Math 365 / Comp 365: Homework 1

Please submit a *stapled* hard copy of your work

Reminder: you are allowed—in fact encouraged—to work on and discuss homework together. You should, however, write up your own assignments in your own words. If you work with another person or if you get significant help from a classmate or external resource, you should give that person or resource credit in your solution (at no penalty to you)

INSERT YOUR NAME HERE

Lawson Busch

```
require(Matrix)
```

Problem 1

The point of this problem is twofold: (i) to illustrate what can happen if you accumulate many truncations of numbers and (ii) to give you practice writing programs.

In the 1999 movie Office Space, a character creates a program that takes fractions of cents that are truncated in a bank's transactions and deposits them into his own account. This is not a new idea, and hackers who have actually attempted it have been arrested. In this exercise, you will simulate the program to determine how long it would take to become a millionaire this way.

Assume the following details:

- You have access to 10,000 bank accounts
- Initially, the bank accounts have values that are uniformly distributed between $100 and $100,000
- The nominal annual interest rate on the accounts is 5%
- The daily interest rate is thus .05/365
- Interested is compounded each day and added to the accounts, except that fractions of a cent are truncated
- The truncated fractions are deposited into an illegal account that initially has a balance of $0
- The illegal account can hold fractional values and it also accrues daily interest

Your job is to write an R script that simulates this situation and finds how long it takes for the illegal account to reach a million dollars.

Here is some R help:

The following code generates the initial accounts:

```
options(digits=10)
accounts=runif(10000,100,100000)
accounts = floor(accounts*100)/100 #Removes the fractional cents by flooring after 2 decimal places (*100) then converting back to dollars
```

The first line expands the number of digits displayed. The second sets up 10,000 accounts with values uniformly between 100 and 100,000. The third line removes the fractions of cents (look at the data before and after that line is applied). To calculate interest for one day:

```
interest = accounts*(.05/365)
```

Depending on how you do it, you might want to use an if-then statement. For example, you might use something like

```
if (illegal > 1000000) break
```

The `break` command breaks out of a loop. Or, perhaps more elegantly, you might use a `while` loop

```
while (illegal < 1000000) { do stuff here }
```

You can find help on syntax in the Help Menu under “The R Language Definition.” That’s where I went to remind myself about the syntax for an if-then statement and a while loop.

Problem 1 Answer

```
countDays = function(accounts){
  illegal = 0
  days = 0
  #ones = rep(1, 10000)
  while(illegal < 1000000) {
    #print(accounts)
    days = days + 1
    #print(days)
    interest = accounts*(.05/365)
    accounts = accounts + interest
    flr = floor(accounts*100)/100
    #print(flr)
    #illegal = illegal + (ones)%*%t(accounts-flr) #This needs to be converted to an i
nteger
    illegal = illegal + sum(accounts-flr)
    illegal = illegal * (1+0.05/365)
    #print(illegal)
    accounts = flr
  }
  return(days)
}

options(digits=10)
testAccounts=runif(10000,100,100000)
countDays(testAccounts)
```

```
## [1] 9627
```

The illegal account would reach one million dollars in 9,627 days, or approximately 26.38 years.

Problem 2

The function

```
f(x)=(cos(x))^2+sin(x)-1
```

has roots at both π and $\pi/2$.

a. Using the convergence properties we discussed in class, estimate how many iterations of the bisection method are necessary to find the accurate solution to eight correct decimal places, when we start with a bracketing interval of [2.5,3.5].

If

```
f(x)=(cos(x))^2+sin(x)-1
```

and the interval is [2.5, 3.5] and we want an accurate solution to 8 correct decimal places, then:

```
(b-a)/(2^(i+1)) <= (1/2) * 10^-8

1/(2^(i+1)) <= (1/2) * 10^-8

2*10^8 <= 2^(i+1)

log10(10^8) <= log10(2^i)

8/(log10(2)) <= i

i >= 26.57 (approximately)
```

This implies that in order to have an accurate solution to 8 correct decimal places then we would need to have 27 iterations of the bisection method.

