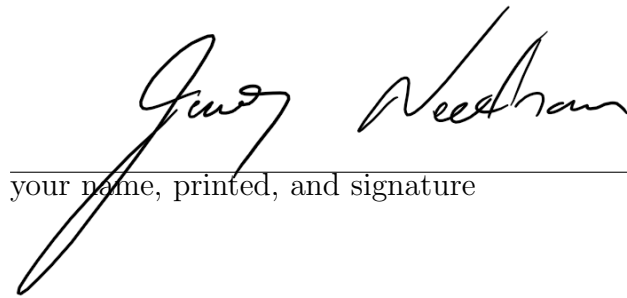


MAE 3210 - Spring 2019 - Project 1 Honor Pledge

REMINDER: Projects are to be treated as take-home exams with NO collaboration or discussion with other students.

Along with all of your code and results, you must sign, scan, and submit the following form in order to receive credit for project 1. By signing this form you are pledging to Utah State University that you abided by the policy written above in the completion of this project. That is, you did NOT discuss the problems assigned, your code, or any challenges inherent to project 1 with other USU students in or out of the class. You are only allowed to ask questions related to this project of the course instructor (Geordie Richards), course TA (Nate Hall), and course grader (Jacob Bryan).

That is, you treated project 3 as a **TAKE-HOME EXAM**. Violation of this policy will be treated as plagiarism and appropriate disciplinary action will be implemented, as outlined in the USU University Policies found on the course syllabus.

A handwritten signature in black ink, appearing to read "Jacob Bryan", is written over a horizontal line.

your name, printed, and signature

MAE 3210 - Spring 2019 - Project 1

Project 1 is due **online** through Canvas by 11:59PM on Thursday, February 21.

IMPORTANT REMARKS (Please read carefully):

- **Projects are to be treated as take-home exams with NO collaboration or discussion with other students.** Plagiarism will be monitored and considered as cheating. **In addition to your project PDF, you are required to sign and submit an honor pledge which can be found adjacent to the project handout, under assignments in Canvas.** Each project is worth 50 points, and a 10 point loss per day will apply to late projects.
- You are required to submit code for all functions and/or subroutines built to solve these problems, which is designed to be easy to read and understand, in your chosen programming language, **and which you have written yourself.** The text from your code should both be copied into a single PDF file submitted on canvas. **Your submitted PDF must also include responses to any assigned questions, which for problems requiring programming should be based on output from your code.** For example, if you are asked to find a numerical answer to a problem, the number itself should be included in your submission.
- In addition to what is required for homework submission, for each numerical answer you reach based on running code you have written yourself, **your submitted project PDF must include a copy of a screenshot** that shows your computer window after your code has been executed, with the numerical answer displayed clearly on the screen.
- Note that, in problem 2 below, you are asked to use code that you have already written for homework 3. **You need to submit any code that you use for this project, including any code that you already wrote and submitted for homework 3.**
- If you are asked to use code that you have written yourself to solve a given problem (e.g. in 1(c), 2(c),(d) below), but you are unable to get that code working, you may choose, instead, to submit numerical answers based on running built-in functions (e.g. determinant computation, root finders, algebraic solvers already available in MATLAB), and you will receive partial credit. **However, you are required to write all code yourself, without relying on built-in functions, in order to get full points.**

1. The general form of a three-dimensional stress field in a continuum material is represented in the standard coordinate system by a 3×3 matrix

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{pmatrix}$$

where the diagonal terms represent tensile or compressive stresses and the off-diagonal terms represent shear stresses. The *principal stresses* λ_1, λ_2 , and λ_3 are defined as the eigenvalues of $\boldsymbol{\sigma}$, which correspond to the roots of the characteristic equation $c(\lambda) = 0$, where $c(\lambda) = \det(\lambda \mathbf{I} - \boldsymbol{\sigma})$ is the characteristic polynomial and of $\boldsymbol{\sigma}$, and \mathbf{I} is the 3×3 identity matrix.

- (a) Write a program that takes the entries of a three-dimensional stress field $\boldsymbol{\sigma}$ as input and outputs a_0, a_1 , and a_2 , the *coefficients* of the characteristic polynomial $c(\lambda) = \det(\lambda \mathbf{I} - \boldsymbol{\sigma}) = \lambda^3 + a_2 \lambda^2 + a_1 \lambda + a_0$.
- (b) Consider the three-dimensional stress field given in units of MPa by

$$\boldsymbol{\sigma} = \begin{pmatrix} 9 & 13 & 24 \\ 13 & 6 & 14 \\ 24 & 14 & 15 \end{pmatrix}.$$

Use the program you wrote for problem 1(a) to obtain the characteristic equation $c(\lambda) = 0$, and solve for the three principal stresses of $\boldsymbol{\sigma}$ in two ways:

- (i) Using the *modified secant* method (see page 161 of course text).
- (ii) Using the *fixed point* method.

