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.

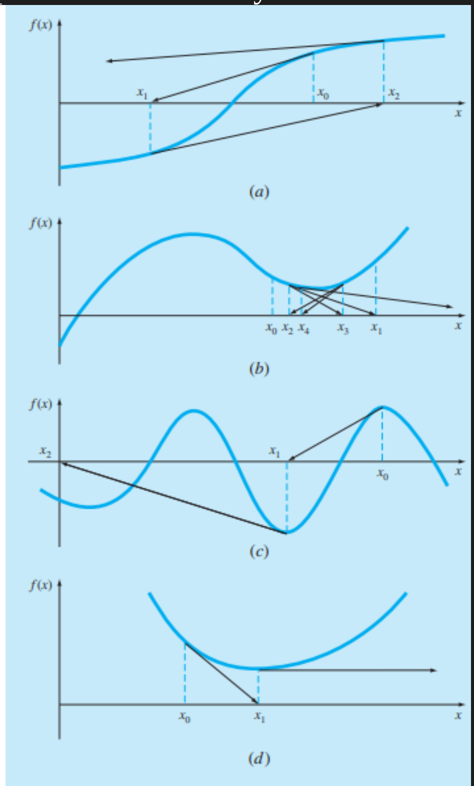


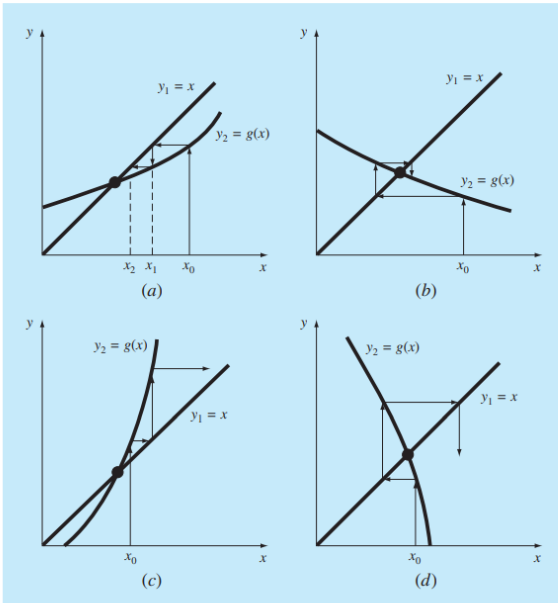
FIGURE 6.6
Four cases where the Newton-Raphson method exhibits poor convergence.

(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 when |f ’(x)| <=1 and x (initial guess) exists on the interval from [a,b]

Fig 6.3 show converges and divergent examples

FIGURE 6.3
Iteration cobwebs depicting convergence (*a* and *b*) and divergence (*c* and *d*) of simple fixed-point iteration. Graphs (*a*) and (*c*) are called monotone patterns, whereas (*b*) and (*d*) are called oscillating or spiral patterns. Note that convergence occurs when $|g'(x)| < 1$.



for Fig. 6.3*c* and *d*, where the iterations diverge from the root. Notice that convergence seems to occur only when the absolute value of the slope of $y_2 = g(x)$ is less than the slope of $y_1 = x$, that is, when $|g'(x)| < 1$. Box 6.1 provides a theoretical derivation of this result.

(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

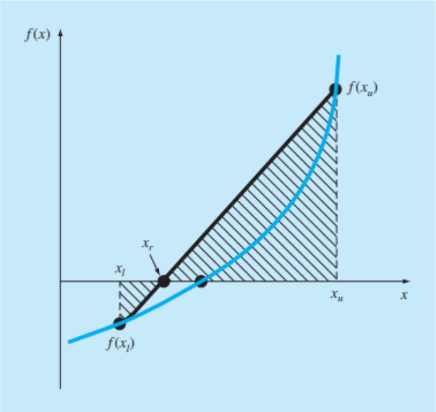
2. (5 points) Consider $f(x) = x^3 + 5x - 2$.
Starting with $x_l = 0$ and $x_u = 1$, evaluate the first two iterations of the false position method to approximate the root x_r satisfying $f(x_r) = 0$ with $0 \le x_r \le 1$. Show your work

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

NOTES from the book:

$$x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}$$

FIGURE 5.12
A graphical depiction of the method of false position. Similar triangles used to derive the formula for the method are shaded.



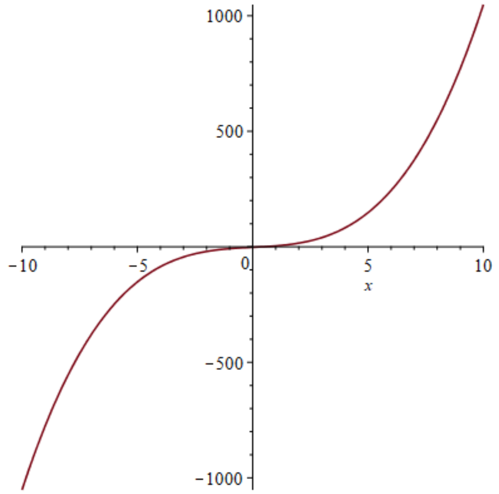
$fun := x^3 + 5*x - 2$

$$fun := x^3 + 5x - 2$$

$fsolve(\ (1)\ )$

$$0.3882914410$$

$smartplot(\ (1)\ )$



based off of the Figure 5.15

FIGURE 5.15
Pseudocode for the modified false-position method.

```
FUNCTION ModFalsePos(xl, xu, es, imax, xr, iter, ea)
  iter = 0
  fl = f(xl)
  fu = f(xu)
```

