Midterm Exam One

MAT325 Numerical Analysis

3/10/2023

Instructions

General Information: This is an open book and open note exam. You can use the textbook, lecture notes, R code/function in lecture/lab notes, and internet resources for the exam. However, you must complete the exam independently. No collaboration is allowed.

Specific Requirements:

1. Use RMarkdown or other typesetting software programs to prepare your solution.
2. The code must be commented on and included in the solution. If you use a typesetting program other than RMarkdown, please put the code in the appendix.
3. Please upload a copy of your solution in PDF format and a copy source file to the D2L drop box.
4. Please feel free to use/modify the code from the lectures and those you developed in the assignments.

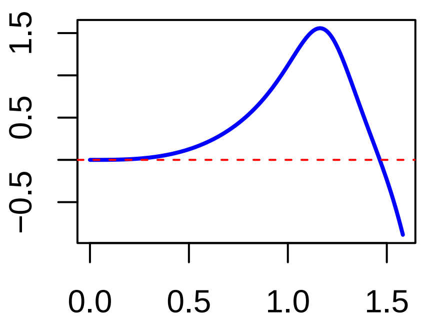
Problem Set

Problem 1: Finding the root of a non-linear equation

Consider the function $f(x) = \tan(\sin(x^3))$ over interval $[0, \sqrt{\frac{5}{2}}]$.

1. Choose 500 equally-spaced base points in the interval $[0, \sqrt{\frac{5}{2}}]$ and plot the curve of the function.

```r
xseq = seq(0, sqrt(5/2), length = 500)
yseq = tan(sin(xseq^3))
plot(xseq, yseq, type = "l", lwd = 2, col = "blue", xlab = "", ylab = "", main = "")
abline(h = 0, lty = 2, col = "red")
```



2. Find all roots of equation $\tan(\sin x^3) = 0$ mathematically over interval $\left[0, \sqrt{\frac{5}{2}}\right]$.

Solution: First of all, $\tan(\sin(x^3)) = 0$ implies $\sin(x^3) = 0$ since the domain is $\sin(x)$ is between $[-1, 1]$. Next, $\sin(x^3) = 0$ implies that $x^3 = n\pi$, or equivalently $x = \sqrt[3]{n\pi}$ for n that satisfies $[0, \sqrt{5}2]$. That is $n = 0 or 1$. Therefore, the solution to the original trigonometric equation is $x = 0$ or $\sqrt[3]{\pi} \approx 1.46459188756152$.

3. Use the Newton method to find these roots (use $TOL = 10^{-6}$) and compare these approximate roots with their corresponding true roots.

Solution: Note that

$$[\tan(\sin(x^3))]' = \sec^2(\sin(x^3)) \times \cos(x^3) \times 3x^2 = 3x^2 \times [\sec(sin(x^2))] \times \cos(x^3) = \frac{3x^2\cos(x^3)}{\cos(\sin(x^3))}$$

```
Newton.Method0 = function(fn,          # function used to define the equation
                           dfn,         # derivative of f(x)
                           x,           # initial value
                           M,           # pre-set maximum iteration
                           TOL,         # error tolerance
                           ...){
    n = 0
    ERR = abs(fn(x)/dfn(x))
    # Result table
    Result = NULL
    # Intermediate Table
    Intermediate.output = data.frame(Iteration = 1:M,
                                     Estimated.root = rep(NA,M),
                                     Absolute.error = rep(NA,M))
    # loop begins
    while(ERR > TOL){
    n = n + 1
    x = x - fn(x)/dfn(x)
    ERR = abs(fn(x)/dfn(x))
    if(ERR < TOL){
       Intermediate.output[n, ] = c(n, x, ERR)
       #Result =c(Total.Iteration = n, Estimated.Root = x, Absolute.Error = ERR)
       break
```

```
      } else{
        Intermediate.output[n, ] = c(n, x, ERR)    # store intermediate outputs
      }
      if (n ==M){
         cat("\n\nThe maximum iterations attained!")
         cat("\nThe algorithm did not converge!")
         break
      }
    }
    # out of the loop
    Intermediate.output = na.omit(Intermediate.output)
    Intermediate.output
}
```