b. Use your `bisect` function to see how many iterations it actually takes.

```r
bisect = function(f,interval,tol=0.5*10^-10,max.its=40,verbose = FALSE){
  history = rep(NA, max.its)  #Creates vector of NAs to allocate memory,
  #Only returns the number of spaces filled when returned
  a = interval[1]
  b = interval[2]
  count = 0
  if(f(a)*f(b) > 0) {
    stop("f(a)f(b)<0 not satisfied") #Returns and prints out this error message
    #Similar to a try catch loop
  }
  while((b-a)/2 > tol){
    count = count + 1
    c = (a+b)/2
    history[count] = c
    if(verbose == TRUE) { #Essentially a debugging loop in R, can toggled with verbos
e input
      print(count)
      print(c(a,history[count],b,b-a))
    }
    if(f(c) == 0){
      root = c
      return (list(root=root,history=history[!is.na(history)], count = count))
    }
    if(f(a)*f(c) < 0) {
      b = c
    } else {
      a = c
    }
  }
  root = c
  return(list(root=root,history=history[!is.na(history)], count = count))
}

f = function(x) {
  return((cos(x))^2+sin(x)-1)
}

intervalB = c(2.5, 3.5)

bisect(f, intervalB)
```

```
## $root
## [1] 3.141592654
##
## $history
##  [1] 3.000000000 3.250000000 3.125000000 3.187500000 3.156250000
##  [6] 3.140625000 3.148437500 3.144531250 3.142578125 3.141601562
## [11] 3.141113281 3.141357422 3.141479492 3.141540527 3.141571045
## [16] 3.141586304 3.141593933 3.141590118 3.141592026 3.141592979
## [21] 3.141592503 3.141592741 3.141592622 3.141592681 3.141592652
## [26] 3.141592667 3.141592659 3.141592655 3.141592653 3.141592654
## [31] 3.141592654 3.141592654 3.141592654 3.141592654
##
## $count
## [1] 34
```

While the biesection method returns after 34 iterations, we can see from looking at the history that the method becomes accurate to 8 decimal points at 27 iterations, as from this point onwards, every result begins with the number 3.14159265, the exact same 8 decimal places, making the function accurate to 8 decimal places after this point.

c. By hand, compute a formula for Newton's iteration for this particular choice of $f(x)$; i.e., write a formula for x_{i+1} in terms of x_i.

Newton's iteration for f(x):

Note: xi1 and xi are the i+1 iteration and ith iteration of x, respectively.

```
xi1 = xi - f(xi)/f'(xi) for (cos(x))^2 + sin(x) - 1

xi1 = xi - ( (cos(x))^2 - sin(x) - 1) / ( -2*sin(x)*cos(x) + cos(x))
```

d. Set x_0 to 4.25, run 50 iterations of your Newton's formula from c) to compute $\{x_i\}_{i=0,1,\ldots,50}$, and compute $e_i = |x_i - \pi|$ for all i. When does the iteration converge? Now plot e_i/e_{i-1} over all i until convergence. What is the convergence order and rate to π?

```
newtons=function(x) {
  Newx = x - (((cos(x))^2+sin(x)-1)/(-2*sin(x)*cos(x)+cos(x)))
  return(Newx)
}

xVec = 1:50
eVec = 1:50

x = 4.25

for(i in 1:50) {
  xVec[i] = x
  ei = abs(x-pi)
  eVec[i] = ei #Still need to plot ei over convergence
  x= newtons(x)
}

#deriv1 = D(cos(x)^2 + sin(x) - 1~x)

deriv1 = function(x) {
  return(-2*sin(x)*cos(x)+cos(x))
}

deriv2 = function(x) {
  return(sin(x)*(2*sin(x)-1)-2 *cos(x)^2)
}

deriv1(pi) #Not zero implies quadratic convergence
```

```
## [1] -1
```

