Math 4610 Final

Cody Frisby

12/8/2017

1. Root-finding Methods

Let

$$f(x) = x^2 - 2$$

A. Given $x_0 = 2$, find x_1, x_2, x_3 if the sequence x_n is defined by Newton's method to approximate a solution of $f(x) = 0$.

$$x_1 = 2 - \frac{2}{4} = 1.5$$
$$x_2 = 1.5 - \frac{0.25}{3} = 1.4166667$$
$$x_3 = 1.4166667 - \frac{0.0069444}{2.8333333} = 1.4142157$$

B. Given $x_0 = 2$, find x_1, x_2, x_3 if the sequence x_n is defined by Modified Newton's method (repeating the initial evaluation of the inverse of the derivative) to approximate a soution of $f(x) = 0$.

Modified Newton's method can be defined as

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})(1 - \frac{f(x_{n-1})f''(x_{n-1})}{(f'(x_{n-1}))^2})}$$

which gets kinda messy and we need to compute one more derivative of f. Here are the first few terms of the sequence.

$$x_1 = 2 - \frac{2}{3} = 1.3333333$$
$$x_2 = 1.3333333 - \frac{-0.2222222}{2.8333333} = 1.4117647$$
$$x_3 = 1.4117647 - \frac{-0.0069204}{2.8284314} = 1.4142114$$

C. Given $x_0 = 2$, find x_1, x_2, x_3 if the sequence x_n is defined by Steffensen's method to approximate a soution of $f(x) = 0$.

$$x_1 = 2 - \frac{2}{6} = 1.6666667$$
$$x_2 = 1.6666667 - \frac{0.7777778}{4.1111111} = 1.4774775$$
$$x_3 = 1.4774775 - \frac{0.1829397}{3.1378947} = 1.4191773$$

D. Given $x_0 = 2$ and $x_1 = 1$, find x_2, x_3 if the sequence x_n is defined by the bisection method to approximate a soution of $f(x) = 0$.

$$x_2 = \frac{1+2}{2} = 1.5$$
$$x_3 = \frac{1+1.5}{2} = 1.25$$
$$x_4 = \frac{1.25+1.5}{2} = 1.375$$

E. Given $x_0 = 2$ and $x_1 = 1$, find x_2, x_3 if the sequence x_n is defined by the secant method to approximate a soution of $f(x) = 0$.

$$x_2 = 2 - (2)\frac{2-1}{2+1} = 1.3333333$$
$$x_3 = 1.3333333 - (-0.2222222)\frac{1.3333333 - 2}{-0.2222222 - 2} = 1.4$$

F. Given $x_0 = 2$ and $x_1 = 1$, find x_2, x_3 if the sequence x_n is defined by the false position method to approximate a soution of $f(x) = 0$.

$$x_2 = 2 - (2)\frac{2-1}{2+1} = 1.3333333$$
$$x_3 = 1.3333333 - (-0.2222222)\frac{1.3333333 - 2}{-0.2222222 - 2} = 1.4$$

Note: This is the same result as secant method, up to this point. They actually diverge on the next iteration, but the first two are the same. The difference with the false position method is that it keeps the root bracketed throughout the algorithm.