HINT: You may need to modify the equation $c(\lambda) = 0$ nearby each root in order to force the fixed point method to converge. It may be useful to recall: what property does $c'(\lambda)$ need to satisfy nearby a given root in order for the fixed point method to converge?

- (c) Draw (by computer or by hand) *iteration cobwebs* to illustrate that the fixed point method you applied in problem 1(b) converges, and attach these drawings to your submission.
2. Linear algebraic equations can arise in the solution of differential equations. For example, the following *heat equation* describes the equilibrium temperature $T = T(x)$ ($^{\circ}\text{C}$) at a point x (in meters m) along a long thin rod,

$$\frac{d^2 T}{dx^2} = h'(T - T_a), \tag{1}$$

where T_a ($^{\circ}\text{C}$) denotes the temperature of the surrounding air, and h' (m^{-2}) is a heat transfer coefficient. Assume that the rod is 12 meters long (i.e. $0 \leq x \leq 12$)

and has boundary conditions imposed at its ends given by $T(0) = 20^\circ\text{C}$ and $T(12) = 160^\circ\text{C}$.

- (a) Using standard ODE methods, which you do not need to repeat here, the general form of an analytic solution to (1) can be derived as

$$T(x) = A + Be^{\lambda x} + Ce^{-\lambda x}, \quad (2)$$

where A , B , C , and λ are constants. With $T_a = 10^\circ\text{C}$ and $h' = 0.02\text{ m}^{-2}$, plug the formula (2) into equation (1) and analyze the results to solve for A and λ .

- (b) Next, impose the boundary conditions $T(0) = 20^\circ\text{C}$ and $T(12) = 160^\circ\text{C}$ to derive a system of 2 linear algebraic equations for B and C . Provide the system of two equations you have derived.
- (c) Use one of the numerical algorithms you developed for homework 3 (Gauss elimination *or* LU decomposition) to solve the algebraic system you derived in question 2(b) above, and obtain an *analytic solution* to (1) of the form (2). By *analytic solution* we mean an explicit solution to equation (1) which is valid for each x in the interval $[0, 12]$.
- (d) Next we will discuss how to obtain a *numerical solution* to (1). That is, we will seek to obtain an approximate solution to (1) which describes the value of T at 5 intermediate points inside the interval $[0, 12]$. More precisely, the equation (1) can be transformed into a linear algebraic system for the temperature at 5 interior points $T_1 = T(2)$, $T_2 = T(4)$, $T_3 = T(6)$, $T_4 = T(8)$, and $T_5 = T(10)$ by using the following finite difference approximation for the second derivative at the i^{th} interior point,

$$\frac{d^2T_i}{dx^2} = \frac{T_{i+1} - 2T_i + T_{i-1}}{(\Delta x)^2}, \quad (3)$$

where $1 \leq i \leq 5$, $T_0 = T(0) = 20^\circ\text{C}$, $T_6 = T(12) = 160^\circ\text{C}$, and Δx is the spacing between the interior points. With $T_a = 10^\circ\text{C}$ and $h' = 0.02\text{ m}^{-2}$, use (3) to rewrite (1) as a system of 5 linear algebraic equations for the unknowns T_1, T_2, T_3, T_4 and T_5 . Provide the system of 5 equations you have derived.

- (e) Use one of the numerical algorithms you developed for homework 3 (Gauss elimination *or* LU decomposition) to solve the system derived in question 2(d) above. Validate your numerical solution by comparison to the analytic solution that you obtained in 2(c).
- (f) Write a function that takes as input the number of interior nodes n desired for your numerical solution (i.e. $n = 5$ in 2(d) above), and outputs the numerical solution to (1) in the form of the interior node values $T_1 = T(\Delta x)$, $T_2 = T(2\Delta x), \dots, T_n = T(n\Delta x)$.

- (g) Produce and submit three plots that compare your analytic solution to (1) derived in question 2(b) to the numerical solution generated in question 2(f) for $n = 5$, $n = 10$, and $n = 20$, respectively.

```

"""
Project 1 Part 1 (a)
This program takes a three-dimensional stress field as specified by the user
and outputs the characteristic polynomial.
@author: Jacob Needham
"""

#getting values of matrix
print("Please input the values of the 3x3 matrix one value at a time")
m = []
for row in range(3):
    m.append([0,0,0])

for row in range(3):
    for cell in range(3):
        m[row][cell] = int(input())

#Invariants of the 3x3 matrix
# I1 = trace(matrix)
I1 = m[0][0] + m[1][1] + m[2][2]
# I2 = 1/2*(trace(matrix)^2+trace(matrix^2))
I2 = m[0][0]*m[1][1] + m[1][1]*m[2][2] + m[0][0]*m[2][2] \
- m[0][1]*m[1][0] - m[1][2]*m[2][1] - m[0][2]*m[2][0]
# I3 = determinat(matrix)
I3 = -m[0][2]*m[1][1]*m[2][0] + m[0][1]*m[1][2]*m[2][0] \
+ m[0][2]*m[1][0]*m[2][1] - m[0][0]*m[1][2]*m[2][1] - m[0][1]*m[1][0]*m[2][2] \
+ m[0][0]*m[1][1]*m[2][2]

#Printing the characteristic polynomial
print("For the entered matrix:")
for i in m:
    print(i)
print("The characteristic equation for is:")
print("0 = x^3 - ",I1,"*x^2 + ",I2,"*x - ",I3,sep="")