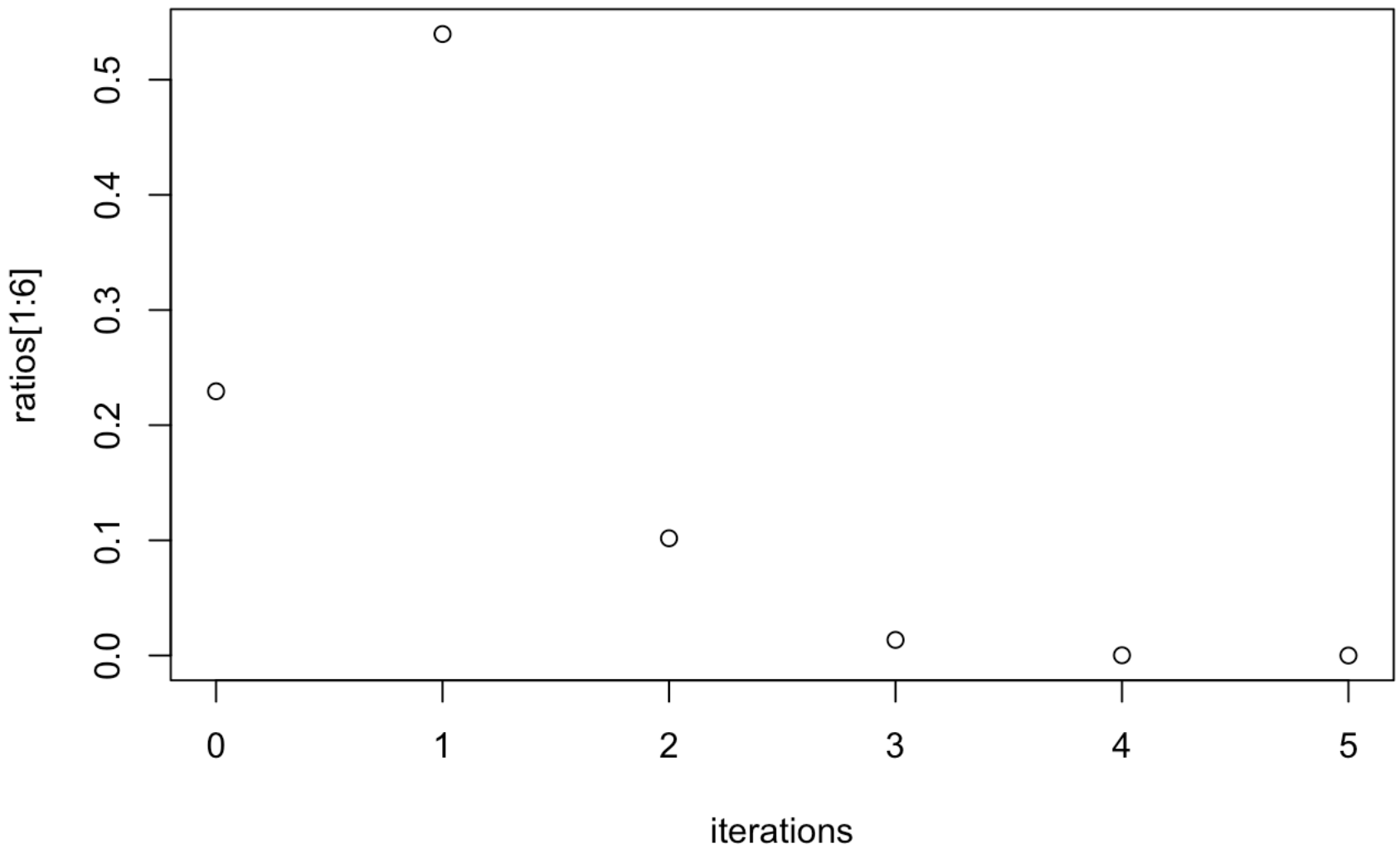
```
#Calculates what errors should converge to
deriv2(pi)/(2*deriv1(pi))
```