$fun := x \rightarrow x^3 + 5*x - 2$

$$fun := x \rightarrow x^3 + 5\,x - 2 \tag{1}$$

$xLower := 0$

$$xLower := 0 \tag{2}$$

$xUpper := 1$

$$xUpper := 1 \tag{3}$$

$iteration = 0$

$$iteration = 0 \tag{4}$$

$ea := 0.0001$

$$ea := 0.0001 \tag{5}$$

$xr := xLower$

$$xr := 0 \tag{6}$$

$xr\_old := xr$

$$xr\_old := 0 \tag{7}$$

$iter := 0$

$$iter := 0 \tag{8}$$

$evalf(fun(xLower) * fun(xUpper))$ # ≥=0 this is false

$$-8. \tag{9}$$

$xr := (xUpper * fun(xLower) - xLower * fun(xUpper)) / (fun(xLower) - fun(xUpper))$

$$xr := \frac{1}{3} \tag{10}$$

$iter := 1 + iter$

$$iter := 1 \tag{11}$$

if $fun(xr) = 0$ **then** $ans := xr$ **else** $ea := \text{abs}((xr - xr\_old) / xr) * 100$ **end if**;

$$ea := 100 \tag{12}$$

$test := fun(xLower) * fun(xr)$

$$test := \frac{16}{27} \tag{13}$$

if $test < 0.$ **then** $xUpper := xr$
elif $test > 0.$ **then** $xLower := xr$

end if;

$$xLower := \frac{1}{3} \tag{14}$$

$xr := (xUpper * fun(xLower) - xLower * fun(xUpper)) / (fun(xLower) - fun(xUpper))$

$$xr := \frac{11}{29} \tag{15}$$

$iter := 1 + iter$

$$iter := 2 \tag{16}$$

if $fun(xr) = 0$ **then** $ans := xr$ **else** $ea := \text{abs}((xr - xr\_old) / xr) * 100$ **end if**;

$$ea := 100 \tag{17}$$

$test := fun(xLower) * fun(xr)$

$$test := \frac{9536}{658503} \tag{18}$$

if $test < 0.$ **then** $xUpper := xr$
elif $test > 0.$ **then** $xLower := xr$

end if;

$$xLower := \frac{11}{29} \tag{19}$$

$evalf(xr)$

$$0.3793103448 \tag{20}$$

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

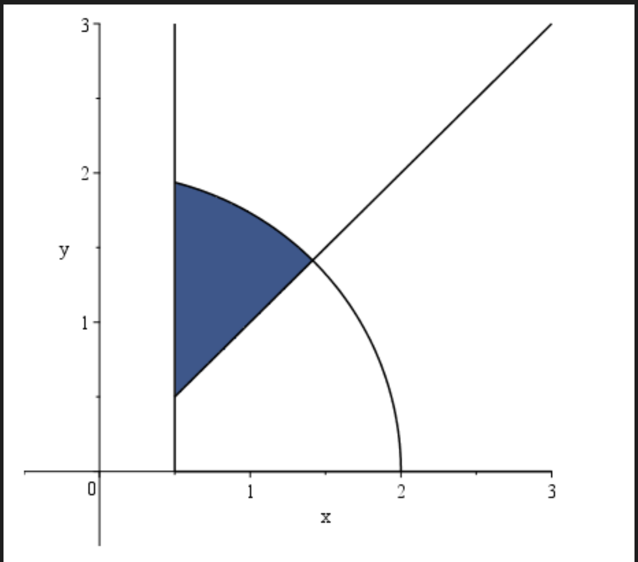
```
def Bisect(function,xLower, xUpper, Ead, n):
    # Bisection method to finds a root of a funtion.
    # xLower: lower bound guess.
    # xUpper: upper bound guess.
    # ead: error threshold.
    # n: max iterations threshold.
    iter =0
    xr = xLower
    ea = Ead
    xr_old = xr

    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
$x^2 + y^2 \le 4$,
$x - y \le 0$,
$x \ge 0.5$,
$y \ge 0$.

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

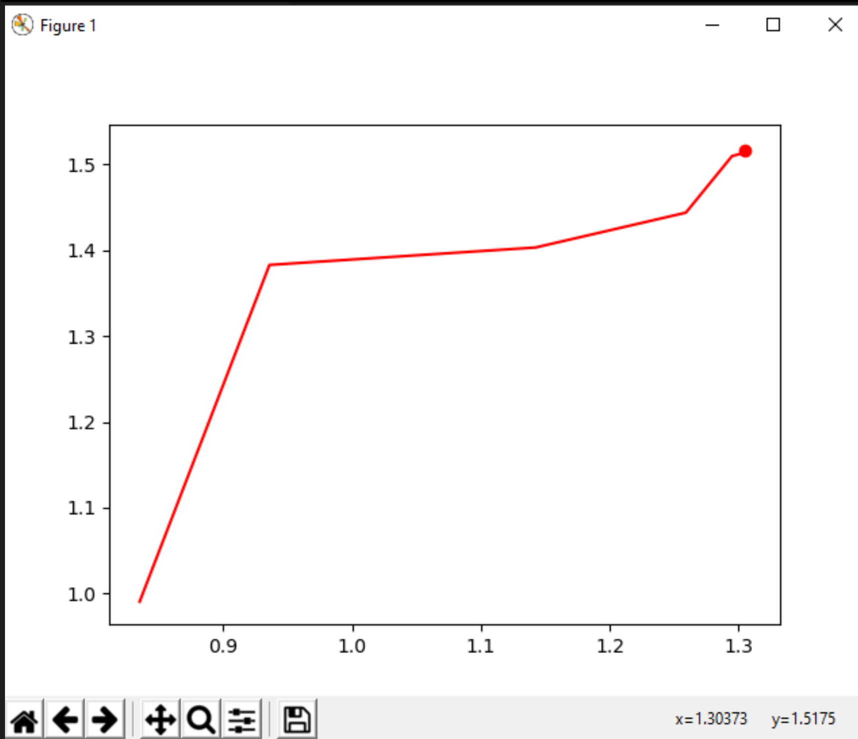
$$x = x_l + (x_u - x_l)r$$