2. What is the rate or order of convergence of Newton's method to a solution of $sin(x) = -1$?

Newton's method for this function is

$$x_{n+1} = x_n - \frac{sin(x_n) + 1}{cos(x_n)}$$

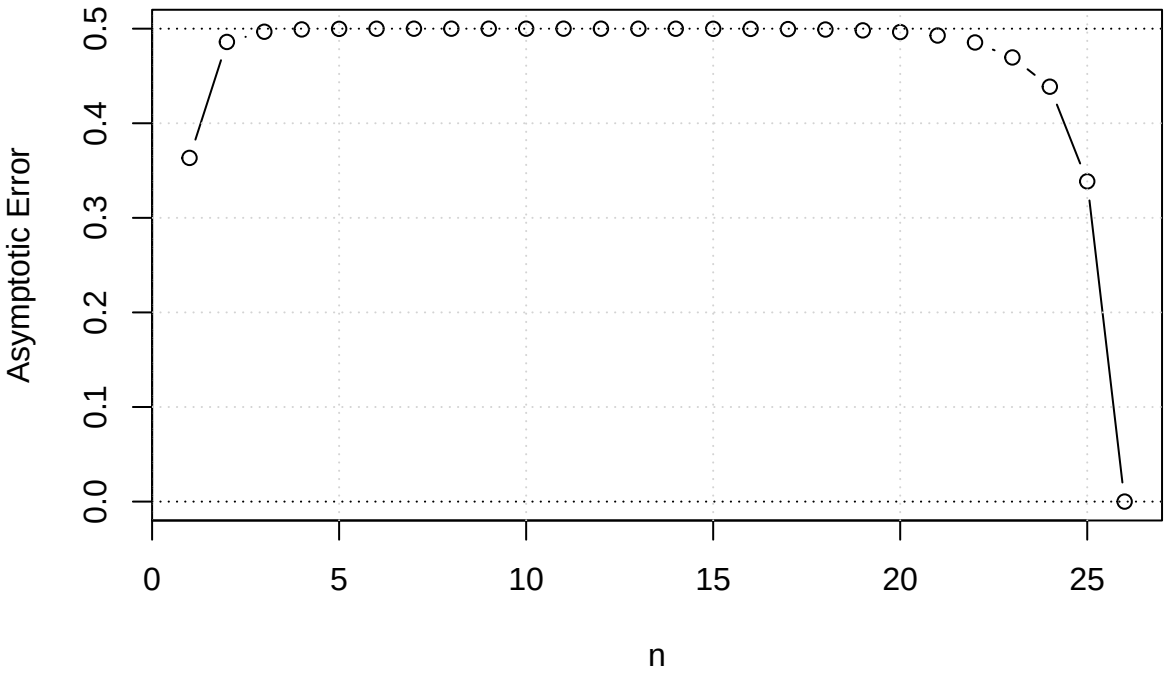
. Even though we are using Newton's method (which in general is quadratically convergent) this is essentially a fixed point problem. So, we should have fixed point convergence, which is linear, at rate = 1. For $sin(x) + 1 = 0$ (which has periodic roots), and $x_0 = 0$ it takes approximately 27 iterations to get with in 1.4901161×10^{-8} of one of the periodic roots.

Using the definition for *order of convergence* we get

$$\lim_{n\to\infty} \frac{\left|x_n - \frac{sin(x_n)+1}{cos(x_n)} - r\right|}{\left|x_{n-1} - \frac{sin(x_{n-1})+1}{cos(x_{n-1})} - r\right|^{\alpha}} = \lambda$$

which can be shown that $\lambda \approx 0.5$ as $n \to \infty$ and $\alpha = 1$. So, Newton's method is only linearly convergent for $sin(x) + 1 = 0$.

I made the claim that the asymptotic error constant (λ) is 0.5 but we can visualize (I like plots) the asymptotic error constant appears to be 0.5 if we plot this value as a function of n.



We can see that other than the first and last few iterations, the asymptotic error is close to 0.5. Although this is not a proof, it is helpful to see what is happening to the approximate error with each iteration.

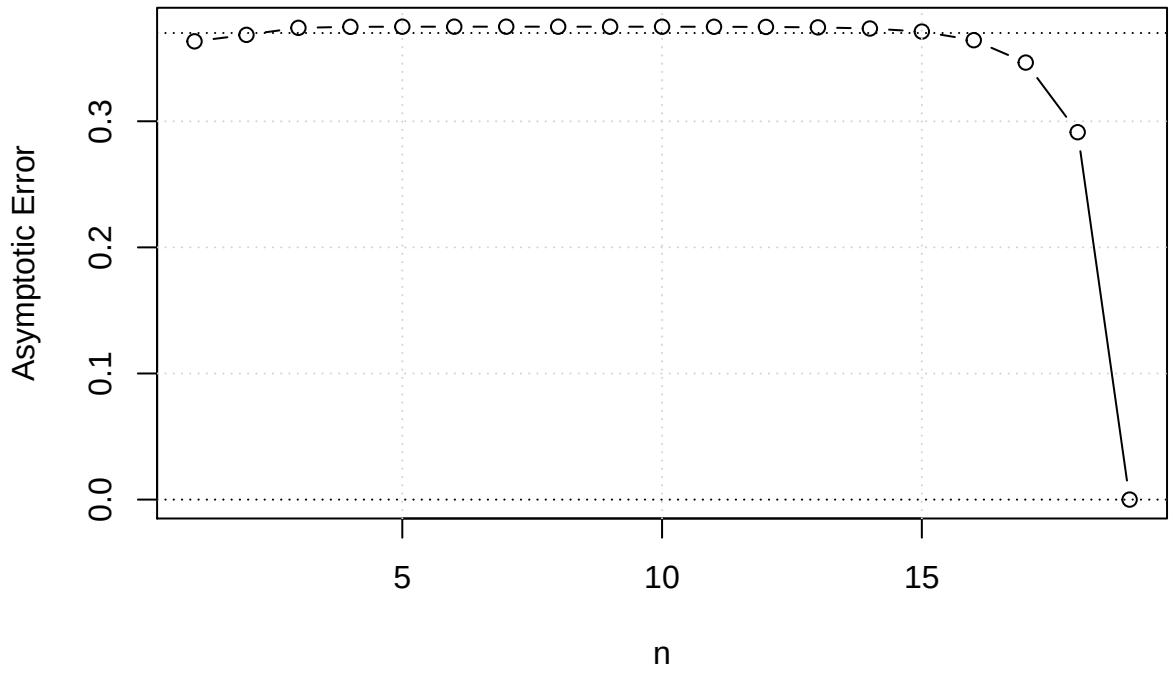
How could you modify the method to improve the performance?