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

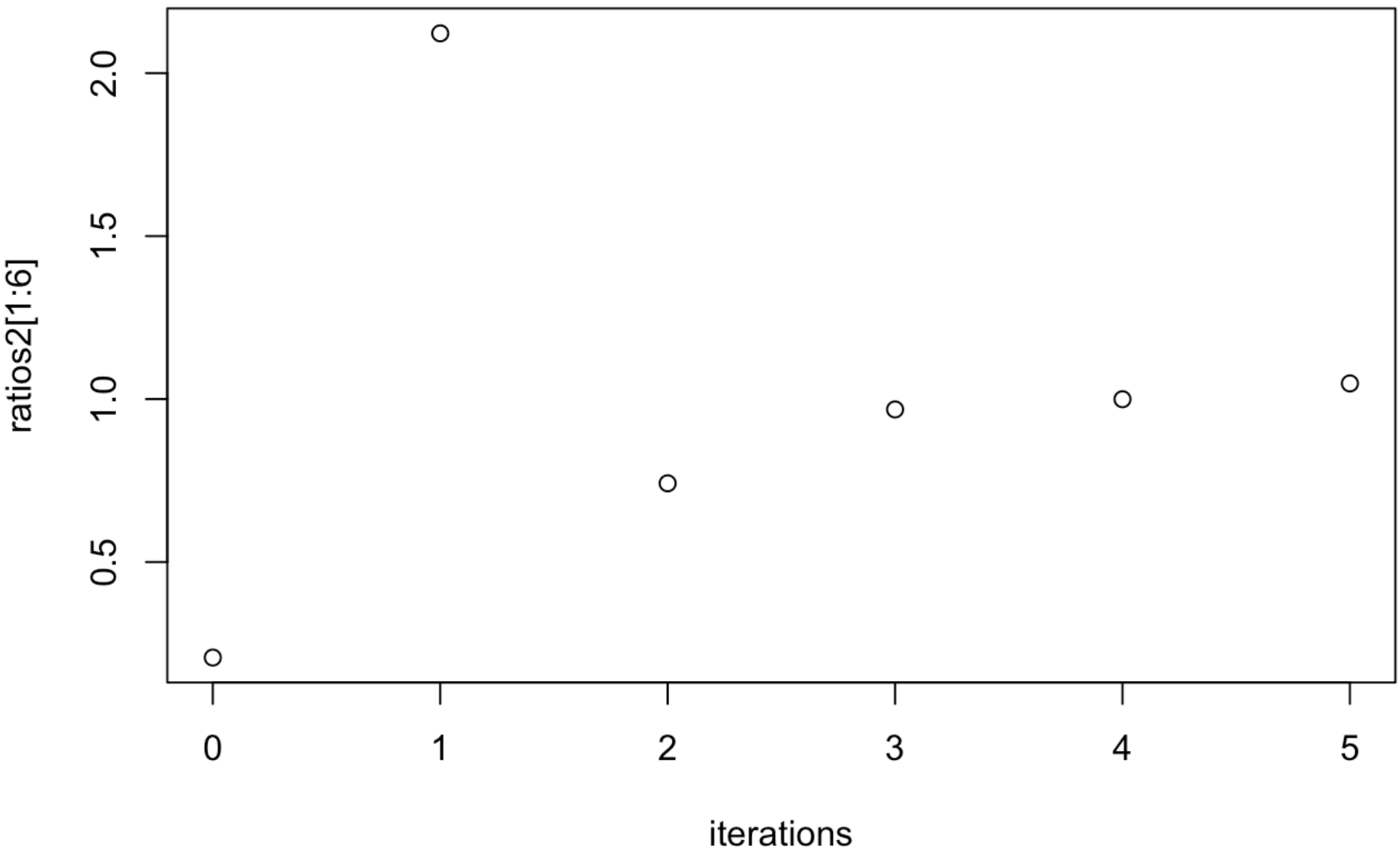
```
errors = eVec
n = length(errors)
(ratios=errors[2:n]/errors[1:(n-1)]^1)
```

```
##  [1] 2.294314454e-01 5.396592772e-01 1.017700902e-01 1.352254004e-02
##  [5] 1.887813381e-04 3.736646347e-08 0.000000000e+00             Inf
##  [9] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [13] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [17] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [21] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [25] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [29] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [33] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [37] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [41] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [45] 0.000000000e+00             Inf 0.000000000e+00             Inf
## [49] 0.000000000e+00
```

```
iterations = seq(0, 5)
plot(iterations, ratios[1:6], type = 'p')
```




```
ratios2=errors[2:n]/errors[1:(n-1)]^2
plot(iterations, ratios2[1:6], type = 'p')
```



Here we can see from the error equation for newton's method that the error ratios should converge to 1 for pi. We also know that since f(r) = 0 and f'(r) != 0, that Newton's method should be locally and quadratically convergent for f(pi). When checking our error ratios for linear convergence, we see that the error ratios do not converge to one, and thus that our function is not linearly convergent for pi, which is what we expect for Newton's method for this case. Further, if we check our error ratios for quadratic convergence, we see that they do in fact converge to 1 quadratically, demonstrating that newtons method is quadratically convergent with a convergence rate of 1 for f(pi).

e. Repeat d) with x_0 =1.5. What is the convergence order and rate to $\pi/2$? If your answer is different from d), explain why.