We need to choose appropriate initial values to approximated the roots in the given interval.

```
fn = function(x) tan(sin(x^3))
dfn = function(x) 3*x^2*cos(x^3)*(cos(sin(x^3)))^(-2)
Newton.Method0( fn,             # function used to define the equation
                dfn,            # derivative of f(x)
                x =0.5,           # initial value
                M =100,         # pre-set maximum iteration
                TOL = 10^(-6)   # error tolerance
                )
```

```
##    Iteration       Estimated.root       Absolute.error
## 1          1 3.34190568154714e-01 1.11345186740816e-01
## 2          2 2.22845381413898e-01 7.42787617264010e-02
## 3          3 1.48566619687497e-01 4.95220290608269e-02
## 4          4 9.90445906266705e-02 3.30148531532535e-02
## 5          5 6.60297374734170e-02 2.20099118830958e-02
## 6          6 4.40198255903212e-02 1.46732751611864e-02
## 7          7 2.93465504291348e-02 9.78218347429543e-03
## 8          8 1.95643669548394e-02 6.52145565149123e-03
## 9          9 1.30429113033482e-02 4.34763710110892e-03
## 10        10 8.69527420223925e-03 2.89842473407933e-03
## 11        11 5.79684946815992e-03 1.93228315605328e-03
## 12        12 3.86456631210663e-03 1.28818877070221e-03
## 13        13 2.57637754140442e-03 8.58792513801475e-04
## 14        14 1.71758502760295e-03 5.72528342534316e-04
## 15        15 1.14505668506863e-03 3.81685561689544e-04
## 16        16 7.63371123379089e-04 2.54457041126363e-04
## 17        17 5.08914082252726e-04 1.69638027417575e-04
## 18        18 3.39276054835151e-04 1.13092018278384e-04
## 19        19 2.26184036556767e-04 7.53946788522557e-05
## 20        20 1.50789357704511e-04 5.02631192348371e-05
## 21        21 1.00526238469674e-04 3.35087461565581e-05
## 22        22 6.70174923131162e-05 2.23391641043721e-05
## 23        23 4.46783282087441e-05 1.48927760695814e-05
## 24        24 2.97855521391627e-05 9.92851737972091e-06
## 25        25 1.98570347594418e-05 6.61901158648061e-06
## 26        26 1.32380231729612e-05 4.41267439098707e-06
## 27        27 8.82534878197415e-06 2.94178292732472e-06
## 28        28 5.88356585464943e-06 1.96118861821648e-06
## 29        29 3.92237723643295e-06 1.30745907881098e-06
## 30        30 2.61491815762197e-06 8.71639385873990e-07
```

Since the curve of the function near the 9 is flat, it takes more iterations to find the root with a given accuracy.

```
fn = function(x) tan(sin(x^3))
dfn = function(x) 3*x^2*cos(x^3)*(cos(sin(x^3)))^(-2)
Newton.Method0( fn,             # function used to define the equation
                dfn,            # derivative of f(x)
                x =1.5,           # initial value
                M =100,         # pre-set maximum iteration
                TOL = 10^(-6)   # error tolerance
                )
```

```
##   Iteration   Estimated.root        Absolute.error
## 1         1 1.46602196337120 1.42864084495302e-03
## 2         2 1.46459332252625 1.43496331959950e-06
## 3         3 1.46459188756293 1.40607166848600e-12
```

Problem 2. Approximation with interpolated Polynomials

The Laguerre polynomial that has applications in differential equations and physics (particularly in quantum mechanics) that is is defined by