In order to improve the performance of Newton's method with this function, we must get α (in the above equation) to increase somehow. We could tack another term onto Newton's method as shown below

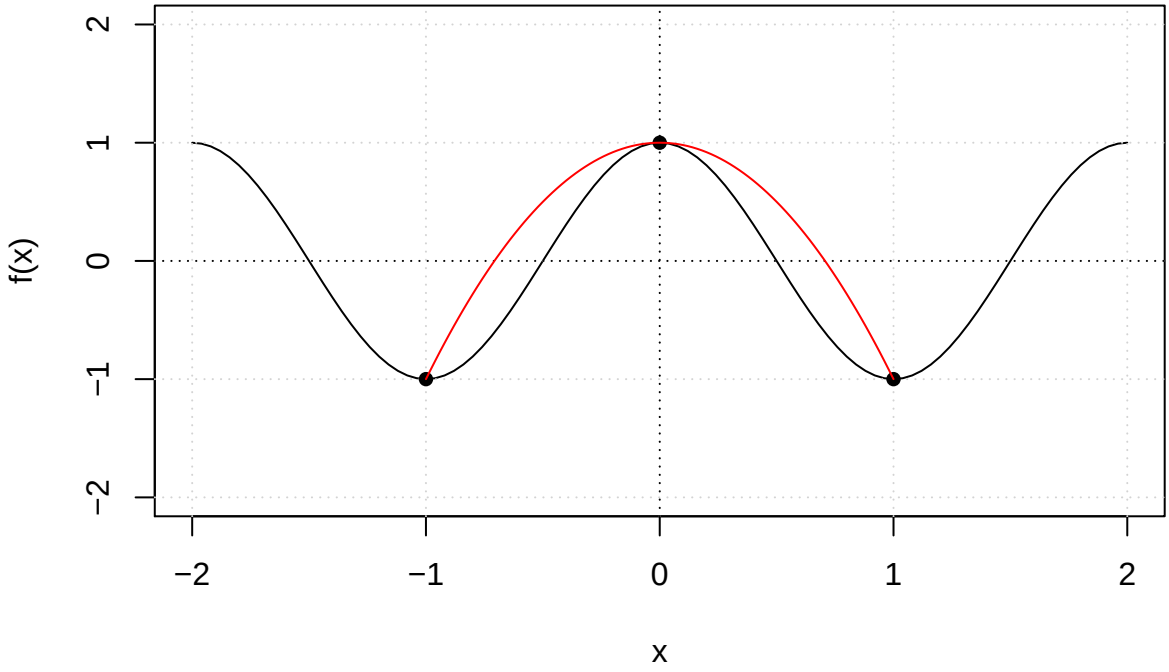
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{1}{2}\frac{f(x_n)^2 f^2(x_n)}{f'(x_n)^3}$$

which when applied to $sin(x) + 1 = 0$ converges 7 steps faster. It appears that this method is still linearly convergent, but we are get nearly 3x better with each iteration rather than 2x like just plain Newton's method above.

Here is a plot like above and I've added a horizontal line at 0.37 for reference as this appears to be the asymptotic error constant.



