Final Exam: MAE 3210 - Numerical Methods

April 23-27, 2020

• This exam is due in PDF format on Canvas by midnight on Monday April 27.

• You will be allowed to submit your answers hand-written on the pages provided or typed/hand-written on any pages of your choosing, provided you clearly indicate where the grader can find solutions to each problem.

• You may consult the course text, lecture notes, videos, and any of your own notes, but you are NOT allowed to collaborate with ANYONE to complete this exam. You must complete all solutions INDEPENDENTLY

|  |  |  |
| --- | --- | --- |
| Question | Value | Score |
| 1 | 5 |  |
| 2 | 5 |  |
| 3 | 10 |  |
| 4 | 10 |  |
| 5 | 10 |  |
| 6 | 10 |  |
| 7 | 10 |  |
| 8 | 10 |  |
| 9 | 15 |  |
| 10 | 15 |  |
| TOTAL | 100 |  |

1. (5 points) For each of the following problems, provide a concise (i.e. maximum single paragraph)

response. You do not need to include an example.

(a) What are benefits and drawbacks of the Newton-Raphson method in comparison to the bisection

method?

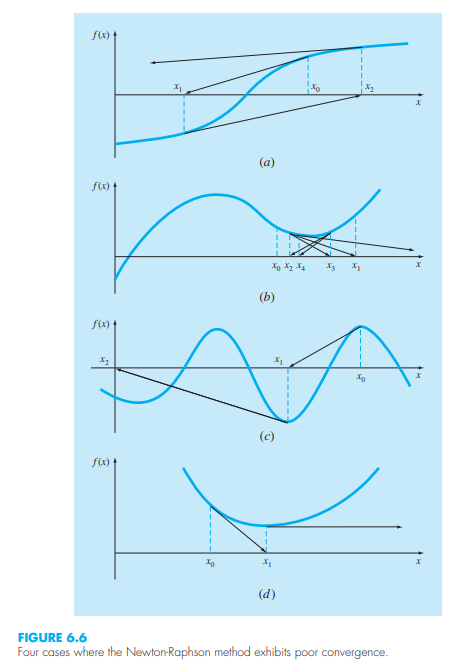
**Benefits**

Bisection method:

If the left point and the right point have opposite singes you will always find the root. There is no risk of division by zero

Newton-Raphson method**:**

The Newton-Raphson method only takes 1 input point. The Newton-Raphson method is very efficient and fast.



**Drawbacks**

Bisection method:

The Bisection method takes 2 point (left and right points). It’s a brute force method is not very efficient and costly.

Newton-Raphson method:

The Newton-Raphson can diverge in in some situations. There also can be problems when the function has more than one root. There is a risk of division by zero. Also it required the calculation of the derivative, in some situations that can difficult or unknown. Its convergence depends the how the of the initial guess was, there are some functions that will never has a good guess to diverge. Figure 6.6 show 4 functions and guesses that fail to converge.

(b) What mathematical conditions on a function f(x) guarantee that the fixed point method will converge

in search of a root xr satisfying f(xr) = 0?

The fixed point method will converge

!@#$%^&\*(!@#$%^&\*()

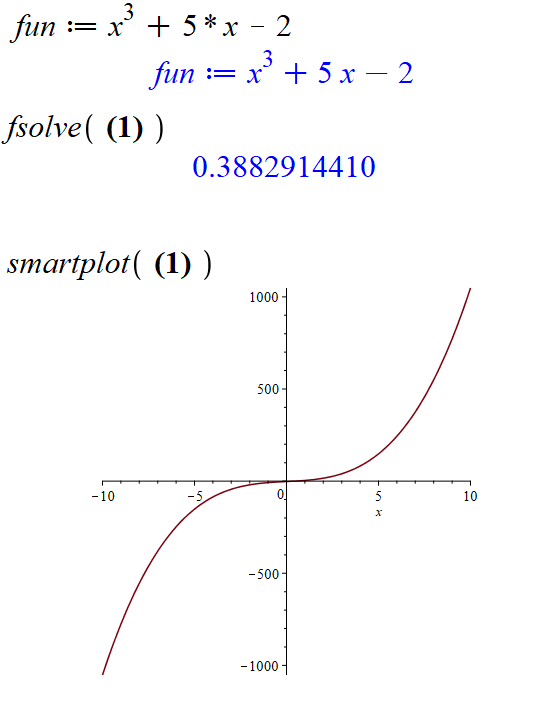
(c) In this course we discussed three distinct methods for solving linear algebraic systems. What are the three methods, and why did we not focus only on Gauss elimination? I.e. in what contexts are the other methods useful?

The three distinct methods:

1. Gauss Elimination
2. Naive Gauss Elimination
3. LU Decomposition
4. Gauss Jorden Elimination

we did not focus only on Gauss elimination for these reasons:

* When Gauss elimination takes place division by small numbers could make approximations less accurate by round off errors. This will result in a system of equations called “ill-condition system” due to the small number division.
* have the possibly of division by zero. (thus, the need for partial pivoting is needed)
* For very large calculations saving the inverted matrices also the computation to be more efficient (thus the use of LU decomposition)
* what contexts are the other methods useful

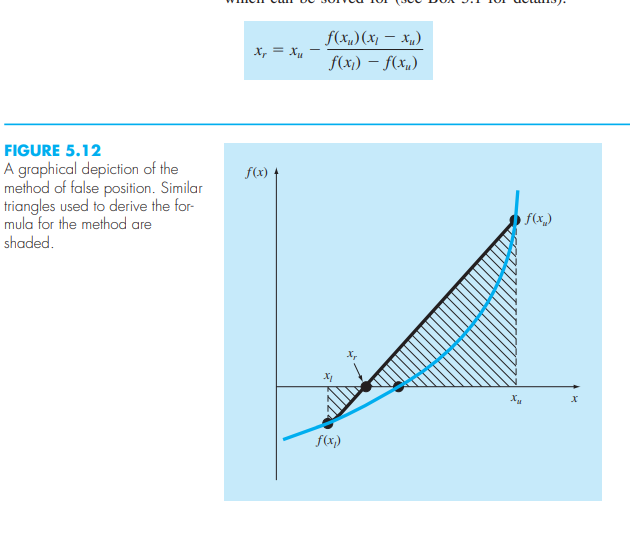
2. (5 points) Consider f(x) = x 3 + 5x − 2.