```

OUTPUT

Please input the values of the 3x3 matrix one value at a time

1

2

3

4

5

6

7

8

9

For the entered matrix:

[1, 2, 3]

[4, 5, 6]

[7, 8, 9]

The characteristic equation for is:

0 = x³ - 15*x² + -18*x - 0

```

"""
Project 1 Part 1 (b)
This program takes the three-dimensional stress field provided by the project
and outputs the eigenvalues of the matrix. Both the modified secant method and
the fixed point method are employed to find the roots of the characteristic
equation  $x^3 - I_1x^2 + I_2x - I_3$ 
@author: Jacob Needham
"""

import numpy as np
import matplotlib.pyplot as graph

#Matrix given in problem statement
m = [[9,13,24],
      [13,6,14],
      [24,14,15]]

#Invariants of the 3x3 matrix
# I1 = trace(matrix)
I1 = m[0][0] + m[1][1] + m[2][2]
# I2 = 1/2*(trace(matrix)^2+trace(matrix^2))
I2 = m[0][0]*m[1][1] + m[1][1]*m[2][2] + m[0][0]*m[2][2] \
- m[0][1]*m[1][0] - m[1][2]*m[2][1] - m[0][2]*m[2][0]
# I3 = determinat(matrix)
I3 = -m[0][2]*m[1][1]*m[2][0] + m[0][1]*m[1][2]*m[2][0] \
+ m[0][2]*m[1][0]*m[2][1] - m[0][0]*m[1][2]*m[2][1] - m[0][1]*m[1][0]*m[2][2] \
+ m[0][0]*m[1][1]*m[2][2]

#Charicteristic polynomial to solve for eigenvalues
#the most important thing for this project is noticing that the fixed point
#method will diverge if, roughly speaking, the funtion is too steep
#around a aroot. Dividing the equation by a value larger than the largest
#eigenvalue will squash the funciton and potntiallall allow it to converge to a
#point.
def function(x):
    y = abs((x**3 - I1*x**2 + I2*x - I3)/ (-3*I3))
    return y

#Creating a graph of the function
def plotSpace():
    #setting up function
    x = np.arange(-10000,10000,.01)
    y = function(x)
    graph.ylim(-20,20)
    graph.xlim(-15,50)
    #plotting function
    graph.plot(x, y)

    #setting up graph
    graph.xlabel('x - axis')
    graph.ylabel('y - axis')
    graph.title('Project 1 Problem 1 B')

    #plotting axis
    graph.plot(x, x*0 + 0 , linewidth = .5, color = 'black')

```

```

graph.plot(x*0 + 0, x, linewidth =.5, color = 'black')

#grpahing
graph.show()

plotSpace()

#The modified secant method takes an inital guess x0 and evaluates the
#function a small distance, d, away. It solves for the intercept of the
#line created and then chooses that value for the new input.
def modSecant(x0,es,iMax):
    ea=es
    xR = x0
    iCount = None
    d=.01
    for iCount in range(0,iMax):
        iCount +=1
        xR = x0 - (d*xR*function(x0))/(function(x0 + (d*x0))-function(x0))
        if xR != 0:
            ea = abs((xR-x0)/xR)*100
        if ea<es or iCount > iMax:
            break
        x0 = xR

    print("Number of iterations: ", iCount)
    print("An eigenvalue is: ", xR)
    print("The error is: ", ea*100 , "%")
    print("")

modSecant(1,.001,30)
modSecant(-15,.001,30)
modSecant(40,.001,30)

#to create space between modified secant and fixed point
print()
print()

#The fixed point method evaluates the funtion at an inital guess x0 and
#adds to the solution the inital guess. This process is repeated until
#the root is found.
def fixedPoint(x0,es,iMax):
    xR = x0
    iCount = 0
    ea = es
    #eaOLD = None
    for iCount in range(iMax):
        iCount += 1
        xRold = xR
        xR = xRold + function(xRold)
        #print(xR) #helpful for making cobwebs
        #eaOLD = ea
        if xR != 0:
            ea = abs((xR-xRold)/xR)
        #if (eaOLD - ea)/ea < 1000: #catches a diverging function
        #    print("Its diverging!")