3. Find the polynomial of degree 2, $p_2(x)$, that interpolates $f(x) = cos(\pi x)$ at the points $x_0 = -1,\ x_1 = 0,\ x_2 = 1$ by the following methods:



A. Lagrange polynomial method

The points that the Lagrange polynomial must go through are $(-1,-1),\ (0,1),\ (1,-1)$.

$$L(x) = -1\frac{(x-0)(x-1)}{(-1-0)(-1-1)} + 1\frac{(x--1)(x-1)}{(0--1)(0-1)} + -1\frac{(x--1)(x-0)}{(1--1)(1-0)} = 1 - 2x^2$$

The Lagrange polynomial is the red line on the above plot.

B. System of linear equations method

Combining x and y into one matrix we get

$$\begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix}$$

where the first columns of X correspond (from left to right) to $x^0,\ x^1,\ x^2$ respectively and the final column are the y values.

Finding the reduced row echelon form of X and applying the same operations to Y will yield the solutions to the system.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 \end{bmatrix}$$

This corresponds to the polynomial ($p(x)$)

$$1 - 2x^2$$

C. Divided difference method

$$P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + ... + a_n(x - x_0)...(x - x_{n-1})$$

Applying this general formula to our example yields the following divided difference polynomials. The first divided differences are

$$f[x_0,\ x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = \frac{1 - (-1)}{0 - (-1)} = 2$$

and

$$f[x_1,\ x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1} = \frac{-1 - 1}{1 - 0} = -2$$

and continuing to the second divided difference (there's only one)

$$f[x_0,\ x_1,\ x_2] = \frac{f[x_1,\ x_2] - f[x_0,\ x_1]}{x_2 - x_0} = \frac{\frac{-1-1}{1-0} - 2}{1 - (-1)} = -2$$

Putting it all together we we

$$P_2(x) = -1 + 2(x + 1) - 2(x + 1)(x) = 1 - 2x^2$$

which is the same result as the above two methods.

4. Compute the determinant of the Vandermonde interpolation matrix in (3?) b directly and confirm the product formula for the determinant of such a matrix.

We found the Vandermonde matrix in 3.b but I display it here again.

$$V = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

$$det(V) = 1(0-0) - (-1)(1-0) + 1(1-0) = 2$$

The product formula for the determinant of a matrix (V) is

$$det(V) = \prod_{1 \le i \le j \le n}^{n} (x_j - x_i)$$

Carrying this out we get

$$det(V) = (1-(-1))(1-0) \times (0-(-1)) = 2$$

which is the same result. Cool!

5. Find the error of the interpolation polynomial in problem (3?) at $x = 0.5$.

The data is displayed here for convenience keeping in mind that $y_n = f(x_n) = cos(\pi x_n)$.

x	y
-1	-1
0	1
1	-1

A. Express the error in terms of a divided difference.

The "just give me one more" method. Defining the error ($E(t)$) as

$$E(t) = f(t) - P_2(t; f)$$

and setting up the problem I take the following formula as my guide, setting $t = x = 0.5$ and where we already know $P_2(x)$ from before.

Using Newton's divided differences we can show that

$$P_{n+1}(x; f) = P_n(x; f) + f[x_0, ..., x_n, t](x - x_0)(x - x_1)...(x - x_n)$$

and so

$$E(t) = f[x_0, ..., x_n, t](t - x_0)...(t - x_n)$$

Note: I found half of the above divided difference in 3.c. Now I just need to find the other half, i.e. $f[x_2, t]$, $f[x_1, x_2, t]$, and $f[x_0, x_1, x_2, t]$.

$$f[x_2,\ t] = \frac{0+1}{0.5-1} = -2$$

$$f[x_1,\ x_2,\ t] = \frac{-2-(-2)}{0.5-0} = 0$$

$$f[x_0,\ x_1,\ x_2,\ t] = \frac{0-(-2)}{0.5-(-1)} = 1.3333\bar{3}$$

So we can now put all the pieces together.