```
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

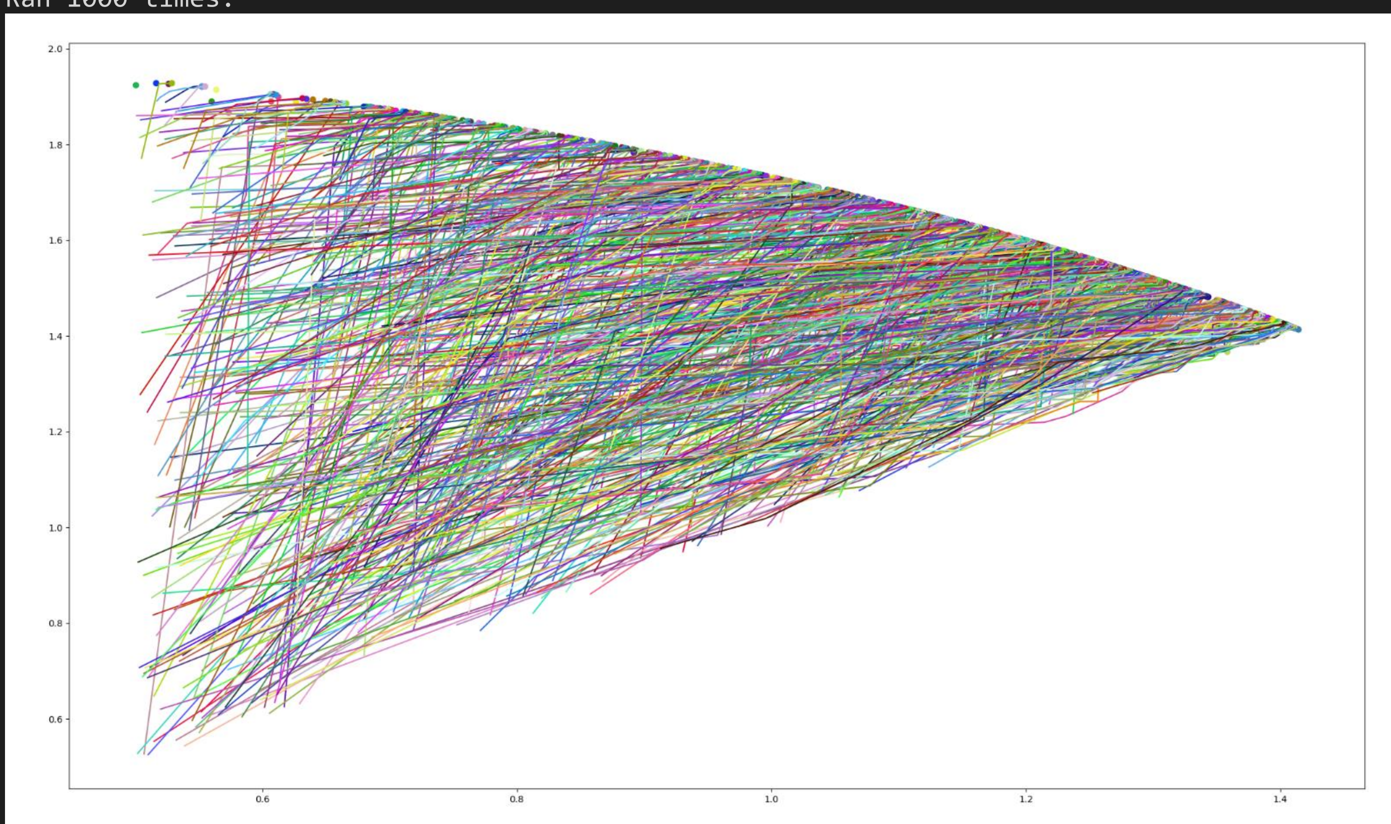
```
xi,yi,xLtempi,yLtempi = randomSearch(1000000,fun,isConstrFun4)
plt.plot(xLtempi,yLtempi,color="red", label="randWalk")
plt.scatter(xi,yi,color="red", label="randWalk")
plt.show()
```




```
\prob4.py'
The optimil Value: ( 1.3052754455400328 , 1.5153402295426164 )
```

hon: Current File (FinalProject) Ln 15, Col 17 Spaces: 4 UTF-8 CRLF Python

Ran 1000 times:



5. (10 points) Consider the optimization problem:

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

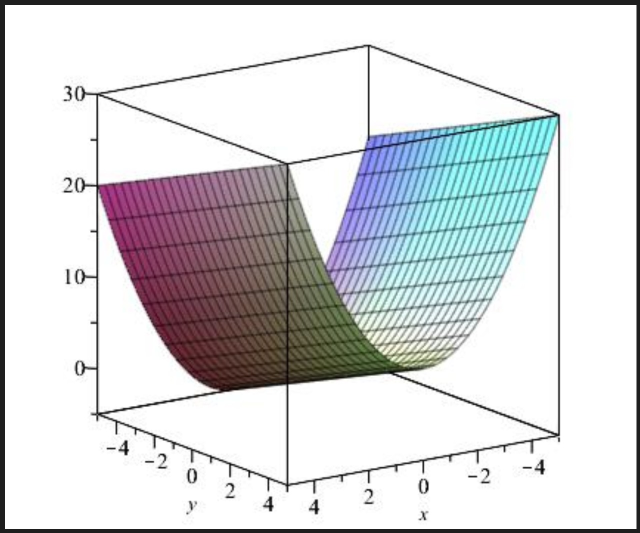
subject to the constraints

$x^2 + y \leq 4,$

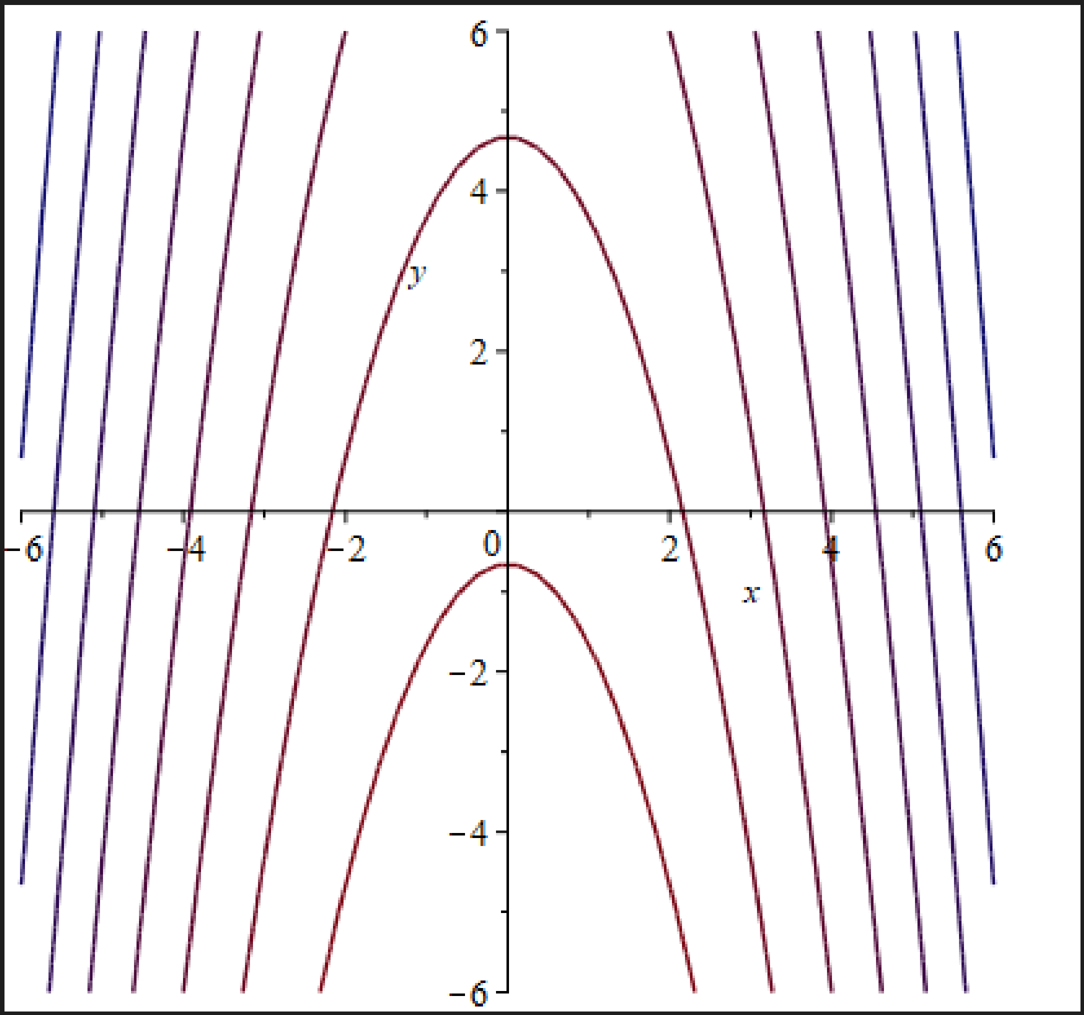
$-3x + y \leq 0,$

$x \geq 0.5,$

$y \geq 0.$