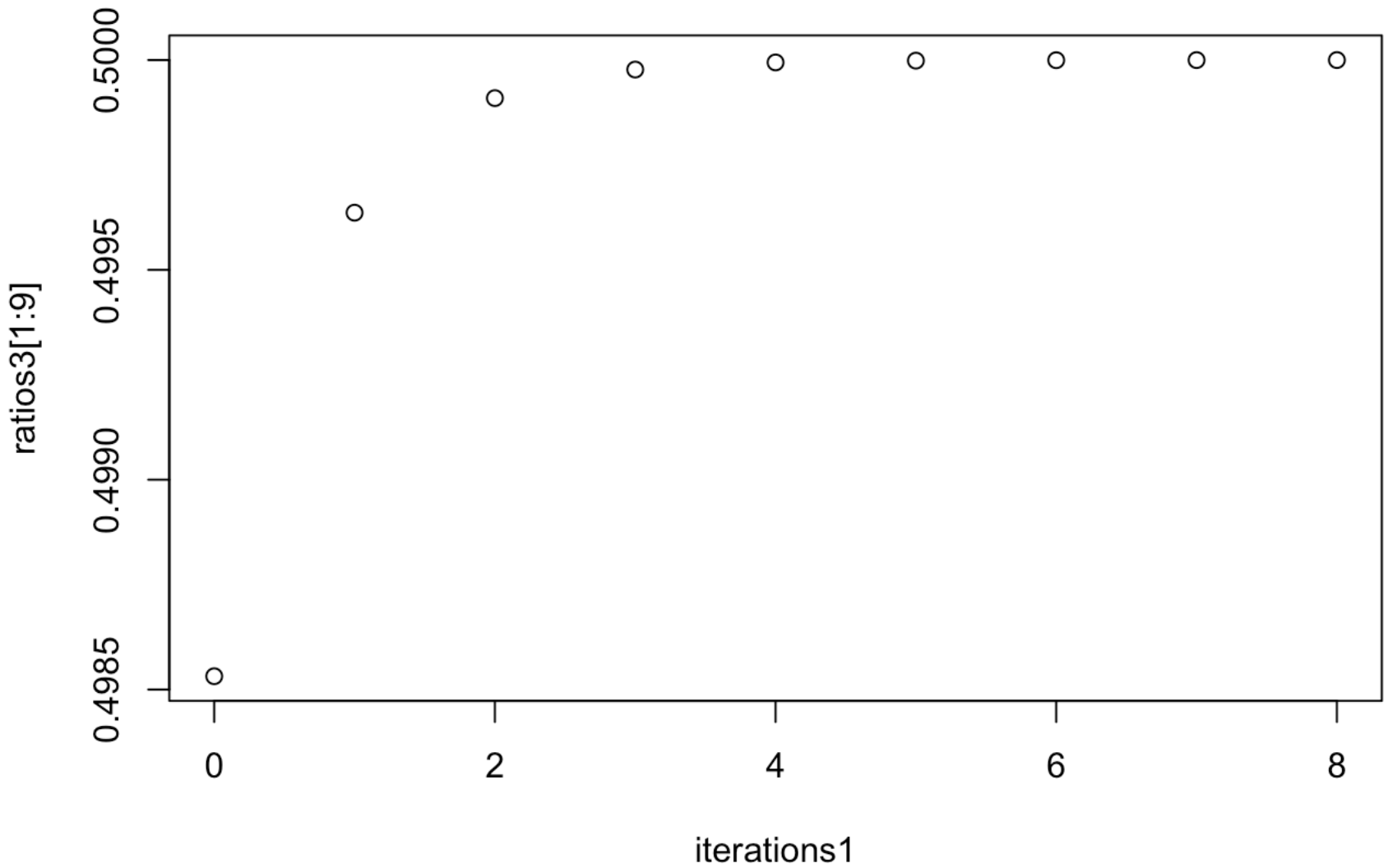
```
xVec = 1:50
eVec = 1:50

x = 1.5

for(i in 1:50) {
  xVec[i] = x
  ei = abs(x-pi/2)
  eVec[i] = ei #Still need to plot ei over convergence
  x = newtons(x)
}

#print(xVec)
#print(eVec)

deriv1(pi/2) #Zero implies not quadratically convergent
```

```
## [1] -6.123233996e-17
```