Starting with xl = 0 and xu = 1, evaluate the first two iterations of the false position method to approximate the

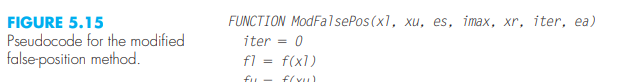
root xr satisfying f(xr) = 0 with 0 ≤ xr ≤ 1. Show your work

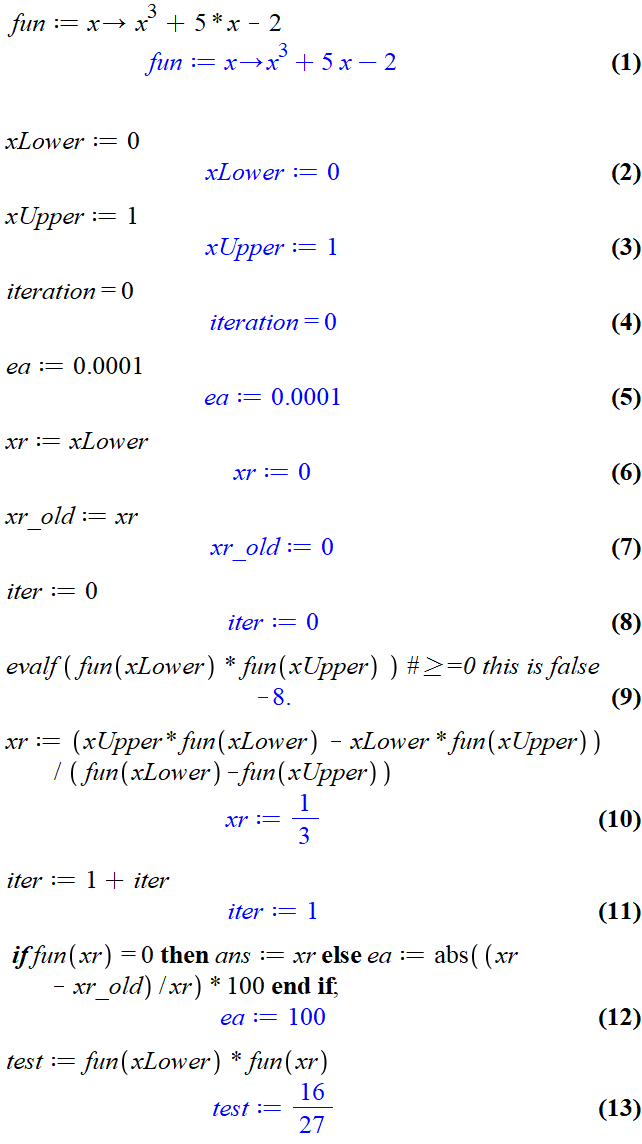
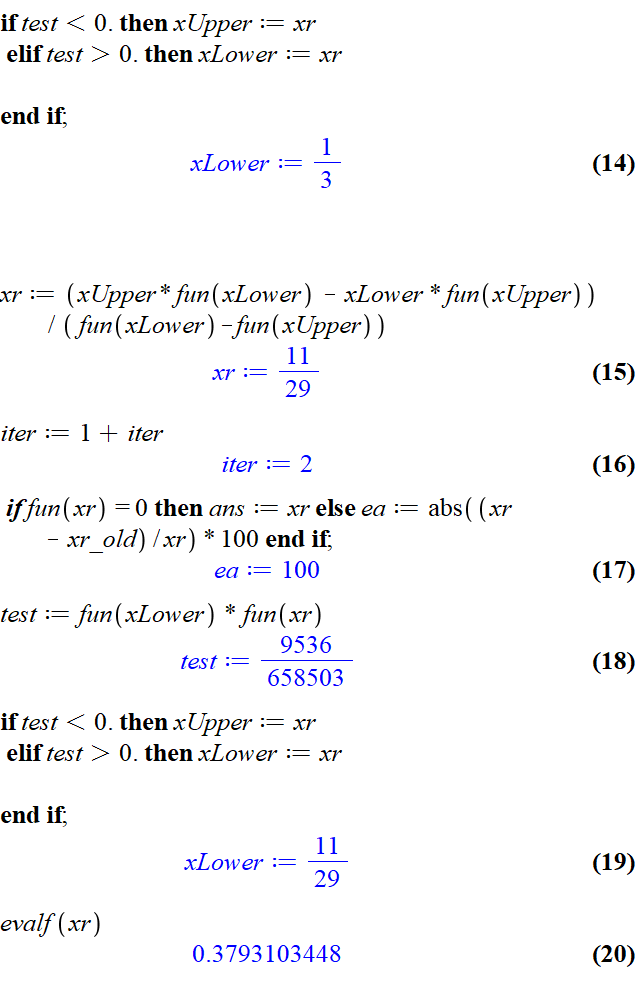
Graph to see what’s going on…see plot to the right

NOTES from the book:



based off of the Figure 5.15



After the 2nd Iteration the value for Xr was found to be 0.3793103448

3. (10 points) Write pseudocode (that is, a programming structure understandable from English words and mathematics alone) for an algorithm which applies the bisection method to find the root of a given function f(x). Design the pseudocode to take as input:

* Initial guesses xl and xu which bracket the location xr of the root.
* A desired absolute approximate error Ea,d. Your pseudocode should begin by determining how many iterations n are needed in order to achieve this approximate error, and your bisection method should then iterate this many times.

Comments are encouraged but not required for full points

def Bisect(function,xLower, xUpper, Ead, n):

    # Bisection method to finds a root of a funtion.

    # xLower: lower bound guess.

    # xUpper: upper bound guess.

    # ead: error threshold.

    # n: max iterations threshold.

    iter =0

    xr = xLower

    ea = Ead

    xr\_old = xr

    if (function(xLower) \* function(xUpper) >= 0):

        if function(xLower) == 0:

            return (xLower)

        elif function(xUpper) == 0:

            return (xUpper)

        print("Your Vales xLower:"+str(xLower)+" and xUpper:"+str(xLower)+"\n")

        return

    while (ea >= Ead) and (iter < n):

        xr\_old = xr\_old

        xr = (xLower + xUpper)/2 # bisection method

        print("Iteration:",iter," xr: ",xr)

        iter +=1

        if xr != 0 :

            ea = abs((xr- xr\_old)/xr)\*100

        else:

            print("Error")

            # maybe Exit

            return None

        test = function(xLower) \* function(xr)

        if test <0:

            # id the value is negative then the root is to the left

            xUpper = xr

        elif  test >0:

            # the vale is positive and the root is to the right

            xLower = xr

        else:

            ea = 0

    x = xr

    print("x: "+str(x)+ " is the root approx Bisection method")

    return x

4. (10 points) Consider the optimization problem:

Maximize f(x, y)

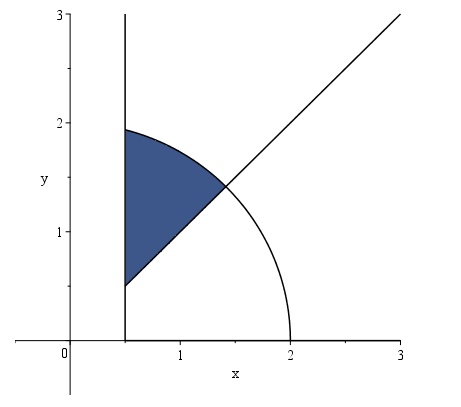
subject to the constraints

x2 + y2 ≤ 4,

x − y ≤ 0,

x ≥ 0.5,

y ≥ 0.

Write pseudocode (that is, a programming structure understandable from English words and mathematics alone) for an algorithm which applies the **random search method** to solve this optimization problem. Design the pseudocode to take the number of iterations n of the random search as input, and write your pseudocode assuming that you can call a function “Rand”, which outputs a random number selected uniformly from the interval [0, 1]. Comments are encouraged but not required for full points.