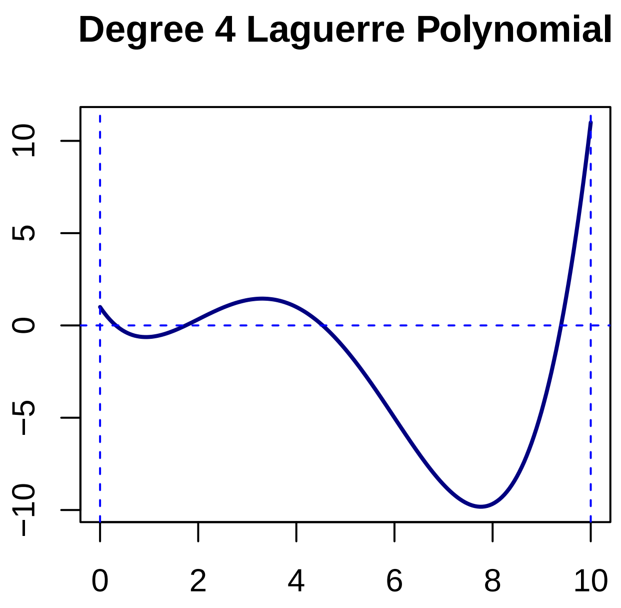
$$L_n(x) = \sum_{k=0}^{n} \binom{n}{k} \frac{(-1)^k}{k!} x^k$$

In this problem, we only consider the degree 4 Laguerre polynomial

$$L_4(x) = \frac{x^4}{24} - \frac{2x^3}{3} + 3x^2 - 4x + 1$$

over interval $[0, 10]$.

```
xx= seq(0, 10, length = 200)
yy = xx^4/24 - (2/3)*xx^3 + 3*xx^2 - 4*xx + 1
plot(xx, yy, type = "l", lwd = 2, col = "navy",
     xlab = "", ylab = "",
     main = "Degree 4 Laguerre Polynomial")
abline(v=c(0, 10), h = 0, lty = 2, col = "blue")
```



The purpose of this problem is to assess the performance of the following three different approximated polynomials:

1. The degree 4 Taylor polynomial expanded at $x = 7.75$.

Solution: First of all, the given degree 4 Laguerre polynomial should be identical to the 4-th order Taylor polynomial since there is no approximation involved. They are the sample polynomial with different forms. The actual value may be slightly different due to rounding errors. Keep this in mind, you will easily check the mathematics in your code.

```r
taylor = function(x0,         # given x value being expanded on
                  pred.x     # x value we are going to calculate
                  ){
     fn0 = (x0^4)/24- ((2*x0^3)/3) + 3*x0^2 - 4*x0 + 1
     #determining all levels of the polynomial
     first.d = (x0^3)/6 - 2*x0^2 + 6*x0 - 4
     second.d = (x0^2)/2 - 4*x0 + 6
     third.d = x0 - 4
     fourth.d = 1
     #
     diff = pred.x - x0
     #calculating difference
     pred = fn0 + first.d*diff + (second.d/2)*diff^2 + (third.d/6)*diff^3 + (fourth.d/24)*diff^4
     pred
}

taylor(x=7.75,pred.x=3)
```

```
## [1] 1.37499999999995
```