$$E(0.5) = (1.3333\bar{3})(0.5-(-1))(0.5-0)(0.5-1) = -0.5$$

B. Express the error in terms of a derivative.

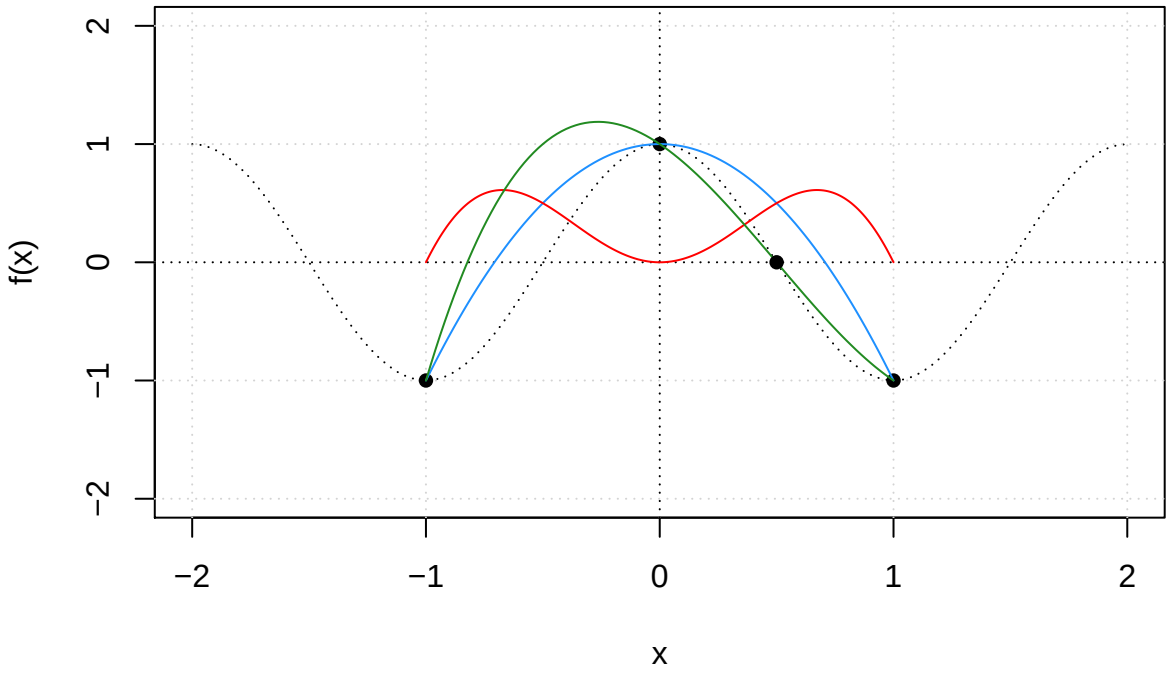
In general I'm going to define the error of the interpolating polynomial at some point x as $e_n(x) = f(x) - p_n(x)$.

Trying my best to express this as a derivative I came across the following identity.

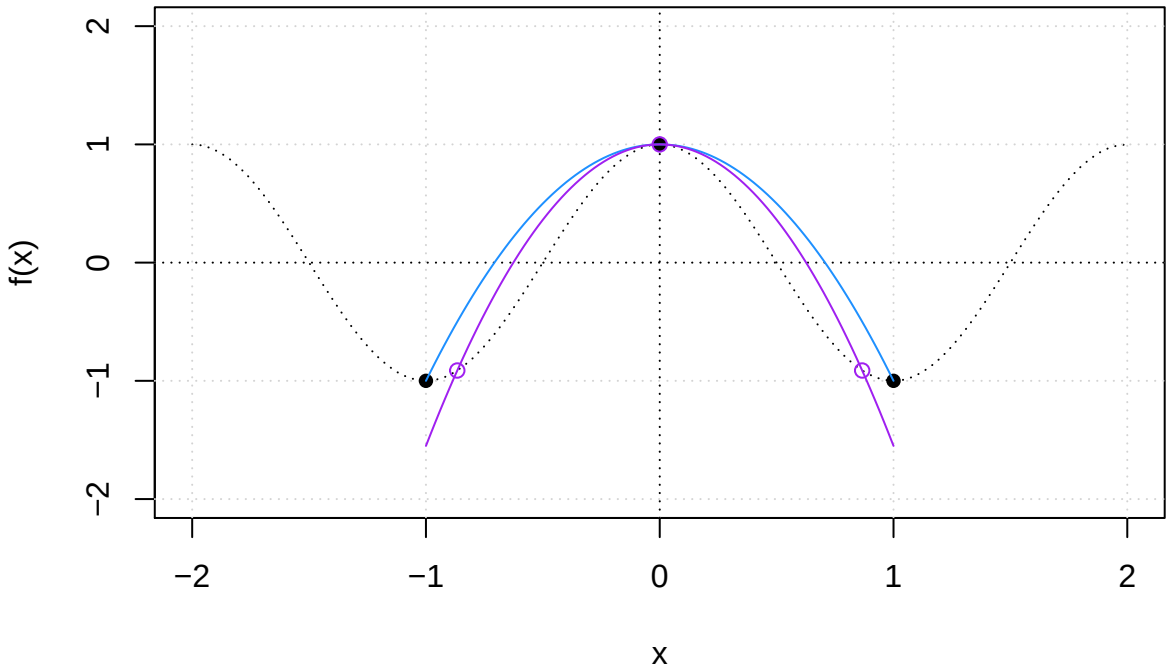
$$e_n(x) = f(x) - p_n(x) = \frac{f^{n+1}(\xi)}{(n+1)!}\prod_{j=0}^{n}(x - x_j)$$