Using this equation form the book on page 371



import matplotlib.pyplot as plt

import numpy as np

import random

def randomSearch(n,function,isConstr):

    x=0

    y=0

    # the random point must be in the constraints

    while not isConstr(x,y):

        x = 10\*random.random()

        y = 10\*random.random()

    iter =0

    lastMax =0 #-sys.maxsize

    difrence = 0

    xList =[x]

    yList =[y]

    # keep going tell it reaches n

    while iter <=n:

        xTest =x + difrence\*random.random()

        yTest =y + difrence\*random.random()

        curVal = function(xTest,yTest)

        # print("Test: ",isConstr(xTest,yTest),"(",xTest,",",yTest,")")

        # if the new value is bigger then the lastMax

        # and check if it works in the Solution Space

        if lastMax <curVal and isConstr(xTest,yTest):

            # print("Inter: ",iter," lastMax: ",lastMax,"> curVal: ",curVal )

            difrence =abs( lastMax - curVal)

            # save this point

            x =xTest

            y =yTest

            # update the (x,y)

            yList.append(y)

            xList.append(x)

            lastMax = curVal

        iter +=1

    print("The optimil Value: (",x,",",y,")")

    # print("The Value is Constrained: ", isConstr(x,y))

    # plt.plot(xList,yList,color='red', label="randWalk")

    # plt.show()

    return x,y,xList,yList

def isConstrFun4(x,y):

    # x2 + y2 ≤ 4,

    # x − y ≤ 0,

    # x ≥ 0.5,

    # y ≥ 0.

    if x\*\*2 + y\*\*2 <= 4 and x - y <= 0 and x >= 0.5 and y >= 0:

        return True

    else:

        return False

def fun(x,y):

    return 3\*x+y

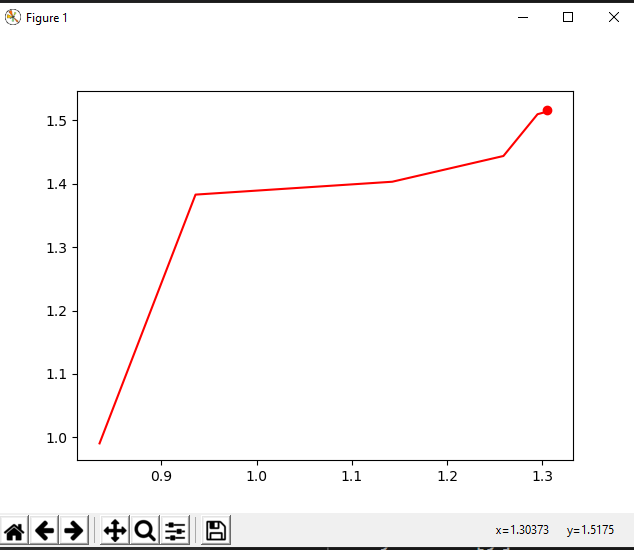
# testing

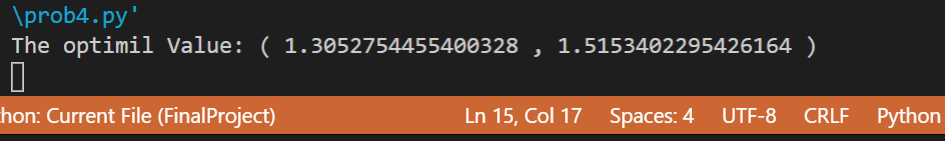
xi,yi,xLtempi,yLtempi = randomSearch(1000000,fun,isConstrFun4)

plt.plot(xLtempi,yLtempi,color="red", label="randWalk")

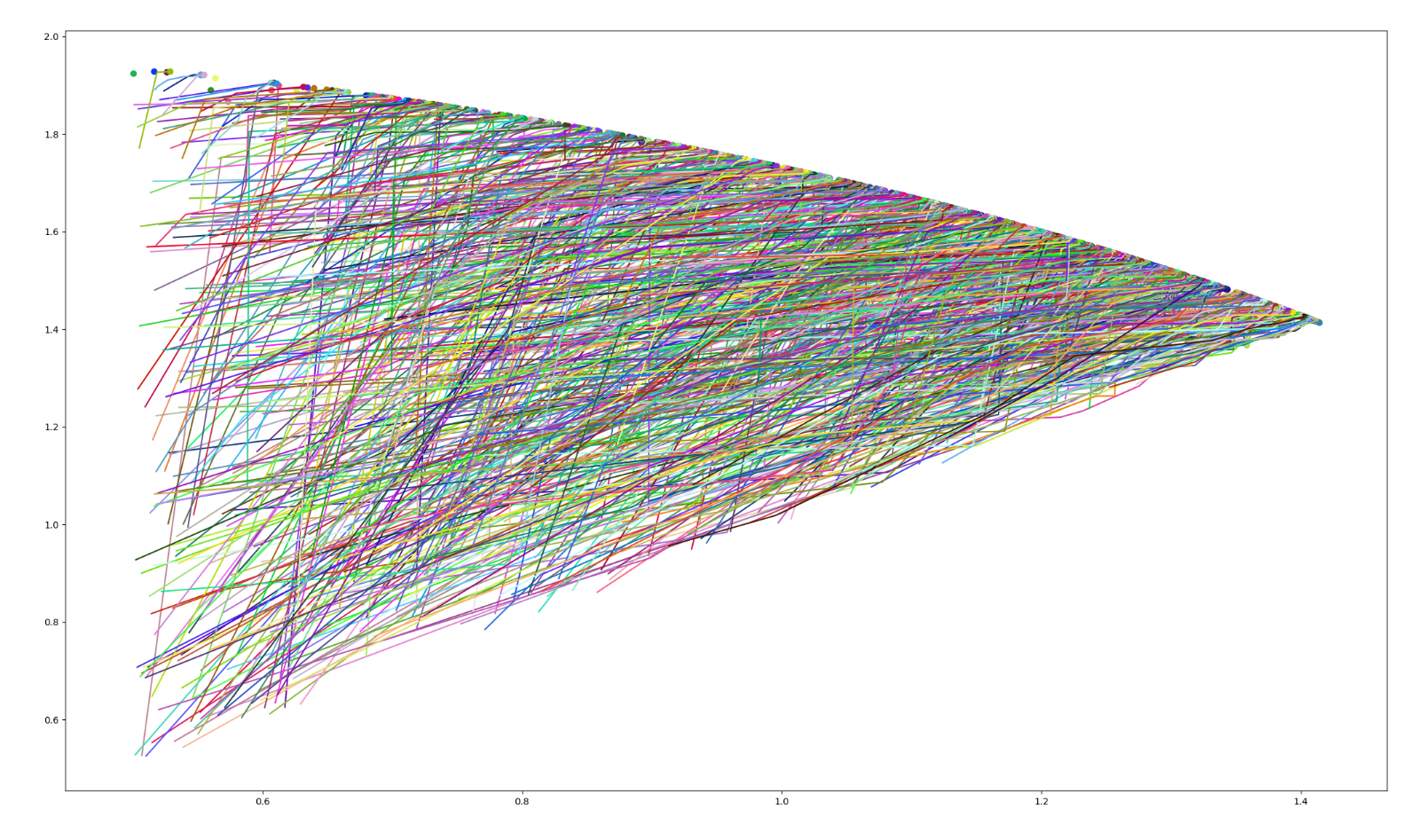
plt.scatter(xi,yi,color="red", label="randWalk")

