Math 365 / Comp 365: Homework 2

Please bring a stapled hard copy of your answers to class, citing any collaborators.

Lawson Busch

Collaborators: Raven McKnight, Amanda Doan, Ximena Silva-Avila

Problem 1

We'll be using polynomial approximation techniques regularly in the class, and it may have been a while since you learned Taylor's Theorem, so here is a chance to review it quickly in Section 0.5:

Problem 8 in Section 0.5 in the book (Taylor polynomial approximation of cosine).

Additional part c) Using R, make a single plot with 3 different curves: cos(x) and the 2nd and 4th degree Tayolor approximations of cosine around x=0. Plot these on the interval $[-\pi/2, \pi/2]$.

Part A:

The fifth degree taylor polynomial for the function $f(x) = cos(x)$ centered at $x = 0$. Note the x^5 term is missing because it cancels out due to sins evaluation of 0.

$$P_5(x) = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4$$

The upper bound for the error can be calculated by using the 6th term of the taylor polynomial, for which the sixth derivative of $f(x) = cos(x)$ is $-cos(x)$, for which the maximum possible value in the interval $[-\pi/2, \pi/2]$ is 1. So if we plug this maximum value into our 6th degree term of the taylor polynomial we can caluclate the upper bound on our error to be:

$$R_n(x) = \frac{1}{6!}(\frac{\pi}{4} - 0)^6$$

$$R_n(x) = \frac{1}{720}(\frac{\pi}{4})^6$$

Which evaluates to 0.0003259919, which implies that $error \leq 0.0003259919$.

The second and fourth degree taylor polynomials are:

$$P_2(x) = 1 - \frac{1}{2!}x^2$$

$$P_4(x) = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4$$

And plotted look like:

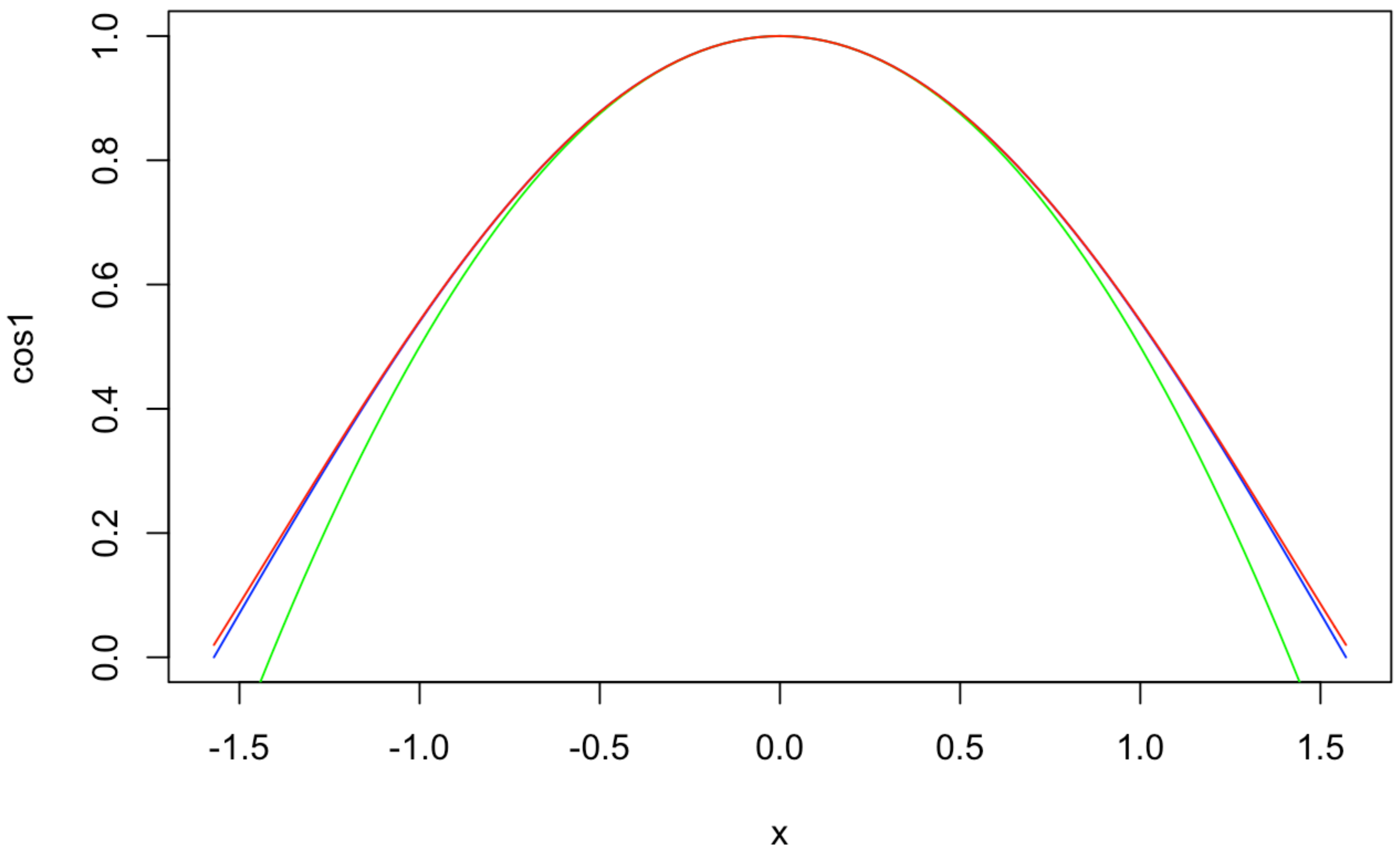
```r
library(ggplot2)

cos1 = function(x){
  cos(x)
}

taylor2 = function(x){
  1 - .5*x^2
}

taylor4 = function(x){
  taylor2(x) + (1/24)*x^4
}

plot(cos1, -pi/2, pi/2 , type = "l", col= "blue")
plot(taylor2, -pi/2, pi/2, col = "green", add=TRUE)
plot(taylor4, -pi/2, pi/2, col = "red", add=TRUE)
```