How I understand this is there there exists some point (expressed as ξ) that is bracketed by two points ($[a, b]$) where the error at x is equal to the right hand side evaluated at some value ξ.

For visual reasons, I display a similar plot to the one above but I add the interpolation error (red) with the acutal function (in black), the interpolated function P_2 (in blue) like above, and the "just one more", P_3, (in green). We can see that the error of P_2 is indeed zero in only three places (the original 3 interpolation points).



6. Now using Chebyshev zeros on [-1, 1] and comparing the results to the the above results for $f(x) = cos(\pi x)$ where $x = 0.5$.



A graph of $cos(\pi x)$, $1 - 2x^2$, and $0.6361 - 1.8835x^2$ (interpolation of the Chebyshev zeros) is shown in purple for comparision. From visual inspection, it can be seen that the interpolation points are not the same (Chevbychev nodes are in purple too).

Since we know the value of the "just one more" at 0.5 from before, which was 0, we can compare the error (which was also 0.5) to that of this new polynomial ($T(x)$). $T(0.5) = 0.362425$ giving us an error of that same value. Less error! Way to go Chebychev!

8. Show that

$$f[x_0, ..., x_n] = \frac{f[x_0, ..., \hat{x}_j, ..., x_n] - f[x_0, ..., \hat{x}_i, ..., x_n]}{x_i - x_j}$$

9. Show that

$$f[x_0, ..., x_n] = \sum_{i=0}^{n} f(x_i) \frac{1}{\prod_{j=0,\ j \neq i}^{n} (x_i - x_j)}$$

10. Find the best approximation of $p(x) = x^3$ on the interval $[-1, 1]$ by polynomials of $degree \leq 2$ based on the following norms.