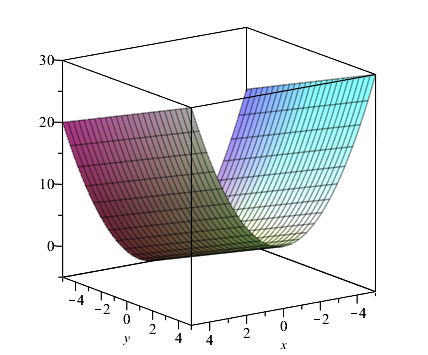
plt.show()





Ran 1000 times:





5. (10 points) Consider the optimization problem:

Maximize f(x, y) = 3x + y

subject to the constraints

x2 + y ≤ 4,

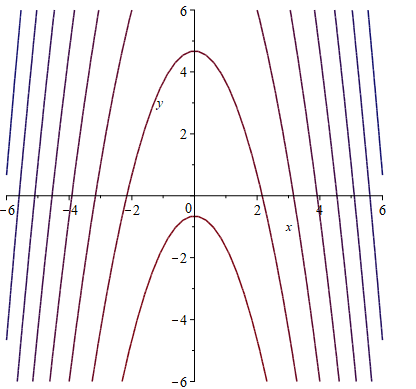
−3x + y ≤ 0,

x ≥ 0.5,

y ≥ 0.

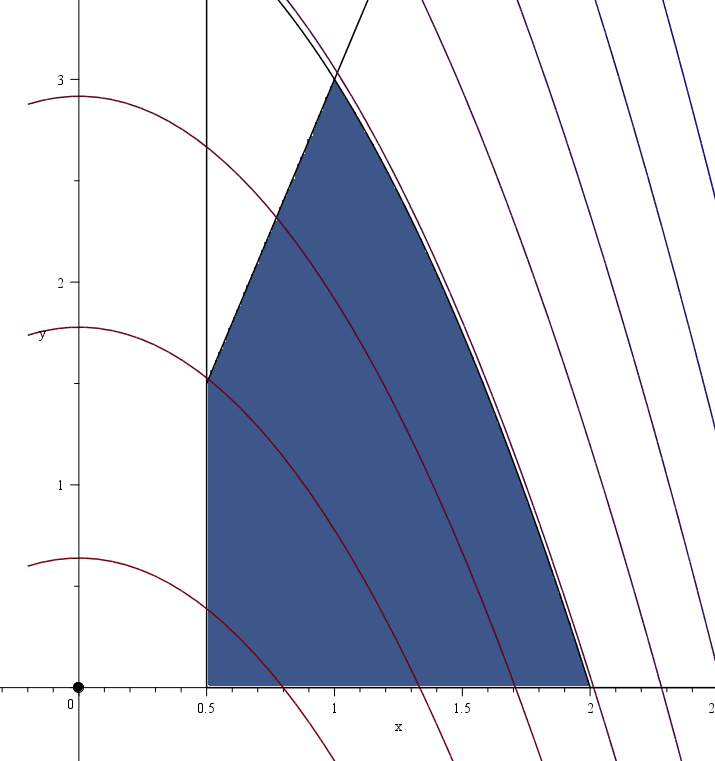
3D representation of the objective function

f(x, y) = 3x + y



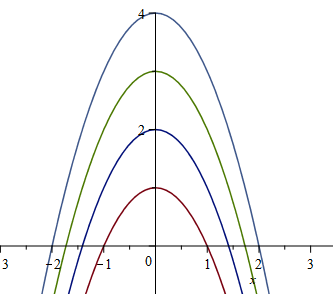
This is a contour plot of the objective function f(x, y) = 3x + y

1. Plot the feasible solution space in the x − y plane.



feasible solution space in the x − y plane

(b) Solve the optimization problem outlined above (i.e. in problem 5) by using the graphical method. Include plots and justify your answer

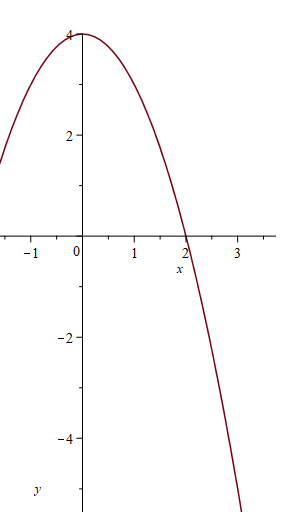
 x2+y =4

x2+y =3

x2+y =2

x2+y =1

Plot if the different z functions



When analyzing the largest output from the objective function it shows that there are infant many combinations of (x,y) that will max out z at z =4

**The all the solutions exist on the curve defined by x2 + y =4**

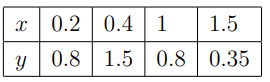
This is the case because one of the constraints was defined t exactly the same as the objective function.

Plot of the function x2 + y =4

6. (10 points) By applying an appropriate transformation, find a way to use a least squares fit to solve for the constants α and β in the nonlinear regression

y = α x eβx

applied to the following data:



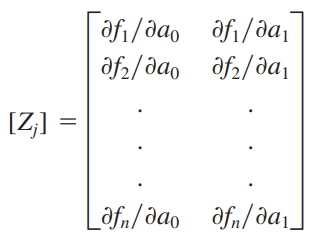
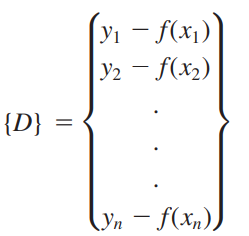
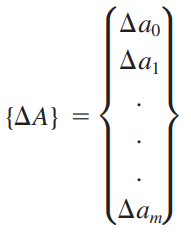
That is, produce an explicit formula for the regression coefficients α and β in terms of matrices with numerical entries. However, you do NOT need to multiply out any of the matrices or find any matrix inverses in order to produce a detailed solution..

Using the Gauss-Newton method:

From the book on page 483: “The Gauss-Newton method is one algorithm for minimizing the sum of the squares of the residuals between data and nonlinear equations. The key concept underlying the technique is that a Taylor series expansion is used to express the original nonlinear equation in an approximate, linear form. Then, least-squares theory can be used to obtain new estimates of the parameters that move in the direction of minimizing the residual”