```



```

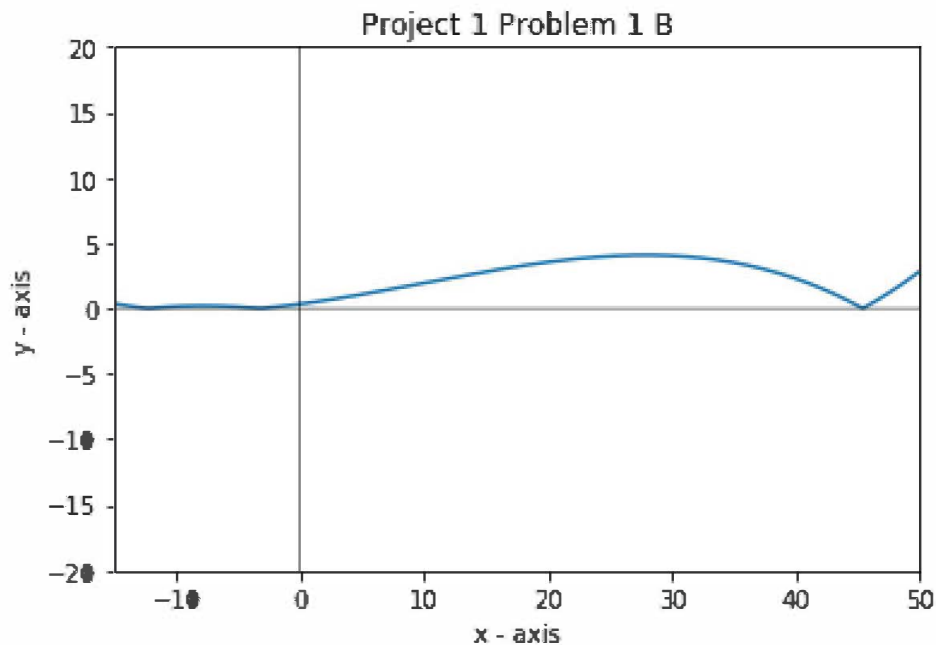
        # break
        if ea<es or iCount > iMax:
            break

    print("Number of iterations: ", iCount)
    print("An eigenvalue is: ", xR)
    print("The error is: ", ea*100 , "%")
    print("")

fixedPoint(-4,.001,30)
fixedPoint(-40,.001,30)
fixedPoint(40,.0001,30)

```

OUTPUT



```

Number of iterations: 14
An eigenvalue is: -3.228658021244206
The error is: 0.041364705640822286 %

```

```

Number of iterations: 5
An eigenvalue is: -12.20841677996702
The error is: 0.015464843757048496 %

```

```

Number of iterations: 5
An eigenvalue is: 45.437101760568424
The error is: 0.007588722040584016 %

```

```

Number of iterations: 30
An eigenvalue is: -3.293825602970278
The error is: 0.17386367747068585 %