A. Least squares

B. Maximum absolute value

Appendix I

I'm including some of the code (this is `R` code) I wrote during this semester and for this exam to solve some of the problems. I'd love to keep going in the course and possibly re-write all the code in a different language, like a LISP or Haskell. Thanks for the great semester. I wish I would have taken this course earlier on in my studies here. What a ride!

```
x0 <- 2
f <- function(x) x^2 - 2 # function to find the root of
fp <- function(x) 2*x
fpp <- function(x) 2

#### Newton's Method ######
newton <- function(x0 = 0, FUN = f, FP = fp,
                   tol = .Machine$double.eps^0.5, n = 100) {
  i <- 1
  guess <- numeric()
  fx <- numeric()
  dfx <- numeric()
  error <- numeric()
  e <- tol + 1
  while (e >= tol && i <= n) {
    fx1 <- FUN(x0)
    dfx1 <- FP(x0)
    guess[i] <- x0  # store ith iteration
    fx[i] <- fx1
    dfx[i] <- dfx1 # store ith iteration
    error[i] <- abs(-fx1/dfx1) # store ith iteration
    x <- x0 - fx1 / dfx1 # calculate x
    if (is.infinite(x - x0)) stop("dx = 0.  Can't divide by 0. Pick a different x0")
    if (abs(x - x0) < tol) {
      return(cbind(guess, fx, dfx, error))
      break
    } else {
      i <- i + 1
      x0 <- x # update for next loop
    }
    #if (i == n) stop("Max number of iterations reached.")
  }
  return(cbind(guess, fx, dfx, error))
}

#### Modified Newton's #######
modifiedNewton <- function(x0 = 0, FUN = f, FP = fp, FPP = fpp,
                     tol = .Machine$double.eps^0.5, n = 100) {
  i <- 1
  guess <- numeric()
  fx <- numeric()
  dfx <- numeric()
  error <- numeric()
  e <- tol + 1
  while (e >= tol && i <= n) {
    fx1 <- FUN(x0)
    dfx1 <- FP(x0)
```

```
    df2x1 <- FPP(x0)
    guess[i] <- x0  # store ith iteration
    fx[i] <- fx1
    dfx[i] <- dfx1 * (1 - (fx1 * df2x1)/(dfx1^2)) # store ith iteration
    error[i] <- abs(-fx[i]/dfx[i]) # store ith iteration
    x <- x0 - (fx[i] / dfx[i])
    if (is.infinite(x - x0)) stop("dx = 0.  Can't divide by 0. Pick a different x0")
    if (abs(x - x0) < tol) {
      return(cbind(guess, fx, dfx, error))
      break
    } else {
      i <- i + 1
      x0 <- x # update for next loop
    }
    #if (i == n) stop("Max number of iterations reached.")
  }
  return(cbind(guess, fx, dfx, error))
}

#### Steffenson's Method #####
steffenson <- function(x0 = 0, FUN = f,
                   tol = .Machine$double.eps^0.5, n = 100) {
  i <- 1
  guess <- numeric()
  fx <- numeric()
  gx <- numeric()
  error <- numeric()
  e <- tol + 1
  while (e >= tol && i <= n) {
    fx1 <- FUN(x0)
    gx1 <- (FUN(x0 + fx1) - fx1) / fx1
    guess[i] <- x0  # store ith iteration
    fx[i] <- fx1
    gx[i] <- gx1 # store ith iteration
    error[i] <- abs(-fx1/gx1) # store ith iteration
    x <- x0 - fx1 / gx1 # calculate x
    if (is.infinite(x - x0)) stop("dx = 0.  Can't divide by 0. Pick a different x0")
    if (abs(x - x0) < tol) {
      return(cbind(guess, fx, gx, error))
      break
    } else {
      i <- i + 1
      x0 <- x # update for next loop
    }
    #if (i == n) stop("Max number of iterations reached.")
  }
  return(cbind(guess, fx, gx, error))
}

### Bisection Method ######
bisection <- function(a, b, FUN = f, tol = .Machine$double.eps^0.5, num = 30){
  if (FUN(0) == 0) stop("Zero is a root.")
  if (FUN(a) * FUN(b) > 0) stop("f(a) and f(b) can not be the same sign.")
```

```
  Fa <- FUN(a)
  Fp <- FUN((a + b) / 2) # need for control
  guess <- vector() # initialize
  a1 <- vector() # initialize
  b1 <- vector() # initialize
  i <- 1
  while (abs(Fp) > tol) {
    p <- (a + b) / 2 # middle of a and b
    Fp <- FUN(p)
    a1[i] <- a # store it
    b1[i] <- b # store it
    guess[i] <- p # store the result in a vector
    i <- i + 1 # increment for while loop
    if (Fa * Fp > 0) { # bisection method test
      a <- p # a moves if Fa and Fp are the same sign
      Fa <- Fp
    } else {
      b <- p # b moves if Fa and Fp are opposite signs
    }
    if (i == num){
      stop("Error > tol.  Max iterations reached")
    }
  }
  RESULT <- cbind(guess, a1, b1) # for printing/storing purposes
  colnames(RESULT) <- c("guess", "a", "b")
  return(RESULT)
}