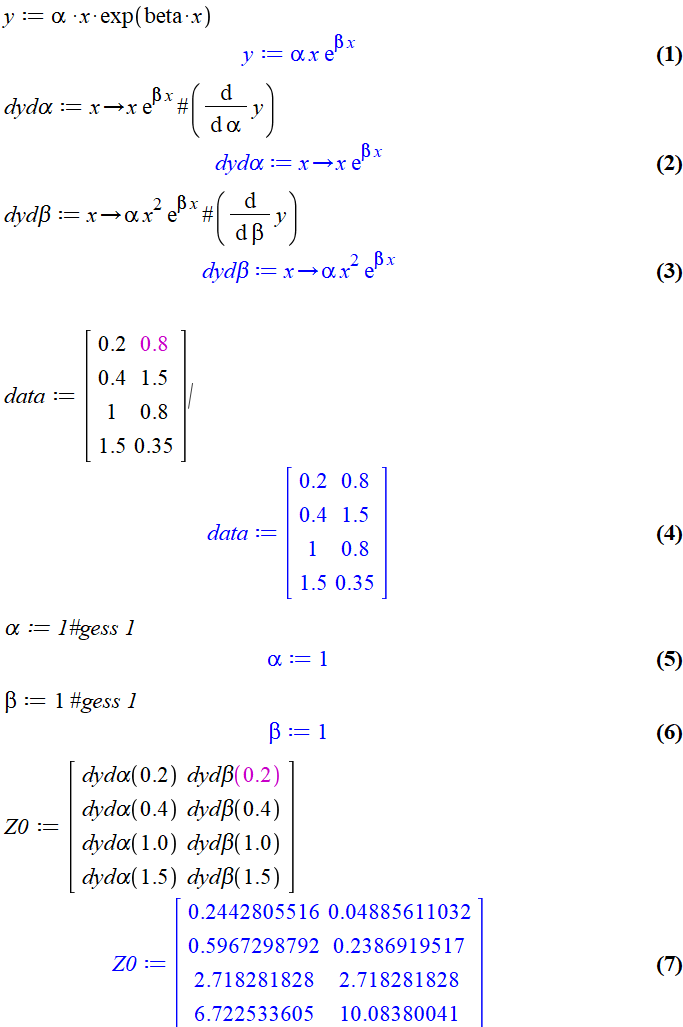
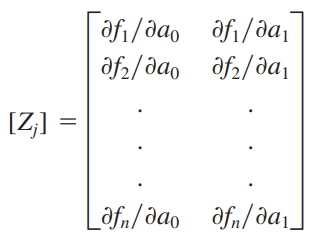
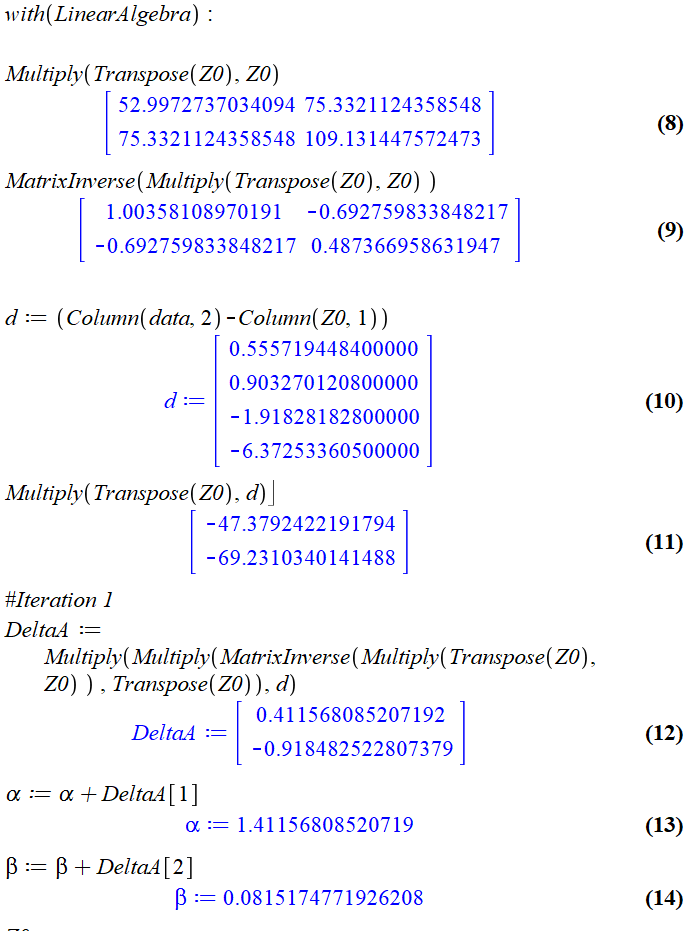
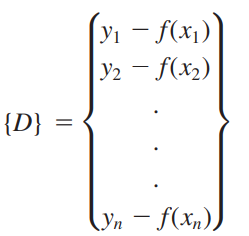
To do this we need to solve equation: 17.35

Find:

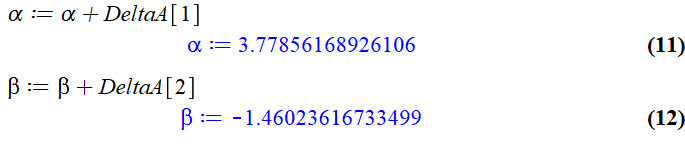
1. 
2. 
3. 
4. Solve for Delta A



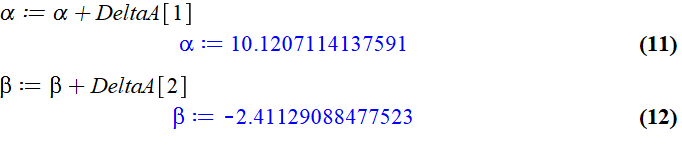
Prob 6 continues to the next page 🡪

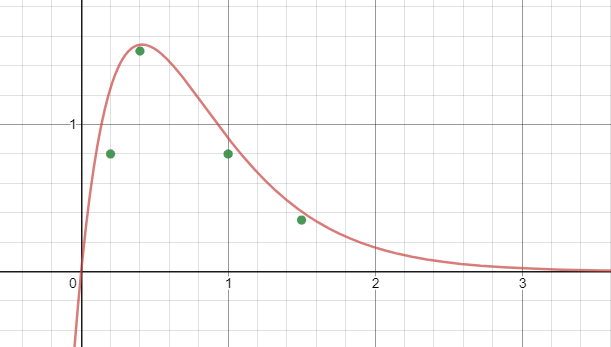
Iteration 2:



Iteration 3



Any more additional iteration were not as close

**alpha := 10.1207114137591 y = α x eβx**

**beta := -2.41129088477523**

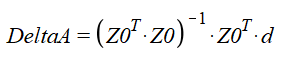
y = α x eβx

explicit formula

Plot of the resulting fitted function with data points

**y = 10.1207114137591 \* x \* e-2.41129088477523\*x**

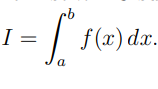
General solution would look something like this







7. (10 points) Write pseudocode (that is, a programming structure understandable from English words and mathematics alone) for an algorithm which, for a given function f(x) and interval bounds a and b with a < b, and a prescribed odd number of subintervals n ≥ 3, applies the multiple-application Simpson’s 1/3 rule on the first n − 3 subintervals, and the Simpson’s 3/8 rule on the last 3 subintervals, in order to approximate