```

```

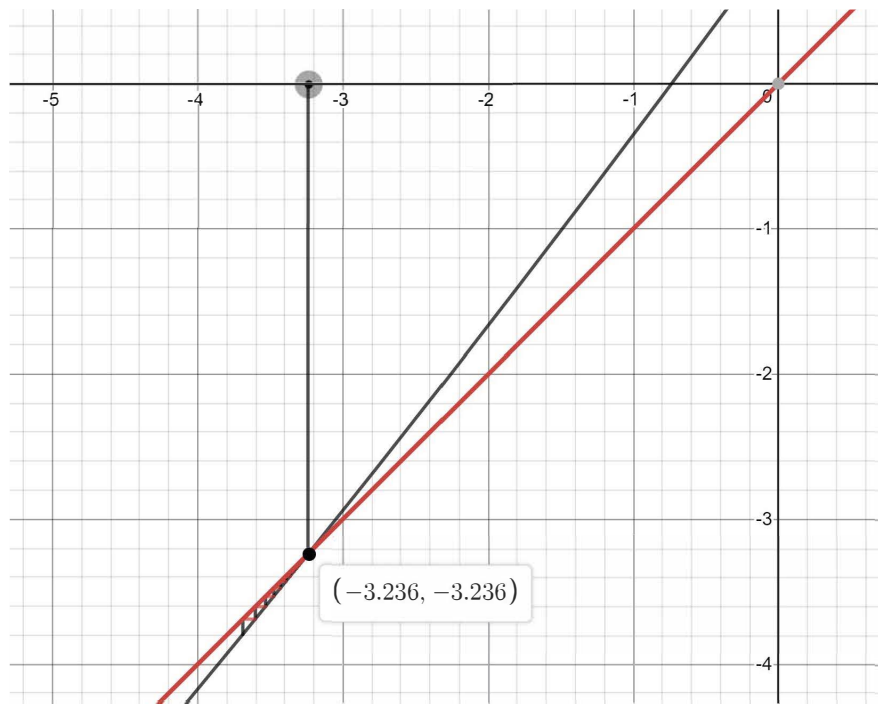
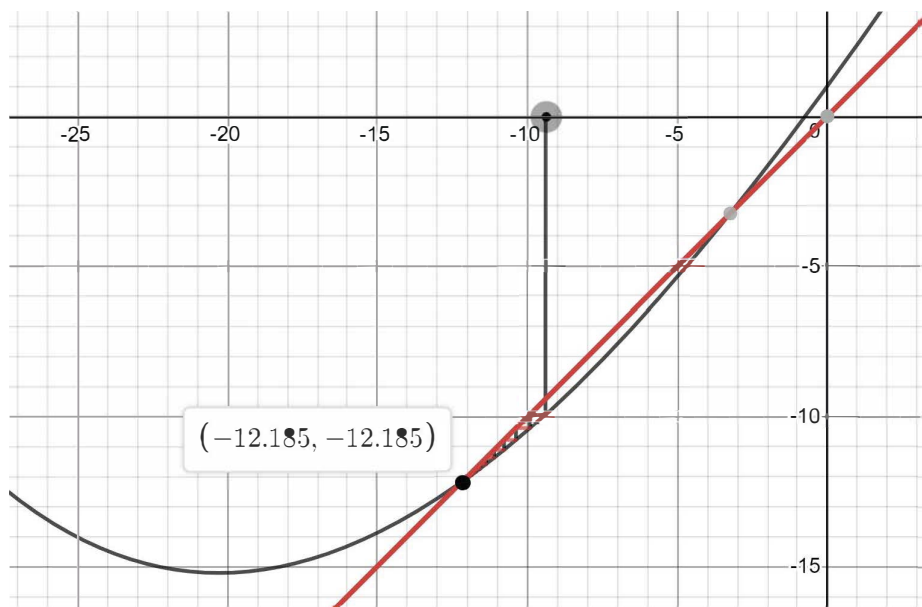
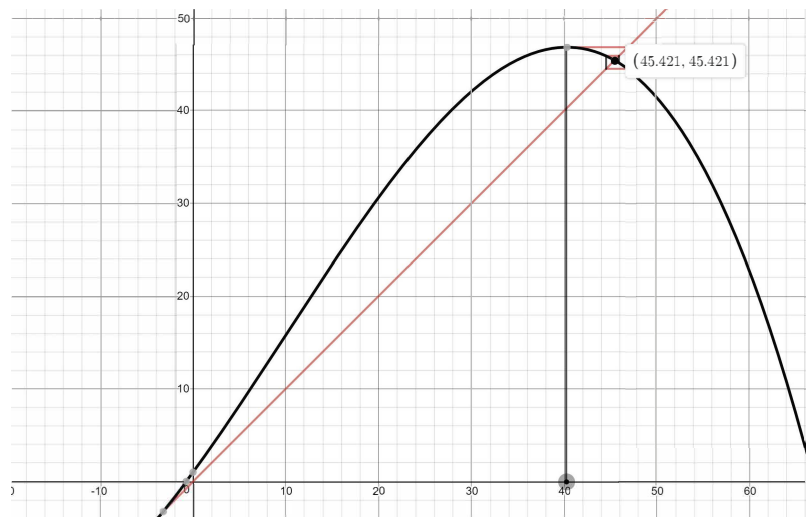
Number of iterations: 30
An eigenvalue is: -12.427967957187443
The error is: 0.19492603858826005 %

```

```

Number of iterations: 11
An eigenvalue is: 45.43457353780319
The error is: 0.0060757546262460655 %