##### Secant Method ####
secant <- function(FUN, p0, p1, n = 20, tol = .Machine$double.eps^0.5) {
  i <- 1
  q0 <- FUN(p0)
  q1 <- FUN(p1)
  guess <- numeric()
  iteration <- numeric()
  e <- numeric()
  while (i <= n) {
    r <- p1 - q1 * (p1 - p0)/(q1 - q0)
    e[i] <- abs(1 - abs(r)/abs(p1))
    guess[i] <- r
    iteration[i] <- i
    if (abs(r - p1) < tol) { # we found the root
      return(cbind(guess, iteration, e))
      break
    } else {
      # update everything
      i <- i + 1
      p0 <- p1
      q0 <- q1
      p1 <- r
      q1 <- FUN(r)
    }
  }
```

```
  return(cbind(guess, iteration, e))
}

##### false potition method #####
Fposition <- function(FUN = f, p0, p1, tol = .Machine$double.eps^0.5, n = 20) {
  i <- 1
  q0 <- FUN(p0)
  q1 <- FUN(p1)
  guess <- numeric()
  iteration <- numeric()
  a <- numeric() # a and b are our brackets.  Root should be within [a,b]
  b <- numeric()
  while (i <= n) {
    p <- p1 - q1 * (p1 - p0)/(q1 - q0)
    guess[i] <- p
    iteration[i] <- i
    a[i] <- p0
    b[i] <- p1
    if (abs(p - p1) < tol) { # we found the root
      return(cbind(guess, a, b, iteration))
      break
    } else {
      # update everything
      i <- i + 1
      q <- FUN(p)
      if (q * q1 < 0) {
        p0 <- p1
        q0 <- q1
      }
      p1 <- p
      q1 <- q
    }
  }
  return(cbind(guess, a, b, iteration))
}

####### Order of convergence ########
convergence <- function(x, index = 1) { # only works on output from newton2
  n <- dim(x)[1]
  p <- x[n, 1] # this is our "root", or at least what we converged to.
  ek <- abs(p - x[,1]) # kth error
  e <- vector()
  for(i in 2:length(ek)) {
    e[i-1] <- ek[i] / ek[i-1]
  }
  return(e) # asymptotic error constant
}
## Visualize order of convergence
plot.convergence <- function(x) {
  plot(convergence(x), xlab = "n", ylab = "Asymptotic Error", type = "b")
  grid()
  abline(v = 0, lty = 3); abline(h = 0, lty = 3)
}
```

```
####### Cubic, or improved Newton's for sin(x) + 1 #########
cubic <- function(x0 = 0, FUN = f, FP = NULL,
                  tol = .Machine$double.eps^0.5, n = 100) {
  if(is.null(FP)) {
    FP <- function(x, ...) {
      h <- tol^(2/3)
      (FUN(x + h, ...) - FUN(x - h, ...))/(2 * h)
    }
  }
  sx <- function(x, ...) {
    h <- tol^(2/3)
    (FP(x + h, ...) - FP(x - h, ...))/(2 * h)
  }
  i <- 1
  guess <- numeric()
  error <- numeric()
  x <- x0
  while (i <= n) {
    fx <- FUN(x0)
    dfx <- FP(x0)
    sx1 <- sx(x0)
    guess[i] <- x0  # store ith iteration
    x <- x0 - (1/dfx) * fx - (1/2) * (fx^2 * sx1)/(dfx^3) # add another "term"?
    error[i] <- abs(x - x0) # store ith iteration
    if (abs(x - x0) < tol) {
      return(cbind(guess, error))
      break
    } else {
      i <- i + 1
      x0 <- x # update for next loop
    }
    #if (i == n) stop("Max number of iterations reached.")
  }
  return(cbind(guess, error))
}

#### My version of finding all the real roots for a polynomial ######
## I was pretty proud of this when I wrote it after we touched on the idea in class.
allRealRoots <- function(a, x0, n = 20) { # a are the coeffs from 0 to n
  b <- vector()
  roots <- vector()
  for (j in 1:(length(a)-1)) { # outer loop to grab all the real roots
    k <- length(a)
    for(i in 1:n) { # inner loop to find the jth real root
      h <- horner1(a, x0) # the numerator of Newton's method
      if(abs(h$x) <= 1e-10) { # test for root
        message(paste(x0, "is a root"))
        a <- rev(h$b)[-1] # gotta knock off a degree!
        roots[j] <- x0 # store the jth root
        break # get out of inner loop
      }
      d <- h$b[-k] # take one off cuz it's the solution to the horner poly
      h2 <- horner2(x0, rev(d)) # denominator of Newton's
```

```
      x0 <- x0 - (h$x / h2) # evaluate and update
    }
  }
  return(list(Roots = roots))
}

##### allRealRoots depends on a few other functions, horner1 and horner2 #####

horner2 <- function(x, v) { # Horner's method
  Reduce(v, right = TRUE, f = function(a, b) b * x + a)
}

horner1 <- function(a, x) { # a are the coeffs from order = 0 to n
  y <- 0
  b <- vector()
  i <- 1
  for(c in rev(a)) {
    y <- y * x + c
    b[i] <- y
    i <- i + 1
  }
  return(list(x = y, b = b))
}
```