3 requirements

1. a < b
2. prescribed odd number of subintervals n ≥ 3
3. multiple-application Simpson’s 1/3 rule on the first n − 3 subintervals
4. Simpson’s 3/8 rule on the last 3 subintervals
5. def testBound(a,b):
6. if a > b:
7. temp = a
8. a = b
9. b = temp
10. return
11. elif a == b:
12. print( "Error : the Lower bound is the smae as the upper bound")
13. exit(-1)
14. def simpsons\_1\_3(a,b,n,funct):
15. if n== 0:
16. return 0
17. testBound(a,b)
18. step = ( b - a) /n
19. sumVal = 0
20. xList = np.arange(a + step, b,step)
21. for i in range(len(xList)):
23. if i == 0 or i % 2 == 0:
24. sumVal += 4 \* funct(xList[i])
25. else:
26. sumVal += 2 \* funct(xList[i])
27. return ( (b - a) \* (funct(a) + sumVal + funct( b )) / (3\*n))
28. def simpsons\_3\_8(a,b,n,funct):
29. if n== 0:
30. return 0
31. testBound(a,b)
32. # Interval Size  = step
33. step = (( b -  a) / n)
34. sumVal = funct( a) + funct( b)
35. for i in range(1, n ):
36. if (i % 3 == 0):
37. sumVal = sumVal + 2 \* funct( a + i \* step)
38. else:
39. sumVal = sumVal + 3 \* funct( a + i \* step)
41. return (( 3 \* step) / 8 ) \* sumVal
42. def prob2Algorithm(a,b,n,funct):
43. # 1.a < b
44. testBound(a,b)
45. if (n%2 == 0 or n < 3):
46. print("as stated in the Problem statement, only prescribed odd (n≥ 3) are allowed")
47. print("You picked n = ",n)
48. return None
50. elif (n >= 3): # 2. prescribed odd number of subintervals n ≥ 3
51. sumTot = 0
52. step = (b-a)/n
53. # 3.multiple-application Simpson’s 1/3 rule on the first n − 3 subintervals
54. sumTot += simpsons\_1\_3(a,b-3\*step,n-3,funct)
56. # 4.Simpson’s 3/8 rule on the last 3 subintervals
57. sumTot += simpsons\_3\_8(b-3\*step, b, 3,funct)
58. return sumTot

8. (10 points) Suppose the temperature distribution T for a rod with a heat source is given by the 2nd order ODE



where 0 ≤ x ≤ 10. We impose the boundary condition T(10) = 150, and assume that the rod is insulated

at x = 0.

Use a finite difference scheme with spacing ∆x = 2.5 to express the numerical solution T to (1) with the

given boundary conditions in terms of a matrix inverse which you do NOT need to compute.

🡨 ------------------0 ≤ x ≤ 10------------------🡪

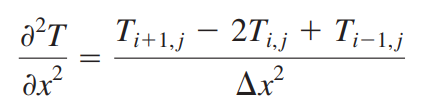
|🡨 ∆x = 2.5 🡪|

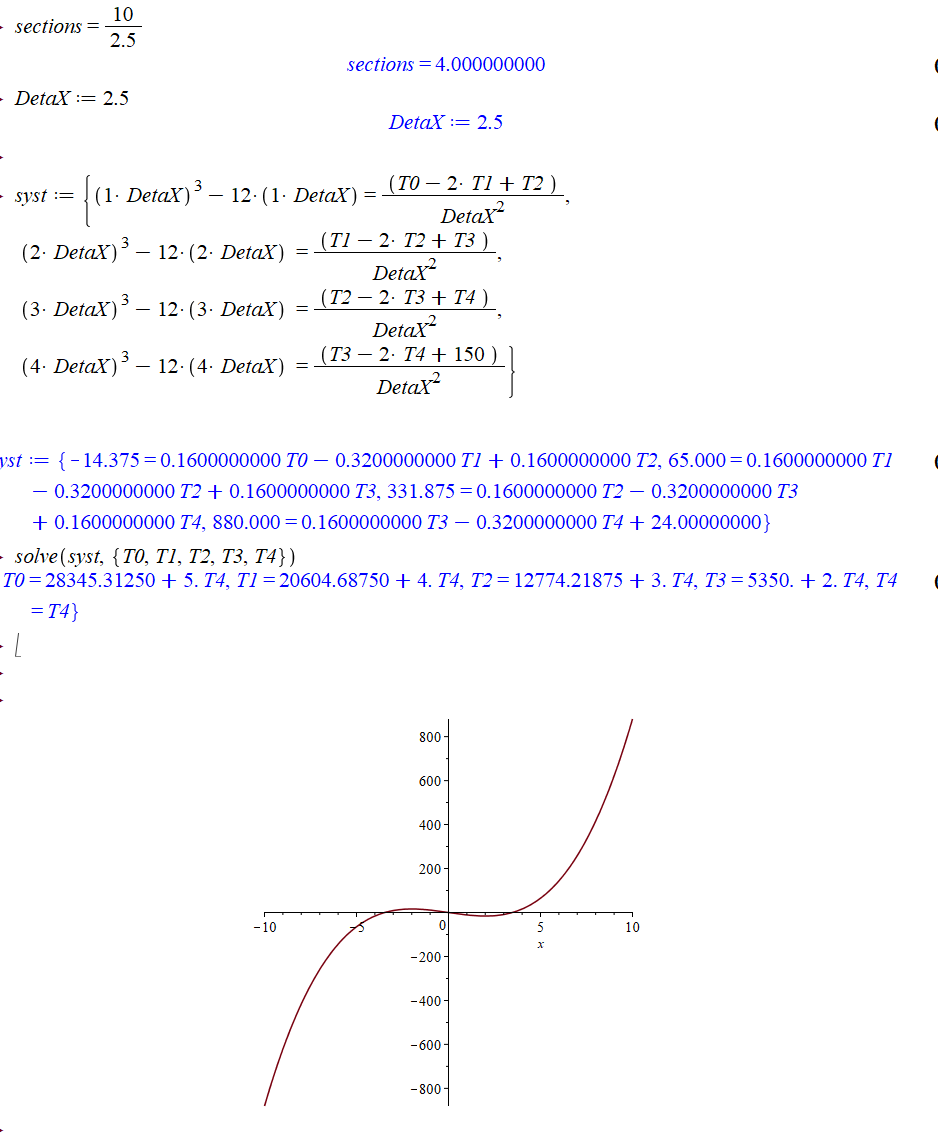
🡨 T(10)=150

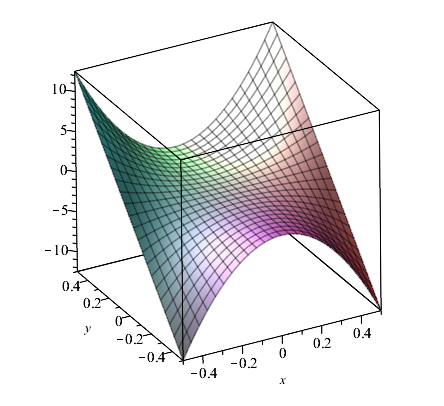
🡸T(0) = 0

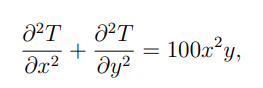
.

From page 855 in the book





9. (15 points) Consider the Poisson equation for a heated plate in the domain [0, 1]×[0, 1] (with internal heating),



with insulated boundary conditions at x = 0, insulated boundary conditions at y = 0, a prescribed temperature of T = 100oC at the boundary where x = 1, and a prescribed temperature of T = 100oC at the boundary where y = 1.

Heat Flux

* insulated boundary conditions at x = 0,
* insulated boundary conditions at y = 0,
* T = 100oC at the boundary where x = 1,
* T = 100oC at the boundary where y = 1.