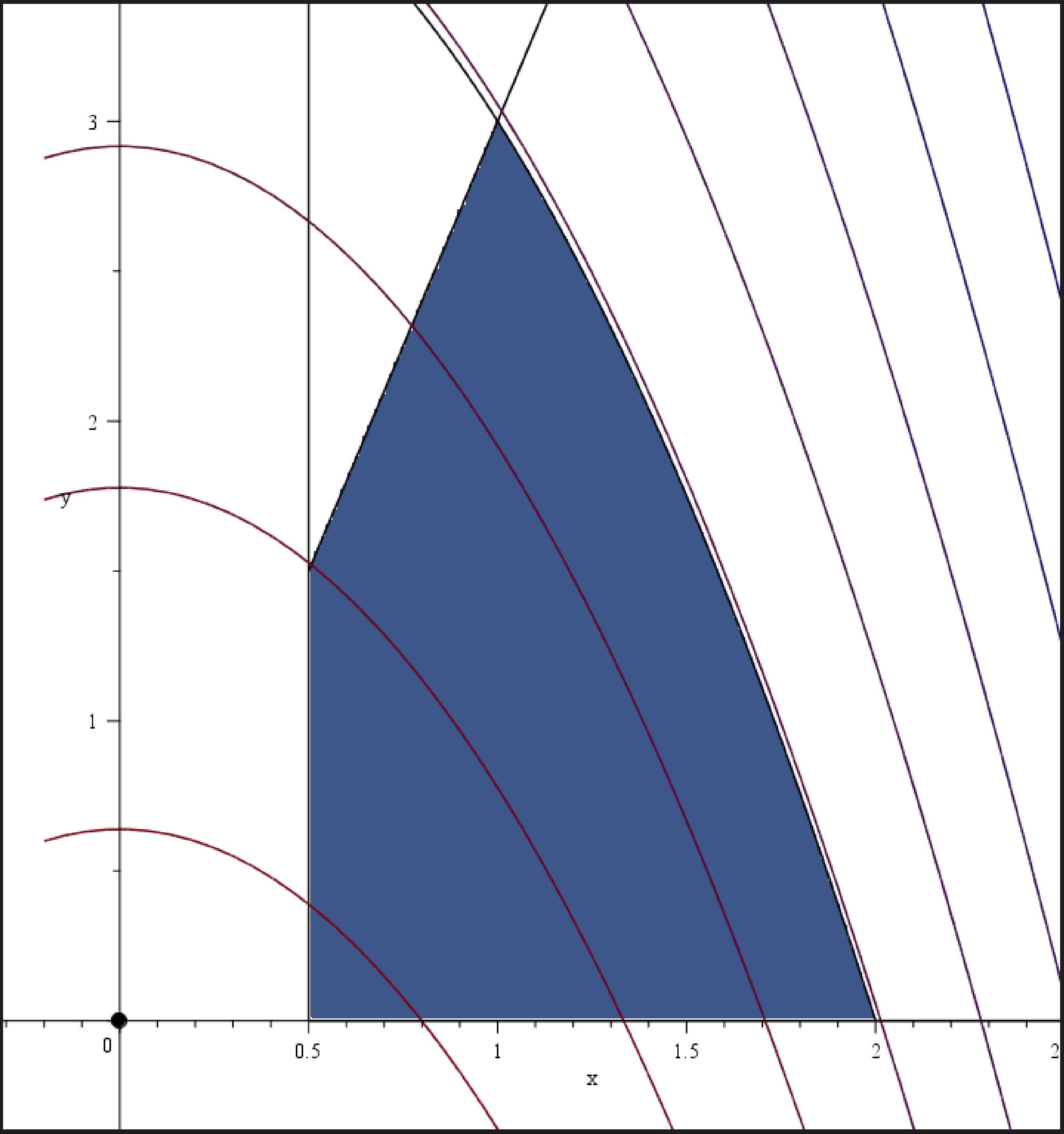


3D representation of the objective function
$f(x, y) = 3x + y$



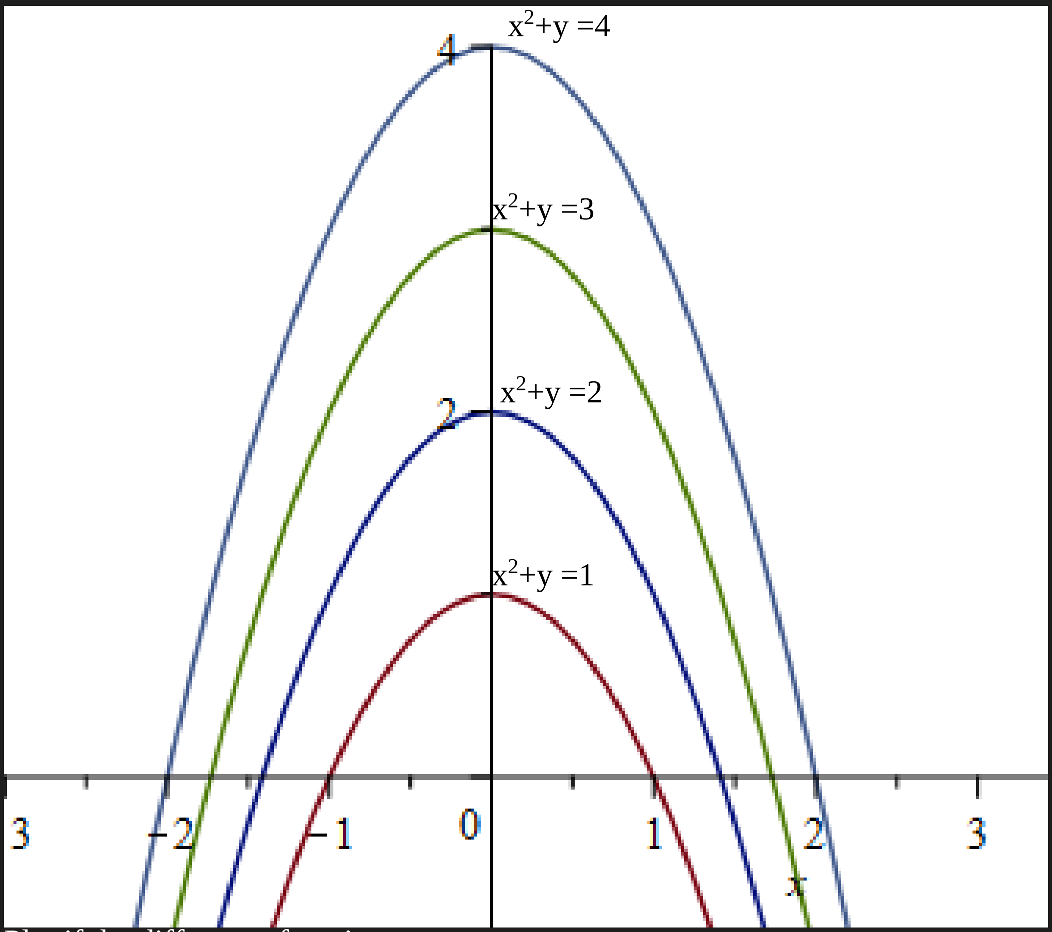
This is a contour plot of the objective function $f(x, y) = 3x + y$