2. The Newton interpolated polynomial using unequally-spaced nodes $x = 0.3, 0.95, 3.3, 7.75, 9.4$

```
Divided.Dif = function(
        vec.x,          # input nodes:
        vec.y = NULL,   # one of vec.y and fn must be given
        fn = NULL,
        pred.x          # scalar x for predicting pn(pred.x)
         ){
   n = length(vec.x)
   if (length(vec.y) == 0) vec.y = fn(vec.x) #
   node.x = vec.x
   A = matrix(c(rep(0,n^2)), nrow = n, ncol = n, byrow = TRUE)
   A[1,] = vec.y     # fill the first row with vec.y
   #
   for(i in 2:(n)){
     for(j in 1:(n-i+1)){
      denominator = vec.x[j] - vec.x[j+1+(i-2)]
      numerator = A[i-1,j]- A[i-1,j+1]
      A[i,j] = numerator/denominator
      }
    }
  A
}
####
Newton.Interpolation = function( vec.x,            # input interpolation nodes
                               vec.y = NULL,
                               fn = NULL,        # either vec.y or fn must be provided
                               pred.x            # VECTOR INPUT!!!
                             ){
   if(length(vec.y) ==0) vec.y = fn(vec.x)
   DivDif = Divided.Dif(vec.x, vec.y)[,1]       # the values in the first column of the div dif matrix
   n = length(vec.x)
   ############
   m = length(pred.x)
   NV = rep(0, m)                   # values of Nn(pred.x)
   for(k in 1:m) {
   ################
   Nn = vec.y[1]                    # f[xo]
   for (i in 1:(n-1)){              # Must be n - 1 according to the last term in the polynomial
     cumProd = 1                    # initial value to calculate the cumulative product
     for(j in 1:i){                 # forward difference formula
       cumProd = cumProd*(pred.x[k]-vec.x[j])   # updating the cumulative product in the inner loop
     }
      Nn = Nn + DivDif[i+1]*cumProd    # adding high order terms alliteratively to the Nn(x)
    }
    NV[k] = Nn                                # return the value the Newton polynomial
   }
 NV
}
```

```
xx = c(0.3 , 0.95, 3.3,  7.75, 9.4)
yy = xx^4/24 - (2/3)*xx^3 + 3*xx^2 - 4*xx + 1
##
pred.x = 3
pred.NIP = Newton.Interpolation(vec.x = xx,
                    vec.y = yy,
                    pred.x = 3 )

pander(cbind(pred.x = pred.x, pred.NIP=pred.NIP))
```

pred.x	pred.NIP
3	1.375

3. The Newton interpolated polynomial using equally-spaced nodes $x = 1, 3, 5, 7, 9$

```
xx = c(1, 3, 5, 7, 9)
yy = xx^4/24 - (2/3)*xx^3 + 3*xx^2 - 4*xx + 1
##
pred.x = 3
pred.NIP = Newton.Interpolation(vec.x = xx,
                    vec.y = yy,
                    pred.x =3)

pander(cbind(pred.x = pred.x, pred.NIP=pred.NIP))
```

pred.x	pred.NIP
3	1.375

Evaluate the above three approximated polynomials and the original Laguerre polynomial at $x = 3$ and compare the three approximated values with the true value obtained from the original Laguerre polynomial. Which polynomial gives the best approximation?

Answer: Mathematically, all three resulting polynomials are identical. Because we did not use specify any arithmetic with the given number of digits. The computational results are essentially the same ()

Problem 3. Application - Approximating Unknown Function and Making Prediction

The table below gives the actual thermal conductivity data for the element mercury. The objective is to find the analytic expression that approximates the relationship between temperature and pressure.

Temperature (Fahrenheit), T	220	230	240	250	260	270	280	290
Pressure (Pound), P	17.19	20.78	24.97	29.82	35.42	41.85	49.18	57.53

Use the Newton interpolation and all given data points to construct a polynomial of degree 8 and use this interpolating polynomial to **predict** the corresponding pressures for temperatures $T = 235, 255, 279, 295$.

Solution:

```
xx = c(220,  230,  240,  250,  260,  270, 280, 290, 300)
yy = c(17.19,20.78,24.97,29.82,35.42,41.85,49.18,57.53,66.98)
##
pred.x = c(235, 255, 279, 295)
pred.NIP = Newton.Interpolation(vec.x = xx,
                    vec.y = yy,
                     pred.x =pred.x)

pander(cbind(pred.x = pred.x, pred.NIP=pred.NIP))
```