```



```

"""
Project 1 Part 2 (c)
A solution to constants B and C from the system of equations found solving the
heat equation using ODE techniques.
@author: Jacob Needham
"""

import math
#System of equations from solution to ODE:
# 30 = B + C
# 170 = B * e^(sqrt(.02)*12) + e^(-sqrt(.02)*12)

#Loading a matrix and solution vector with above equations
matrix = [[1,1],
           [math.exp(math.sqrt(.02)*12),math.exp(-math.sqrt(.02)*12)]]
b = [30,170]

#Creating necessary vectors to use Gauss Elimination
n=2
o=None
#solution space
Xn = [None]*n
#lists to hold the maxes and ratios of each row
theMax = [None]*n
ratio = [None]*n
k = i = j = 0

def partialPivoting(o,matrix,i):
    for row in matrix:
        theMax[o] = max(row, key=abs)
        o += 1
    for m in range(len(matrix)):
        ratio[m] = abs(matrix[m][m]/theMax[m])
    if i < n:
        if ratio[i-1] < max(ratio[i:], key=abs):
            swapIndex = max(ratio[i:])
            matrix[i-1] = matrix[swapIndex]

def forwardElim(k,i,j,matrix):
    for k in range(1,n):
        for i in range(k+1,n+1):

            #partial pivoting
            o=0
            partialPivoting(o,matrix,i)

            #forward elimination
            factor = matrix[i-1][k-1]/matrix[k-1][k-1]
            for j in range(k+1,n+1):
                matrix[i-1][j-1] = matrix[i-1][j-1] - factor * matrix[k-1][j-1]
            b[i-1] = b[i-1] - factor * b[k-1]

def backwardSub(Xn,b,matrix):
    Xn[n-1] = b[n-1]/matrix[n-1][n-1]
    for i in range(n-1,0,-1):
        Sum = b[i-1]
        for j in range(i+1,n+1):

```

```

        Sum = Sum - matrix[i-1][j-1] * Xn[j-1]
        Xn[i-1] = Sum/matrix[i-1][i-1]

def main(o,i,j,k,matrix,b,Xn):
    #calling each function
    forwardElim(k,i,j,matrix)
    backwardSub(Xn,b,matrix)

    #printing answers
    print("OUTPUT:")
    print("Coefficients A and B are:")
    print("A:", Xn[0])
    print("B:", Xn[1])

main(o,i,j,k,matrix,b,Xn)

```

```

OUTPUT:
Coefficients A and B are:
A: 31.187623614686984
B: -1.1876236146869834

```

```

"""
Project 1 Part 2 (e)
This section of code finds a numerical approximation to the finite difference
approximation equation with a spacing of 5 intervals (ranging from 2 to 10).
@author: Jacob Needham
"""

```

```

#Creating the matrix to solve the

```

```

n=5
matrix = [[-2/(12/n)**2-.02,1/(12/n)**2,0,0,0],
          [1/(12/n)**2,-2/(12/n)**2-.02,1/(12/n)**2,0,0],
          [0,1/(12/n)**2,-2/(12/n)**2-.02,1/(12/n)**2,0],
          [0,0,1/(12/n)**2,-2/(12/n)**2-.02,1/(12/n)**2],
          [0,0,0,1/(12/n)**2,-2/(12/n)**2-.02]]
b = [-.2-20/((12/n)**2),
     -.2,
     -.2,
     -.2,
     -.2-160/((12/n)**2)]

```

```

#Creating necessary vectors to use Gauss Elimination

```

```

o=None
Xn = [None]*n
theMax = [None]*n
ratio = [None]*n
k = i = j = 0

```

```

def partialPivoting(o,matrix,i):

```

```

    for row in matrix:
        theMax[o] = max(row, key=abs)
        o += 1
    for m in range(len(matrix)):
        ratio[m] = abs(matrix[m][m]/theMax[m])
    if i < n:
        if ratio[i-1] < max(ratio[i:], key=abs):
            swapIndex = max(ratio[i:])
            matrix[i-1] = matrix[swapIndex]

```

```

def forwardElim(k,i,j,matrix):

```

```

    for k in range(1,n):
        for i in range(k+1,n+1):

            #partial pivoting
            o=0
            partialPivoting(o,matrix,i)

            #forward elimination
            factor = matrix[i-1][k-1]/matrix[k-1][k-1]
            for j in range(k+1,n+1):
                matrix[i-1][j-1] = matrix[i-1][j-1] - factor * matrix[k-1][j-1]
            b[i-1] = b[i-1] - factor * b[k-1]

```

```

def backwardSub(Xn,b,matrix):

```

```

    Xn[n-1] = b[n-1]/matrix[n-1][n-1]
    for i in range(n-1,0,-1):
        Sum = b[i-1]

```

```

Xn[i-1] = Sum/matrix[i-1][i-1]
Xn[j-1]

def main(o,i,j,k,matrix,b,Xn):
    #calling each function
    forwardElim(k,i,j,matrix)
    backwardSub(Xn,b,matrix)

    #printing an answer
    print("OUTPUT")
    print("The finite difference approximation with 5 nodes is:" )
    w=1
    for element in Xn:
        print("T(",w*2,") = ",element,sep="")
        w += 1
    main(o,i,j,k,matrix,b,Xn)

```

OUTPUT

```

The finite difference approximation with 5 nodes is:
T(2) = 30.859586404008866
T(4) = 44.12219716175956
T(6) = 61.315685032544955
T(8) = 84.42073981907954
T(10) = 116.09906383277209