(a) 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

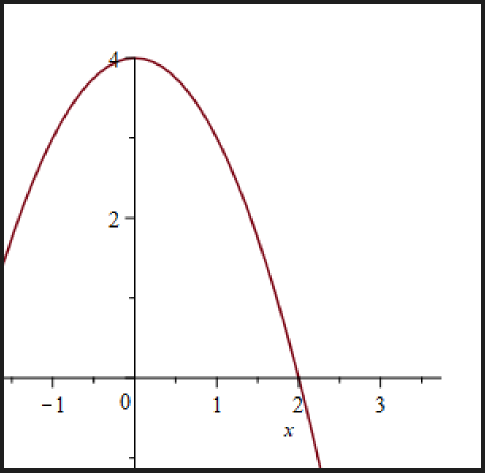


Plot if the different z functions

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 $x^2 + y = 4$

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



Plot of the function $x^2 + y = 4$

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

$y = \alpha x e^{\beta x}$

applied to the following data:

x	0.2	0.4	1	1.5
y	0.8	1.5	0.8	0.35

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

$$[[Z_j]^T[Z_j]]\{\Delta A\} = \{[Z_j]^T\{D\}\} \tag{17.35}$$

Find:

1. $$[Z_j] = \begin{bmatrix} \partial f_1/\partial a_0 & \partial f_1/\partial a_1 \\ \partial f_2/\partial a_0 & \partial f_2/\partial a_1 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \partial f_n/\partial a_0 & \partial f_n/\partial a_1 \end{bmatrix}$$

2. $$\{D\} = \begin{Bmatrix} y_1 - f(x_1) \\ y_2 - f(x_2) \\ \cdot \\ \cdot \\ \cdot \\ y_n - f(x_n) \end{Bmatrix}$$

3.
$$\{\Delta A\} = \begin{Bmatrix} \Delta a_0 \\ \Delta a_1 \\ \cdot \\ \cdot \\ \cdot \\ \Delta a_m \end{Bmatrix}$$

4. Solve for Delta A

$$[[Z_j]^T[Z_j]]\{\Delta A\} = \{[Z_j]^T\{D\}\} \tag{17.35}$$

Prob 6 continues to the next page →

$y := \alpha \cdot x \cdot \exp(\mathrm{beta} \cdot x)$

$$y := \alpha\, x\, \mathrm{e}^{\beta\, x} \tag{1}$$

$dyd\alpha := x \rightarrow x\, \mathrm{e}^{\beta\, x}$ #$\left(\frac{\mathrm{d}}{\mathrm{d}\,\alpha}\, y\right)$

$$dyd\alpha := x \rightarrow x\, \mathrm{e}^{\beta\, x} \tag{2}$$

$dyd\beta := x \rightarrow \alpha\, x^2\, \mathrm{e}^{\beta\, x}$ #$\left(\frac{\mathrm{d}}{\mathrm{d}\,\beta}\, y\right)$

$$dyd\beta := x \rightarrow \alpha\, x^2\, \mathrm{e}^{\beta\, x} \tag{3}$$

$$data := \begin{bmatrix} 0.2 & 0.8 \\ 0.4 & 1.5 \\ 1 & 0.8 \\ 1.5 & 0.35 \end{bmatrix}$$

$$data := \begin{bmatrix} 0.2 & 0.8 \\ 0.4 & 1.5 \\ 1 & 0.8 \\ 1.5 & 0.35 \end{bmatrix} \tag{4}$$

$\alpha := 1$#gess 1

$$\alpha := 1 \tag{5}$$

$\beta := 1$ #gess 1

$$\beta := 1 \tag{6}$$

$$Z0 := \begin{bmatrix} dyd\alpha(0.2) & dyd\beta(0.2) \\ dyd\alpha(0.4) & dyd\beta(0.4) \\ dyd\alpha(1.0) & dyd\beta(1.0) \\ dyd\alpha(1.5) & dyd\beta(1.5) \end{bmatrix} \qquad [Z_j] = \begin{bmatrix} \partial f_1/\partial a_0 & \partial f_1/\partial a_1 \\ \partial f_2/\partial a_0 & \partial f_2/\partial a_1 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \partial f_n/\partial a_0 & \partial f_n/\partial a_1 \end{bmatrix}$$

$$Z0 := \begin{bmatrix} 0.2442805516 & 0.04885611032 \\ 0.5967298792 & 0.2386919517 \\ 2.718281828 & 2.718281828 \\ 6.722533605 & 10.08380041 \end{bmatrix} \tag{7}$$

$with(LinearAlgebra):$

$Multiply(Transpose(Z0), Z0)$

$$\begin{bmatrix} 52.9972737034094 & 75.3321124358548 \\ 75.3321124358548 & 109.131447572473 \end{bmatrix} \quad \textbf{(8)}$$

$MatrixInverse(Multiply(Transpose(Z0), Z0))$

$$\begin{bmatrix} 1.00358108970191 & -0.692759833848217 \\ -0.692759833848217 & 0.487366958631947 \end{bmatrix} \quad \textbf{(9)}$$

$d := (Column(data, 2) - Column(Z0, 1))$

$$d := \begin{bmatrix} 0.557194484000000 \\ 0.903270120800000 \\ -1.91828182800000 \\ -6.37253360500000 \end{bmatrix} \quad \{D\} = \begin{Bmatrix} y_1 - f(x_1) \\ y_2 - f(x_2) \\ \cdot \\ \cdot \\ \cdot \\ y_n - f(x_n) \end{Bmatrix} \quad \textbf{(10)}$$