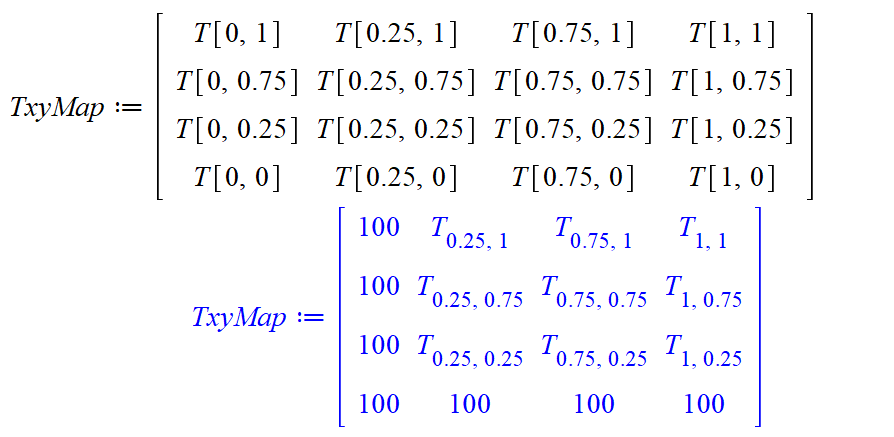
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | 100oC | T0.25,1 | T0.75,1 | T1,1 | | T0,0.75 | T0.25,0.75 | T0.75,0.75 | T1,0.75 | | T0,0.25 | T0.25,0.25 | T0.75,0.25 | T1,0.25 | | T0,0 | T0.25,0 | T0.75,0 | 100oC | |

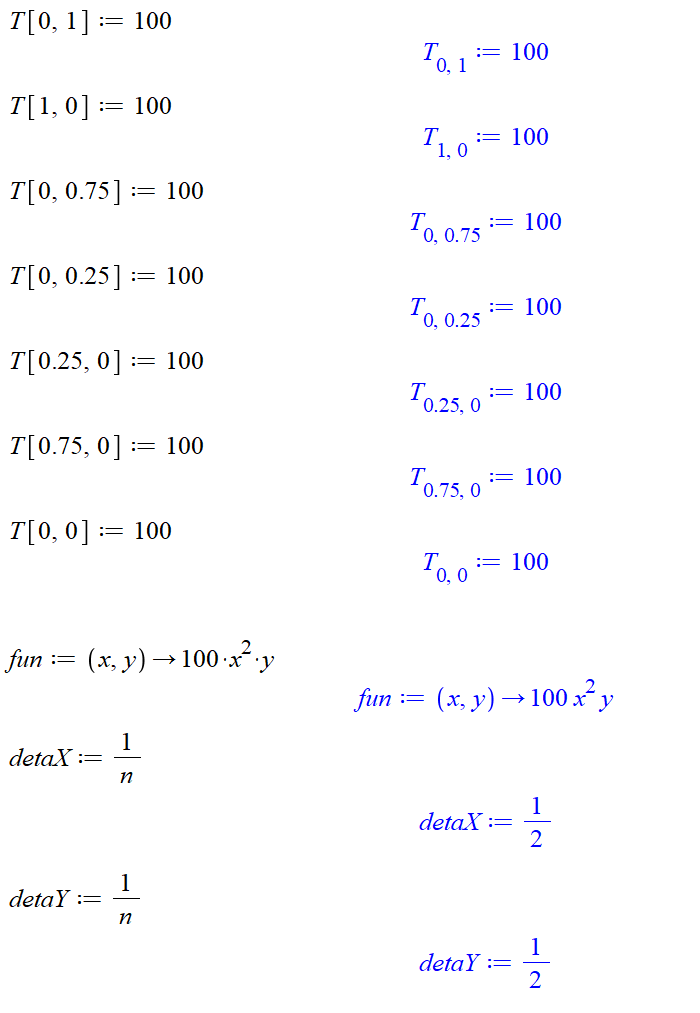
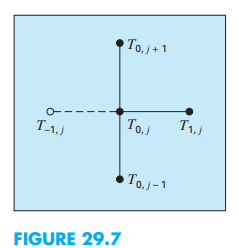
(a) Using n = 2 subintervals in each of the x and y directions, set up a system of equations for the

temperature Ti,j , at any nodes where the temperature is not specified in the problem

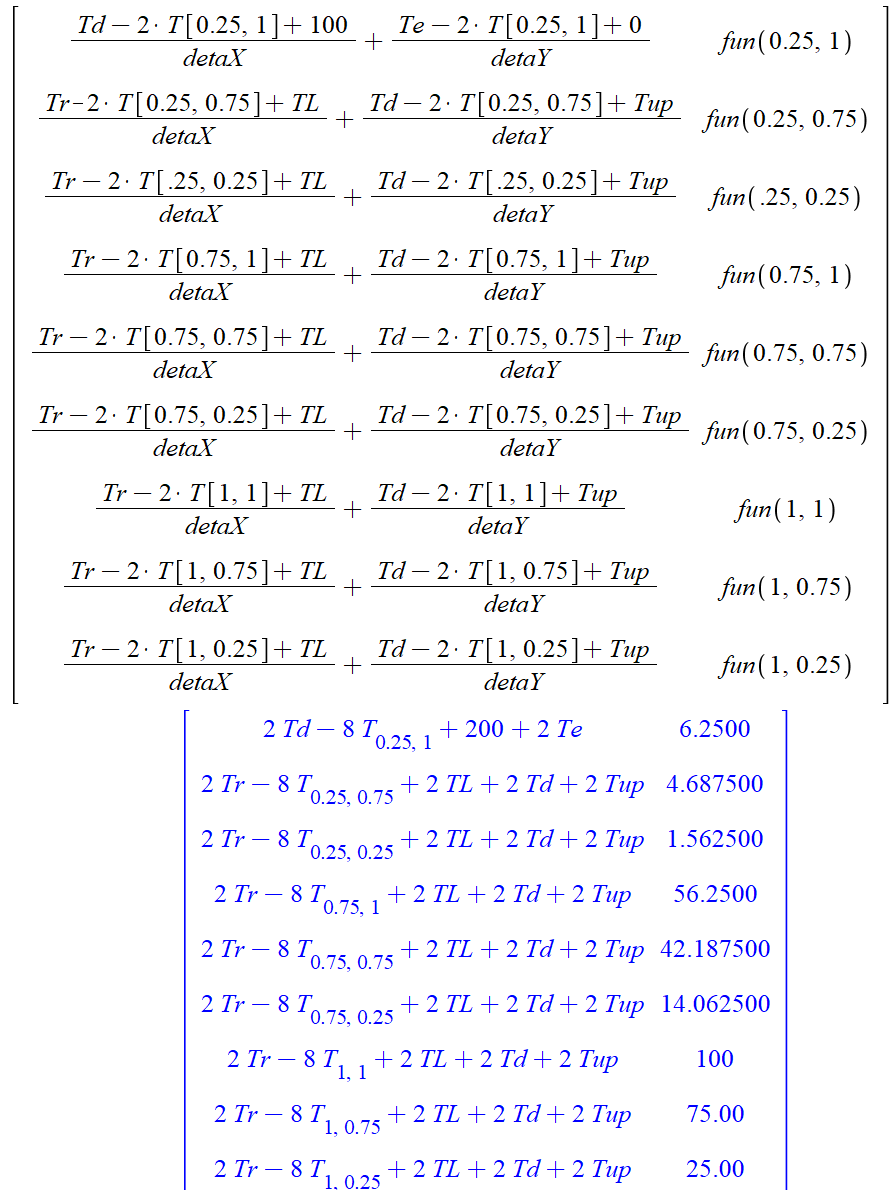
to relate the flux to a location we can use this