```

```

"""
Project 1 Part 2 (f)
This section of code finds a numerical approximation to the finite difference
approximation equation with a spacing of n intervals between boundary
conditions at x=0 and x=12.
@author: Jacob Needham
"""

#Getting user input for number of nodes
n = int(input("How many nodes would you like?\n"))

#Initializing finite difference approximation matrix of size n.
matrix = []
for i in range(n):                                #empty matrix
    matrix.append([0]*n)

for i in range(n-1):                              #the 1/(deltaX/n)^2 term (Ti-1 and Ti+1)
    matrix[i][i+1] = 1/((12/n)**2)
    matrix[i+1][i] = 1/((12/n)**2)
for i in range(n):                                #the current term - solution matrix term
    matrix[i][i] = -2/((12/n)**2)-.02              #(Ti -.2Ti)

b=[None]*n
for i in range(1,n-1):
    b[i] = -.2
b[0] = -.2-20/((12/n)**2)
b[n-1] = -.2-160/((12/n)**2)

#Creating the necessary matrices to use Gauss Elimination
o=None
Xn = [None]*n #solution space
theMax = [None]*n #lists to hold the maxes and ratios of each row
ratio = [None]*n
k = i = j = 0

def partialPivoting(o,matrix,i):
    for row in matrix:
        theMax[o] = max(row, key=abs)
        o += 1
    for m in range(len(matrix)):
        ratio[m] = abs(matrix[m][m]/theMax[m])
    if i < n:
        if ratio[i-1] < max(ratio[i:], key=abs):
            swapIndex = max(ratio[i:])
            matrix[i-1] = matrix[swapIndex]

def forwardElim(k,i,j,matrix):
    for k in range(1,n):
        for i in range(k+1,n+1):

            #partial pivoting
            o=0
            partialPivoting(o,matrix,i)

            #forward elimination
            factor = matrix[i-1][k-1]/matrix[k-1][k-1]
            for j in range(k+1,n+1):
                matrix[i-1][j-1] = matrix[i-1][j-1] - factor * matrix[k-1][j-1]

```

```

        b[i-1] = b[i-1] - factor * b[k-1]

def backwardSub(Xn,b,matrix):
    Xn[n-1] = b[n-1]/matrix[n-1][n-1]
    for i in range(n-1,0,-1):
        Sum = b[i-1]
        for j in range(i+1,n+1):
            Sum = Sum - matrix[i-1][j-1] * Xn[j-1]
        Xn[i-1] = Sum/matrix[i-1][i-1]

def main(o,i,j,k,matrix,b,Xn):
    #calling Gauss Elimination
    forwardElim(k,i,j,matrix)
    backwardSub(Xn,b,matrix)

    #printing each answer
    w=12/(n+1)
    print()
    print("The finite difference approximation with",n,"nodes is:" )
    for element in range(len(Xn)):
        print("T(",round(w*(element+1),2),") = ",Xn[element],sep="")
main(o,i,j,k,matrix,b,Xn)

```

OUTPUT

How many nodes would you like?