$Multiply(Transpose(Z0), d)$

$$\begin{bmatrix} -47.3792422191794 \\ -69.2310340141488 \end{bmatrix} \quad \textbf{(11)}$$

#Iteration 1 $[[Z_j]^T[Z_j]]\{\Delta A\} = \{[Z_j]^T\{D\}\}$ (17.35)

$DeltaA := Multiply(Multiply(MatrixInverse(Multiply(Transpose(Z0), Z0)), Transpose(Z0)), d)$

$$DeltaA := \begin{bmatrix} 0.411568085207192 \\ -0.91848252280737 \end{bmatrix} \quad \textbf{(12)}$$

$\alpha := \alpha + DeltaA[1]$

$$\alpha := 1.41156808520719 \quad \textbf{(13)}$$

$\beta := \beta + DeltaA[2]$

$$\beta := 0.081517477192620 8 \quad \textbf{(14)}$$

Iteration 2:

$$\alpha := \alpha + DeltaA[1]$$

$$\alpha := 3.77856168926106 \tag{11}$$

$$\beta := \beta + DeltaA[2]$$

$$\beta := -1.46023616733499 \tag{12}$$

Iteration 3

$$\alpha := \alpha + DeltaA[1]$$

$$\alpha := 10.1207114137591 \tag{11}$$

$$\beta := \beta + DeltaA[2]$$

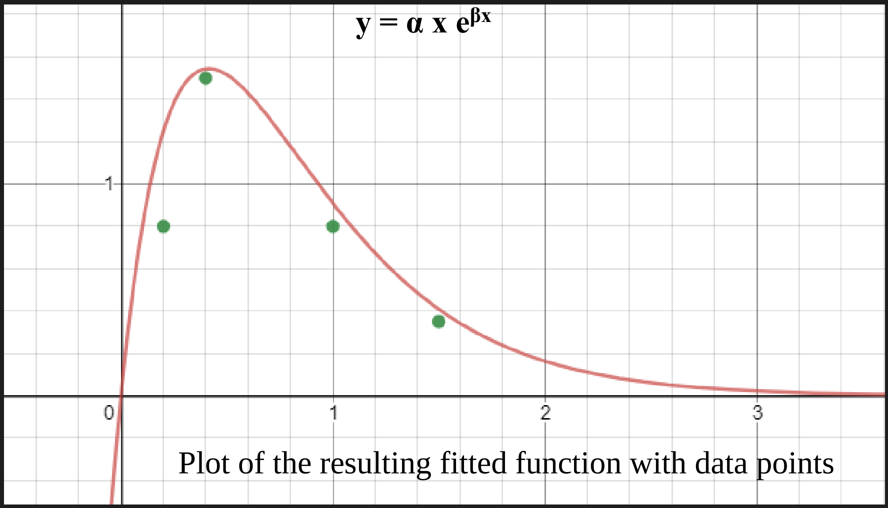
$$\beta := -2.41129088477523 \tag{12}$$

Any more additional iteration were not as close

alpha := 10.1207114137591
beta := -2.41129088477523

$y = \alpha$ x $e^{\beta x}$

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

$$DeltaA = \left(Z0^T \cdot Z0\right)^{-1} \cdot Z0^T \cdot d$$

$$\alpha := \alpha + DeltaA[1]$$

$$\beta := \beta + DeltaA[2]$$

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 $n \geq 3$, 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

$$I = \int_a^b f(x)\,dx.$$

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

```
def testBound(a,b):
    if a > b:
        temp = a
        a = b
        b = temp
        return
    elif a == b:
        print( "Error : the Lower bound is the smae as the upper bound")
        exit(-1)

def simpsons_1_3(a,b,n,funct):
    if n== 0:
        return 0
    testBound(a,b)

    step = ( b - a) /n

    sumVal = 0
    xList = np.arange(a + step, b,step)
    for i in range(len(xList)):

        if i == 0 or i % 2 == 0:
            sumVal += 4 * funct(xList[i])
        else:
            sumVal += 2 * funct(xList[i])

    return ( (b - a) * (funct(a) + sumVal + funct( b )) / (3*n))

def simpsons_3_8(a,b,n,funct):
    if n== 0:
        return 0
    testBound(a,b)
```

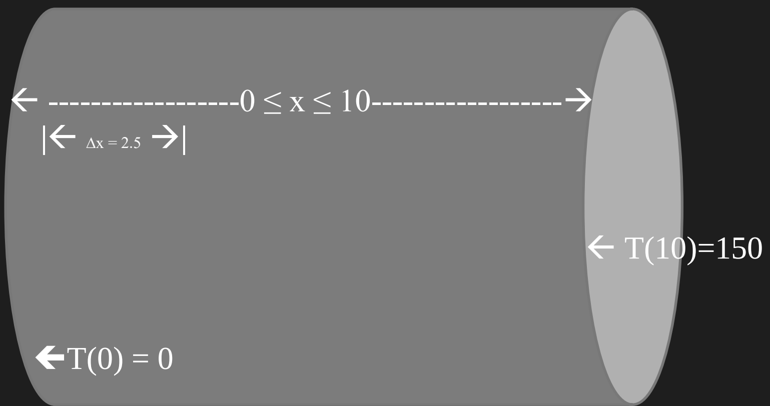
```

    # Interval Size  = step
    step = (( b -  a) / n)
    sumVal = funct( a) + funct( b)

    for i in range(1, n ):
        if (i % 3 == 0):
            sumVal = sumVal + 2 * funct( a + i * step)
        else:
            sumVal = sumVal + 3 * funct( a + i * step)

    return (( 3 * step) / 8 ) * sumVal

def prob2Algorithm(a,b,n,funct):
    # 1.a < b
    testBound(a,b)

    if (n%2 == 0 or n < 3):
        print("as stated in the Problem statement, only prescribed odd (n≥ 3) are allowed")
        print("You picked n = ",n)
        return None

    elif (n >= 3): # 2. prescribed odd number of subintervals n ≥ 3
        sumTot = 0
        step = (b-a)/n
        # 3.multiple-application Simpson’s 1/3 rule on the first n – 3 subintervals
        sumTot += simpsons_1_3(a,b-3*step,n-3,funct)

        # 4.Simpson’s 3/8 rule on the last 3 subintervals
        sumTot += simpsons_3_8(b-3*step, b, 3,funct)

        return sumTot
```