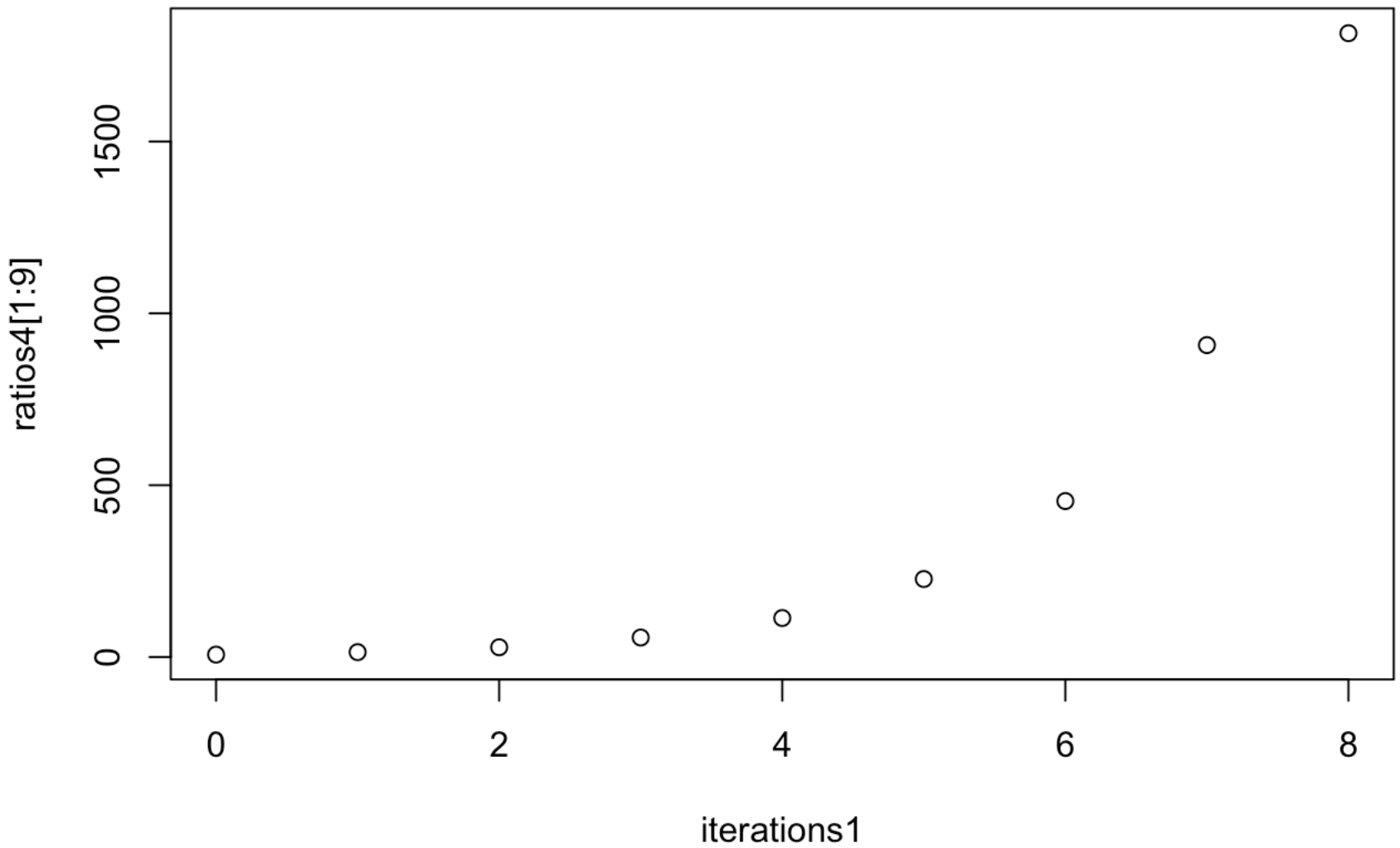
```
#Linear
errors = eVec
n = length(errors)
(ratios3=errors[2:n]/errors[1:(n-1)]^1)
```

```
##  [1] 0.4985317175 0.4996362814 0.4999092766 0.4999773320 0.4999943338
##  [6] 0.4999985835 0.4999996459 0.4999999113 0.4999999787 0.4999999928
## [11] 0.4999999984 0.5000000440 0.4999997579 0.4999985248 0.4999985709
## [16] 0.5000076981 0.5000710348 0.5000213447 0.5000640296 0.4971257411
## [21] 0.5037014058 0.4131623443 1.0000000000 1.0000000000 1.0000000000
## [26] 1.0000000000 1.0000000000 1.0000000000 1.0000000000 1.0000000000
## [31] 1.0000000000 1.0000000000 1.0000000000 1.0000000000 1.0000000000
## [36] 1.0000000000 1.0000000000 1.0000000000 1.0000000000 1.0000000000
## [41] 1.0000000000 1.0000000000 1.0000000000 1.0000000000 1.0000000000
## [46] 1.0000000000 1.0000000000 1.0000000000 1.0000000000
```

```
iterations1 = seq(0, 8)
plot(iterations1, ratios3[1:9], type = 'p')
```




```
#Quadratic
ratios4=errors[2:n]/errors[1:(n-1)]^2
plot(iterations1, ratios4[1:9], type = 'p')
```