From these plots we can see that the taylor polynomial gives a pretty good approximation of the $f(x) = cos(x)$, especially for the higher degree taylor polynomials.

Problem 2

Problem 4 in the Section 0.2 Exercises, parts (a), (b), and (f) only (on converting to binary). Do all computations by hand.

Part A:

11.25

$$\begin{gathered} 11 = 2^3 + 2^1 + 2^0 = 1011 \\ 0.25 * 2 = 0.5 + 0 \\ 0.5 * 2 = 0 + 1 \\ \rightarrow 11.25_{10} = 1011.01_2 \end{gathered}$$

Part B:

2/3

$$\begin{gathered} \frac{2}{3} * 2 = \frac{1}{3} + 1 \\ \frac{1}{3} * 2 = \frac{2}{3} + 0 \\ \frac{2}{3} * 2 = \frac{1}{3} + 1 \\ \frac{1}{3} * 2 = \frac{2}{3} + 0 \\ \rightarrow \frac{2}{3}_{10} = 0.\overline{10}_2 \end{gathered}$$

Part C:

99.9

$$\begin{gathered} \frac{99}{2} = 49R1 \\ \frac{49}{2} = 24R1 \\ \frac{24}{2} = 12R0 \\ \frac{12}{2} = 6R0 \\ \frac{6}{2} = 3R0 \\ \frac{3}{2} = 1R1 \\ \frac{1}{2} = 0R1 \\ \rightarrow 99_{10} = 1100011_2 \end{gathered}$$

$$0.9 * 2 = 0.8 + 1$$
$$0.8 * 2 = 0.6 + 1$$
$$0.6 * 2 = 0.2 + 1$$
$$0.2 * 2 = 0.4 + 0$$
$$0.4 * 2 = 0.8 + 0$$
$$0.8 * 2 = 0.6 + 1$$
$$0.6 * 2 = 0.2 + 1$$
$$\rightarrow 0.9_{10} = .1\overline{1100}_2$$
$$\rightarrow 99.9_{10} = 1100011.1\overline{1100}_2$$

Problem 3

Section 0.3: Exercise 9. In this problem, you should do the following: write 7/3, 4/3, and 1/3 in binary and in their IEEE machine expressions. Then perform the required subtractions using machine arithmetic. Finally, use the command `options(digits=20)` before doing the computations in R. Recall that the command `.Machine$double.eps` will give you machine epsilon in R.

In binary:

$$\frac{7}{3}_{10} = 10.\overline{01}_2$$
$$\frac{4}{3}_{10} = 1.\overline{01}_2$$
$$\frac{1}{3}_{10} =. \overline{01}_2$$

In IEE Representation:

$$\frac{7}{3}_{10} = 1.0\overline{01} * 2^1$$
$$\frac{4}{3}_{10} = 1.\overline{01} * 2^0$$
$$\frac{1}{3}_{10} = 1.\overline{01} * 2^{-2}$$

In the floating point representation:

$$\frac{7}{3}_{10} = 0|10000000000|0010101...10|101010$$
$$\rightarrow \textit{mantissa rounds up to } 001010...11$$
$$\rightarrow \frac{7}{3}_{10} = 0|10000000000|0010101...11$$
$$\textit{and } \frac{4}{3}_{10} = 0|01111111111|0101...01|01$$
$$\rightarrow \textit{mantissa rounds down}$$

This implies that there will be an extra bit at the end of the subtraction of $(\frac{7}{3} - \frac{4}{3})$ since one of the numbers was rounded up and the other was rounded down, implying that after the subtraction $0|01111111111|000000...01$ remains, which is just larger than one. So when 1 is subtracted from this number just larger than one, machine epsilon is left (since only the 2^{-52} bit contains a one).