pred.x	pred.NIP
235	22.8
255	32.52
279	48.4
295	62.13

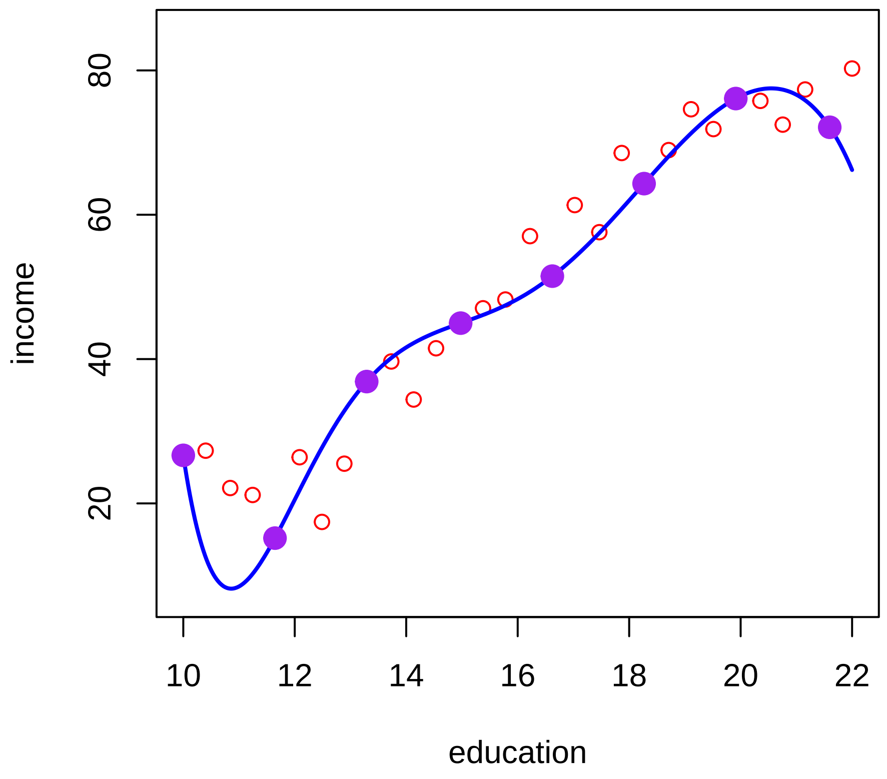
Extension Problem 3: Real-world Applications

- **Principle of Parsimony**

In real-world applications, we usually avoid high-degree polynomials for prediction just because the complex models usually generate more predictive errors (the issue of over-fitting). Too few nodes will also generate large prediction errors (issue of under-fitting)!

See the following example.

```
income = read.csv("https://raw.githubusercontent.com/pengdsci/MAT143/main/w07/Income.csv")
education = income$Education
income = income$Income
## Newton's Approximation
pred.x = seq(10,22, length = 400)
ID = seq(1, 29, by=4)                                  # sample points for interpolating the approximat
pred.NIP = Newton.Interpolation(vec.x = education[ID],
                                vec.y = income[ID],
                               pred.x = pred.x)
pred.y = pred.NIP
##plotting
ylm = range(pred.y)    # Setting up the limit of y-axis
plot(education, income, pch = 21, ylim=c(0.9*ylm[1], 1.1*ylm[2]), col = "red", main = "")    # plotting
lines(pred.x, pred.y, lwd = 2, col = "blue")     # curve the approximated interpolated polynomial
points(education[ID], income[ID], pch = 19, cex = 1.5, col = "purple")  # plot the points used in the i
```

