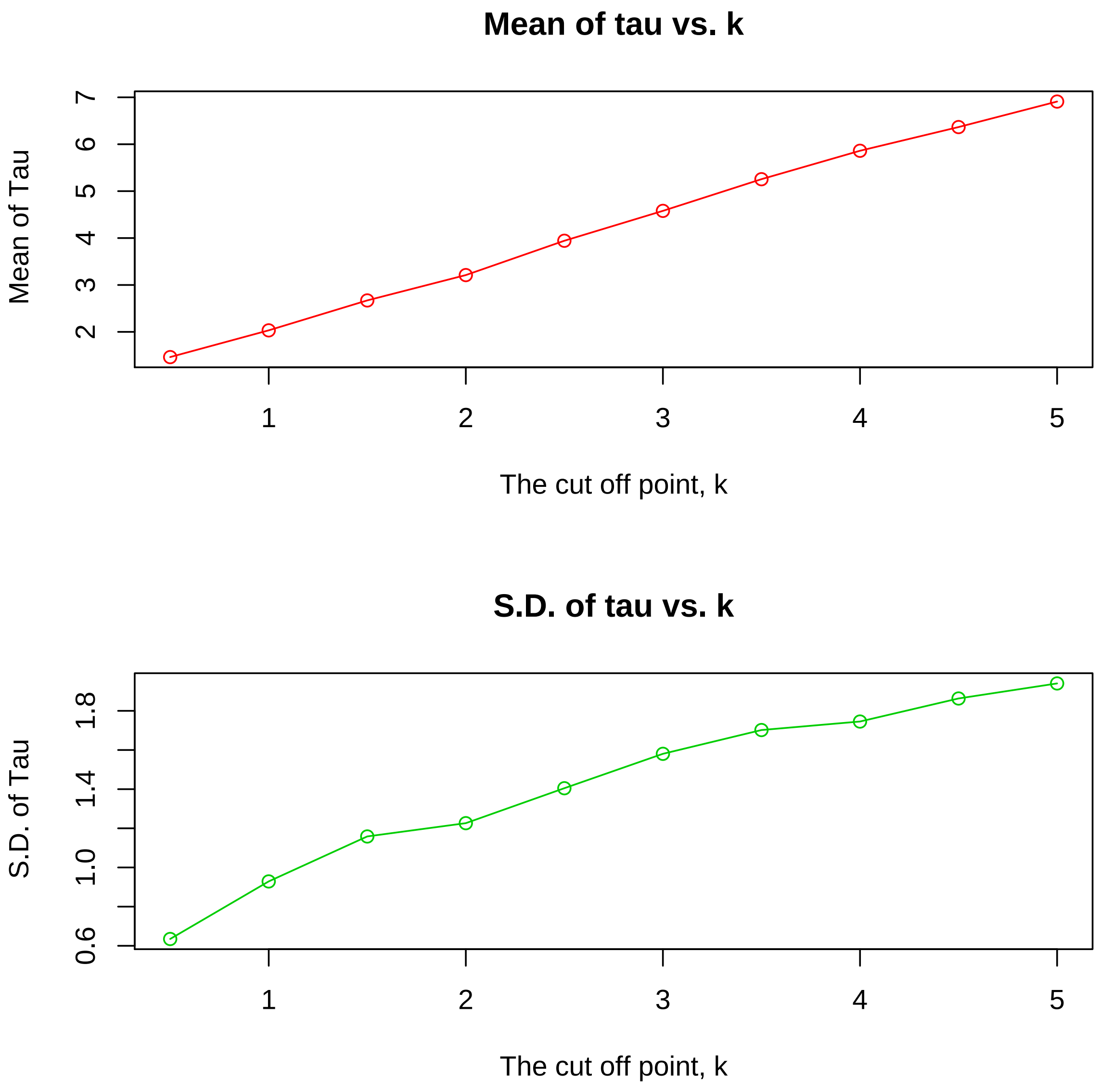


Figure 3: Estimated mean and standard deviation of τ_k vs. k.