Since f’(r) = 0 for f(pi/2), we know that newton’s formula does not converge locally and quadratically for f(pi/2). For an initial guess of x = 1.5, we can see that this function is linearly convergent to 1/2, as the errors eventually converge to 1/2 for an exponent of 1 in the error equation. We can also see that it is not quadratically convergent, as it keeps growing continuosly with an exponent value of 2 in the error equation. This shows that the function is linearly convergent with a convergence rate of 1/2 for a starting x value of 1.5. This matches the expected results of newton’s method for f(pi/2) since we expected it to not converge quadratically.

Problem 3

Computer Problem 7 from Section 1.1 of the book (involving a determinant).

```
problem3 = function(x) {
  A = rbind(c(1, 2, 3, x), c(4, 5, x, 6), c(7, x, 8, 9), c(x, 10, 11, 12))
  a = det(A) - 1000
  return(a)
}

interval = c(0, 20)

# 9.708299123

bisect(problem3, interval)
```

```
## $root
## [1] 9.708299123
##
## $history
##  [1] 10.000000000  5.000000000  7.500000000  8.750000000  9.375000000
##  [6]  9.687500000  9.843750000  9.765625000  9.726562500  9.707031250
## [11]  9.716796875  9.711914062  9.709472656  9.708251953  9.708862305
## [16]  9.708557129  9.708404541  9.708328247  9.708290100  9.708309174
## [21]  9.708299637  9.708294868  9.708297253  9.708298445  9.708299041
## [26]  9.708299339  9.708299190  9.708299115  9.708299153  9.708299134
## [31]  9.708299125  9.708299120  9.708299122  9.708299123  9.708299123
## [36]  9.708299123  9.708299123  9.708299123
##
## $count
## [1] 38
```

```
interval2 = c(-20, 0)

# -17.18849815

bisect(problem3, interval2)
```

```
## $root
## [1] -17.18849815
##
## $history
##  [1] -10.00000000 -15.00000000 -17.50000000 -16.25000000 -16.87500000
##  [6] -17.18750000 -17.34375000 -17.26562500 -17.22656250 -17.20703125
## [11] -17.19726562 -17.19238281 -17.18994141 -17.18872070 -17.18811035
## [16] -17.18841553 -17.18856812 -17.18849182 -17.18852997 -17.18851089
## [21] -17.18850136 -17.18849659 -17.18849897 -17.18849778 -17.18849838
## [26] -17.18849808 -17.18849823 -17.18849815 -17.18849812 -17.18849814
## [31] -17.18849814 -17.18849815 -17.18849815 -17.18849815 -17.18849815
## [36] -17.18849815 -17.18849815 -17.18849815
##
## $count
## [1] 38
```

```
problem3(9.708299123)
```

```
## [1] -2.912872787e-07
```

```
problem3(-17.18849815)
```

```
## [1] -1.302615988e-05
```

```
A1 = rbind(c(1, 2, 3, 9.708299123), c(4, 5, 9.708299123, 6), c(7, 9.708299123, 8, 9),
c(9.708299123, 10, 11, 12))
det(A1)
```

```
## [1] 999.9999997
```

```
A2 = rbind(c(1, 2, 3, -17.18849815), c(4, 5, -17.18849815, 6), c(7, -17.18849815, 8,
9), c(-17.18849815, 10, 11, 12))
det(A2)
```

```
## [1] 999.999987
```

By using the bisection method on the interval [0, 20] we can find the first root of 9.708299123, and by using the bisection method on the interval [-20, 0] we can find the second root of -17.18849815. When plugged back in to our problem 3 equation, we see that we get an incredibly small number that is essentially 0 for both equations, demonstrating that both roots make the determinate of the matrix = 1000 when substituted for x. This is further evidenced by reconstructing the matrix with these numbers substututed for x, and taking the determinant of the matrix. Both determinants result in an answer that is essentially equal to 1000.