Note that $(\frac{4}{3} - \frac{1}{3}) - 1$ will not result in machine epsilon, because like $\frac{4}{3}$, $\frac{1}{3}$ rounds down in IEE double representation, which is represented as $0|01111111101|0101...01$. This implies that $\frac{4}{3} - \frac{1}{3}$ will equal exactly one in IEE representation. Which implies that $(\frac{4}{3} - \frac{1}{3}) - 1$ will evaluate to exactly zero.

These results can be verified using R.

Here we can see the value of machine epsilon in R for IEE double precision:

```
options(digits=20)
print(.Machine$double.eps)
```

```
## [1] 2.220446049250313080808e-16
```

```
options(digits=20)
(7/3 - 4/3)-1
```

```
## [1] 2.220446049250313080808e-16
```

```
(4/3-1/3)-1
```

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

The above calculations verify that $(\frac{7}{3} - \frac{4}{3}) - 1$ can be used to calculate machine epsilon, while $(\frac{4}{3} - \frac{1}{3}) - 1$ cannot be used to calculate machine epsilon, as demonstrated above using the floating point representations of these numbers.

Problem 4

Using R, compute: 1000000000000000000 + 100 - 1000000000000000000. What happens? Explain why this is.

```
1000000000000000000 + 100 - 1000000000000000000
```

```
## [1] 128
```

Instead of resulting in 100, as it should, this calculation results in 128. This is because of rounding error involved in representing floating point numbers, and in this case is caused by the number 1000000000000000000_{10}, which in binary translates to $110111100000101101101011001110100111011001000000000000000000_2$. This number in binary takes 59 digits to represent, and thus cannot be represented accurately using IEE floating point representation as it is too large to represent exactly with 52 decimal places. Thus when 100_{10} is added to it, some rounding error occurs since the new integer cannot be represented exactly in IEE precision. As a result, when 1000000000000000000_{10} is subtracted from the result of $1000000000000000000_{10} - 100_{10}$, the result is not exactly 100_{10}, due to the inaccuracy of stored numbers required more than 52 digits in binary using IEE double precision.

Problem 5

Find the smallest positive integer i such that i is not exactly representable using the IEEE standard in double precision; i.e., $\text{fl}(i) \neq i$.

The smallest possible integer i that cannot be represented exactly in double precision format is $2^{53} + 1$. This is because the largest possible floating point number that can be represented in double precision format is 2^{53} (since we drop the leading 1), which would fill up the mantissa with 1 values. Thus by adding 1 to 2^{53}, we would add a 53rd decimal place to the mantissa (there would be 52 0s between the leading 1 and the end 1), causing rounding, which resulting in an unexact representation for the integer $2^{53} + 1$ in double precision representation.

Problem 6

Create an R program for “naive” Gaussian elimination (meaning no row exchanges allowed). Use it to solve the system in exercise 2a. You may put together the code fragments in Section 2.1 to construct your function. However, you may construct a more “vectorized” code (using at least one fewer for loop) for at least one bonus point.

```r
naiveGaussian = function(mt) {
  n = nrow(mt)
  x = rep(1,n) #Instatiate our result vector
  col = ncol(mt)
  print(nrow(mt))
  #Elimination Section
  for(j in 1:(n-1)) { #Iterates over the columns
    if(abs(mt[j,j]) < .Machine$double.eps) { #Checks if a 0 pivot
      break
    }
    for(i in (j+1):n) { #Iterate over rows
      mult = mt[i,j] / mt[j,j] #Calculate the reduction factor
      for(k in (j):n) {
        mt[i,k] = mt[i,k] - mult*mt[j,k] #Update row values
      }
      mt[i, col] = mt[i, col] - mult*mt[j, col] #Adjust our resulting b values
    }
  }
  print(mt) #Used to check accuracy of elimination step
  #Backwards substitution step: this works
  x[n] = mt[n, col] / mt[n, n]#calculate x[n] value
  for(i in (n-1):1) { #iterate over rows
    for(j in (i+1):n) { #Iterate over columns
      mt[i, col] = mt[i, col] - mt[i,j]*x[j] #Calculate b value
    }
    x[i] = mt[i, col]/ mt[i,i] #Store final x value
  }
  return(x)
}

m1 = cbind(c(2, 4, -2), c(-2, 1, 1), c(-1, -2, -1), c(-2, 1, -3))
naiveGaussian(m1)
```

```
## [1] 3
##      [,1] [,2] [,3] [,4]
## [1,]    2   -2   -1   -2
## [2,]    0    5    0    5
## [3,]    0    0   -2   -4
```

```
## [1] 1 1 2
```

As we can see, this function is correct as it outputs the correct matrix reduction and x values for the input matrix.