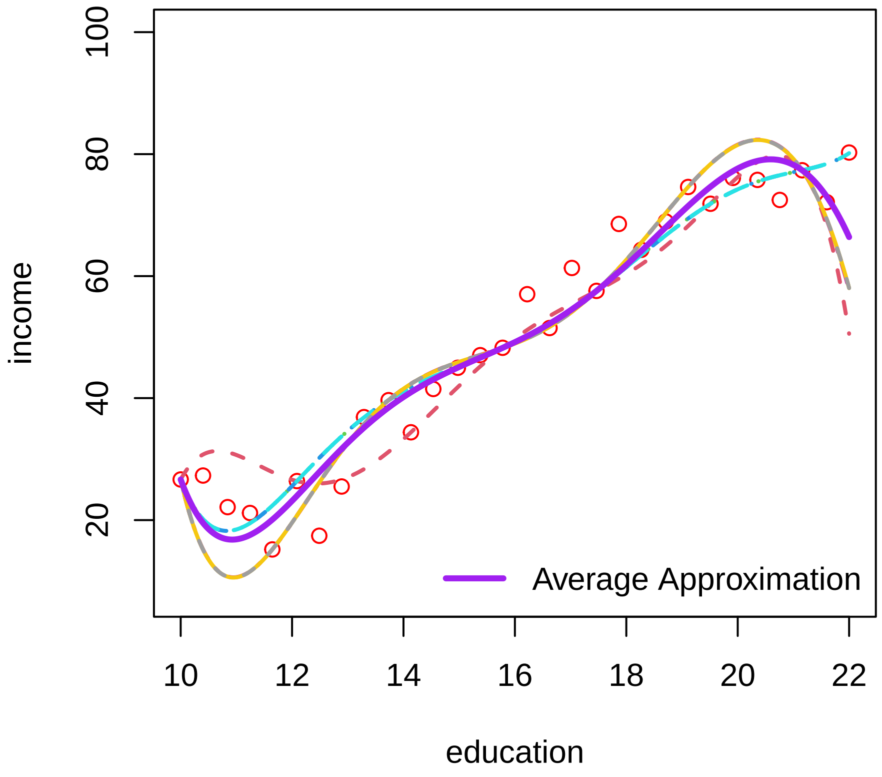


- **Single Approximation Is NOT Enough!**

Averaging multiple approximations to improve stability and accuracy is a common practice in real-world applications.

In the above example, there are a total of 29 data points. If keeping the first and the last data points (Caution, x-coordinates are sorted in ascending order!), and using the **leave-k-out** approach, we can create multiple approximations based on the same data given data points.

```
pred.x = seq(10,22, length = 400)
ID = seq(1, 28, by=4)      # sample points for interpolating the approximated polynomials
pred.NIP = Newton.Interpolation(vec.x = education[ID],
                                vec.y = income[ID],
                                pred.x = pred.x)
pred.y = pred.NIP
##plotting
ylm = range(pred.y)     # Setting up the limit of y-axis
plot(education, income, pch = 21, ylim=c(0.9*ylm[1], 100), col = "red", main = "")     # plotting the or
## leave-three-out
PRED = matrix(0, ncol = 400, nrow = 7)
for(i in 2:8){
sid = c(-i, -(8+i), -(18+i))
education0 = education[sid]
income0 = income[sid]
pred.NIP = Newton.Interpolation(vec.x = education0[ID],
                                vec.y = income0[ID],
                                pred.x = pred.x)
PRED[i-1,] = pred.NIP
pred.y = pred.NIP
lines(pred.x, pred.y, lwd =2, lty = i, col = i)      # curve the approximated interpolated polynomial
}
pred.avg = apply(PRED, 2, mean)
lines(pred.x, pred.avg, lwd = 3, col = "purple")
legend("bottomright","Average Approximation", lwd=3, col="purple", bty = "n")
```



- **Performance Assessment**

If $f(x)$ is unknown, for a given value $x = x_0$, how do we know $P_n(x_0)$ is good or bad? In other words, how do we know whether $P_n(x)$ is close to the UNKNOWN function $f(x)$?

One strategy for addressing the performance assessment is described in the following:

1. hold up a subset of data points: $(x_1^{\text{hold}}, y_1^{\text{hold}}), (x_2^{\text{hold}}, y_2^{\text{hold}}), \cdots, (x_k^{\text{hold}}, y_k^{\text{hold}})$.
2. use the rest of the data points to find the averaged interpolated polynomial.
3. use the hold-up data points to assess the approximation error:

$$\text{Approx. error} = \sqrt{\frac{1}{k}\sum_{j=1}^{k}\left[y_j^{\text{hold}} - P_n(x_j^{\text{hold}})\right]^2}.$$

That is, the mean square error is a good measure to assess the goodness of approximation.