20

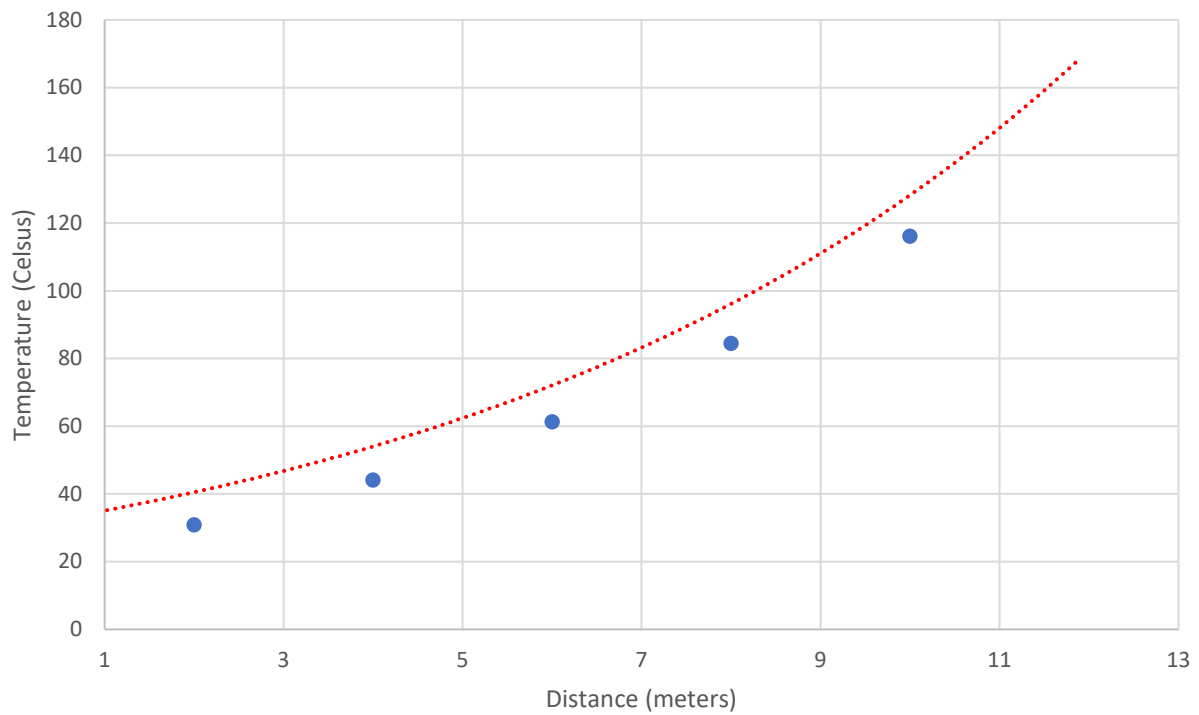
The finite difference approximation with 20 nodes is:

```

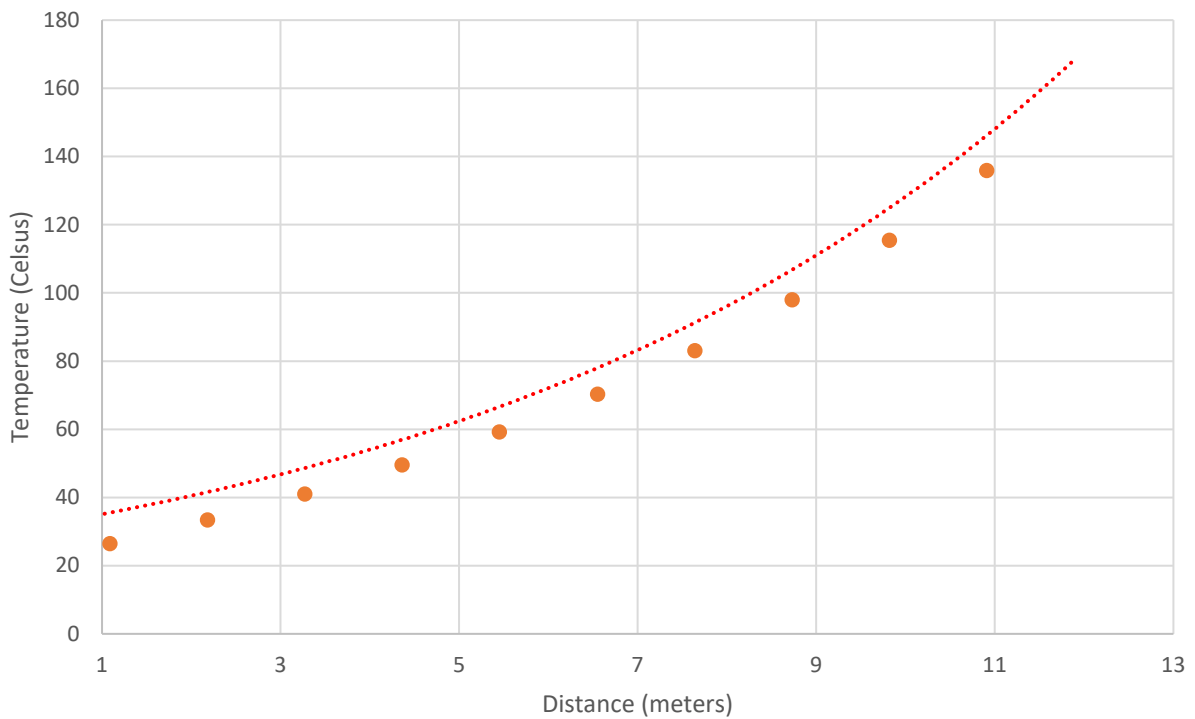
T(0.57) = 23.5531556330066
T(1.14) = 27.203893986570844
T(1.71) = 30.97850037683839
T(2.29) = 34.90415196981917
T(2.86) = 39.00911345698264
T(3.43) = 43.322940561036376
T(4.0) = 47.87669283712957
T(4.57) = 52.70315730165009
T(5.14) = 57.83708449874247
T(5.71) = 63.31543870422579
T(6.29) = 69.17766406837953
T(6.86) = 75.4659686138256
T(7.43) = 82.22562813329118
T(8.0) = 89.50531217531646
T(8.57) = 97.35743446500402
T(9.14) = 105.83853028283957
T(9.71) = 115.00966351871156
T(10.29) = 124.93686633191825
T(10.86) = 135.69161458271475
T(11.43) = 147.35134245850676

```


$T(x)$ vs 5 point Finite Difference Approximation



$T(x)$ vs 5 point Finite Difference Approximation



$T(x)$ vs 5 point Finite Difference Approximation