Problem 4

Computer Problem 8 from Section 1.1 of the book (involving the Hilbert matrix)

```
library(Matrix)
H=Hilbert(5)

problem4 = function(x) {
  B = Hilbert(5)
  B[1, 1] = x
  b = det(B - pi*diag(5)) #diag(5) creates the identity matrix
  return(b)
}

inter = c(0, 20)

bisect(problem4, inter)
```

```
## $root
## [1] 2.948010765
##
## $history
##  [1] 10.000000000  5.000000000  2.500000000  3.750000000  3.125000000
##  [6]  2.812500000  2.968750000  2.890625000  2.929687500  2.949218750
## [11]  2.939453125  2.944335938  2.946777344  2.947998047  2.948608398
## [16]  2.948303223  2.948150635  2.948074341  2.948036194  2.948017120
## [21]  2.948007584  2.948012352  2.948009968  2.948011160  2.948010564
## [26]  2.948010862  2.948010713  2.948010787  2.948010750  2.948010769
## [31]  2.948010759  2.948010764  2.948010766  2.948010765  2.948010765
## [36]  2.948010765  2.948010765  2.948010765
##
## $count
## [1] 38
```

```
# 2.948010765

problem4(2.948010765)
```

```
## [1] -6.316480907e-09
```

```
test = Hilbert(5)
test[1,1] = 2.948010765

max(eigen(test)$values)
```

```
## [1] 3.141592654
```

By creating replacing the [1,1] column of the Hilbert matrix with a variable, x, we can use the bisection method to solve for the value of x at B[1, 1] in our function. Since we know that we want pi as our maximum eigenvalue, we simply substitute pi for lamda in the equation det(B) - lamda*I = 0 to solve for eigenvalues. After using the bisection method we get the root 2.948010765, which when plugged in for x in our problem 4 equation, gives us an incredibly small value essentially equal to 0, demonstrating that it is indeed the value for B[1, 1] that would make pi an eigenvalue of the hilbert matrix. It also makes pi the maximum eigenvalue for the matrix, as can be seen by max(eigen(test)$values) resulting in pi.

Some hints for Problems 3 and 4:

1. To bind rows together into a matrix:

```
rbind(c(1,2,3,44),c(4,5,44,6),c(7,44,8,9),c(44,10,11,12))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3   44
## [2,]    4    5   44    6
## [3,]    7   44    8    9
## [4,]   44   10   11   12
```

2. If A is a matrix, then `det(A)` is its determinant
3. For Problem 3, make a function of `x` that puts `x` on the backward diagonal, computes its determinant, and subtracts 1000. Then use your `bisect` on this function.
4. For Problem 4, to get the Hilbert matrix, `H`, you can use the `Hilbert` function from the `Matrix` package. Here is an example of how to generate a Hilbert matrix and change the first entry:

```
H=Hilbert(5)
H[1,1]=132
```

5. To get the maximum eigenvalue of a matrix `A`, use

```
max(eigen(A)$values)
```

6. In R, your favorite irrational and transcendatal number is written as `pi`:

```
pi
```

```
## [1] 3.141592654
```

R Markdown Tips

You can assemble your homework writeup however you like, but I strongly encourage you to give R Markdown a shot. I was skeptical of learning yet another new tool, but it only takes 15 minutes to get a pretty good handle, and so far I really like it.

If the R code you place inside the hash marks has printed output, it will display like this:

```
(uniformSamples<-runif(10,0,1))
```

```
##  [1] 0.51541832951 0.35980169522 0.38504839782 0.74397035129 0.26156682079
##  [6] 0.94585744431 0.44961418165 0.92148022004 0.79509323626 0.02514774934
```

```
mean(uniformSamples)
```

```
## [1] 0.5402998426
```

You can also include comments and embed plots:

```
# Define a polynomial function
f = function(x) {x^3 + x^2 - 24*x + 36}
# Plot the function
x = seq(-8,4,len=10000)
plot(x,f(x),type="l",lwd=3,main="f(x) = x^3 + x^2-24 * x +36")
abline(0,0,col="red")
```