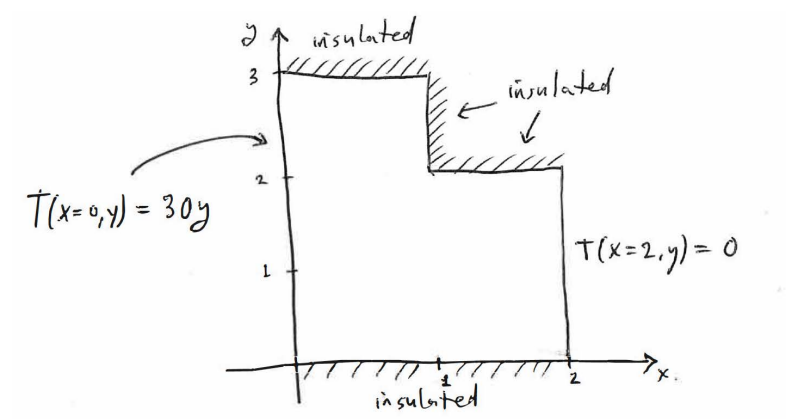
 

Each TL,Tr, Td,Tup, and Td is dependent on the location of the current T, if we could use python or something I would have the computer updated those vales based on Figure 29.7



(b) Starting with an initial guess that all unknown values are zero, apply two iterations of Liebmann’s method to compute the unknown temperatures. Use a relaxation of factor of λ = 0.8 in your iteration.

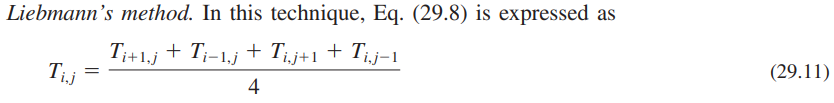
10. (15 points) Consider the “L-shaped” region in the domain depicted below such that the temperature T = T(x, y) satisfies Laplace’s equation with boundary conditions as drawn in the plot.

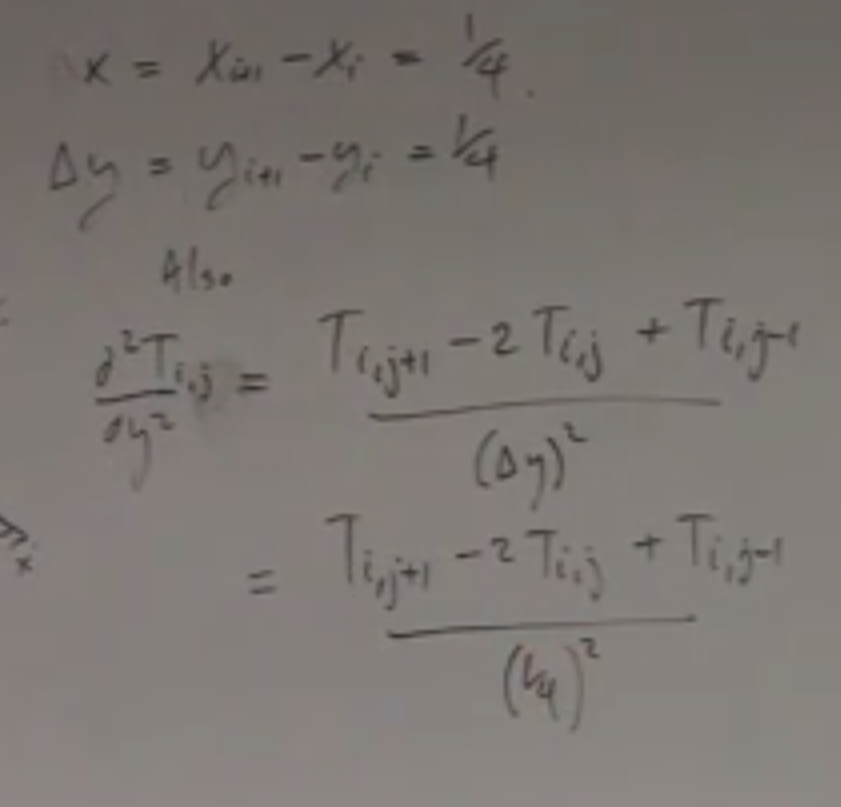


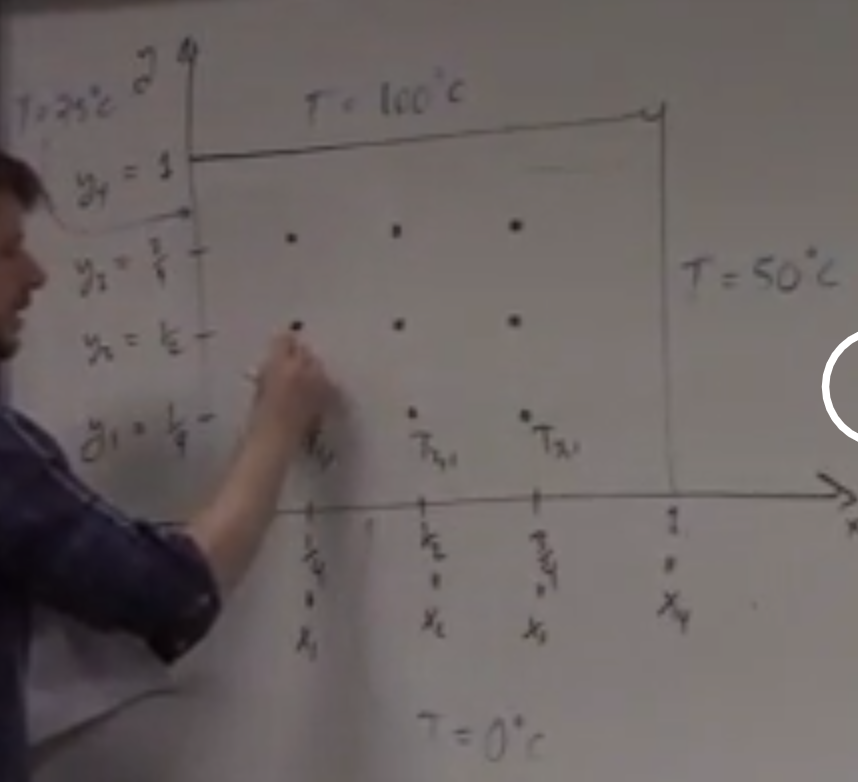
1. Using **n = 2** subintervals in the x direction and **m = 3** subintervals in the y direction, set up a system of equations for the temperature Ti,j , at any nodes where the temperature is not specified in the problem.

**n = 2**

**m = 3**

****





1. Starting with an initial guess that all unknown values are zero, apply two iterations of Liebmann’s method to compute the unknown temperatures. Use a relaxation of factor of λ = 1.2 in your iteration.