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

$$\frac{d^2T}{dx^2} - 0.15T = x^3 - 12x, \qquad (1)$$

where $0 \leq x \leq 10$. 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 $\Delta x = 2.5$ 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.



.

From page 855 in the book

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2}$$

$$sections = \frac{10}{2.5}$$

$$sections = 4.000000000$$

$$DetaX := 2.5$$

$$DetaX := 2.5$$

$$syst := \left\{ (1\cdot DetaX)^3 - 12\cdot(1\cdot DetaX) = \frac{(T0 - 2\cdot T1 + T2)}{DetaX^2}, \right.$$
$$(2\cdot DetaX)^3 - 12\cdot(2\cdot DetaX) = \frac{(T1 - 2\cdot T2 + T3)}{DetaX^2},$$
$$(3\cdot DetaX)^3 - 12\cdot(3\cdot DetaX) = \frac{(T2 - 2\cdot T3 + T4)}{DetaX^2},$$
$$\left. (4\cdot DetaX)^3 - 12\cdot(4\cdot DetaX) = \frac{(T3 - 2\cdot T4 + 150)}{DetaX^2} \right\}$$

$$syst := \{-14.375 = 0.1600000000\,T0 - 0.3200000000\,T1 + 0.1600000000\,T2, 65.000 = 0.1600000000\,T1 - 0.3200000000\,T2 + 0.1600000000\,T3, 331.875 = 0.1600000000\,T2 - 0.3200000000\,T3 + 0.1600000000\,T4, 880.000 = 0.1600000000\,T3 - 0.3200000000\,T4 + 24.00000000\}$$

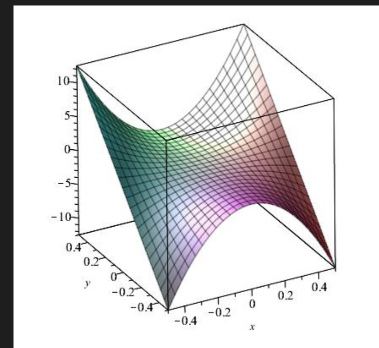
$$solve(syst, \{T0, T1, T2, T3, T4\})$$

$$\{T0 = 28345.31250 + 5.\,T4, T1 = 20604.68750 + 4.\,T4, T2 = 12774.21875 + 3.\,T4, T3 = 5350. + 2.\,T4, T4 = T4\}$$

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

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 100x^2 y,$$

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



Heat Flux

- insulated boundary conditions at x = 0,
- insulated boundary conditions at y = 0,
- T = 100°C at the boundary where x = 1,
- T = 100°C at the boundary where y = 1.

100°C	$T_{0.25,1}$	$T_{0.75,1}$	$T_{1,1}$
$T_{0,0.75}$	$T_{0.25,0.75}$	$T_{0.75,0.75}$	$T_{1,0.75}$
$T_{0,0.25}$	$T_{0.25,0.25}$	$T_{0.75,0.25}$	$T_{1,0.25}$
$T_{0,0}$	$T_{0.25,0}$	$T_{0.75,0}$	100°C

(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

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} = 0$$

$$TxyMap := \begin{bmatrix} T[0, 1] & T[0.25, 1] & T[0.75, 1] & T[1, 1] \\ T[0, 0.75] & T[0.25, 0.75] & T[0.75, 0.75] & T[1, 0.75] \\ T[0, 0.25] & T[0.25, 0.25] & T[0.75, 0.25] & T[1, 0.25] \\ T[0, 0] & T[0.25, 0] & T[0.75, 0] & T[1, 0] \end{bmatrix}$$

$$TxyMap := \begin{bmatrix} 100 & T_{0.25, 1} & T_{0.75, 1} & T_{1, 1} \\ 100 & T_{0.25, 0.75} & T_{0.75, 0.75} & T_{1, 0.75} \\ 100 & T_{0.25, 0.25} & T_{0.75, 0.25} & T_{1, 0.25} \\ 100 & 100 & 100 & 100 \end{bmatrix}$$

$T[0, 1] := 100$

$T_{0, 1} := 100$

$T[1, 0] := 100$

$T_{1, 0} := 100$

$T[0, 0.75] := 100$

$T_{0, 0.75} := 100$

$T[0, 0.25] := 100$

$T_{0, 0.25} := 100$

$T[0.25, 0] := 100$

$T_{0.25, 0} := 100$

$T[0.75, 0] := 100$

$T_{0.75, 0} := 100$

$T[0, 0] := 100$

$T_{0, 0} := 100$

$fun := (x, y) \rightarrow 100 \cdot x^2 \cdot y$

$fun := (x, y) \rightarrow 100\, x^2\, y$

$detaX := \frac{1}{n}$

$detaX := \frac{1}{2}$

$detaY := \frac{1}{n}$

$detaY := \frac{1}{2}$

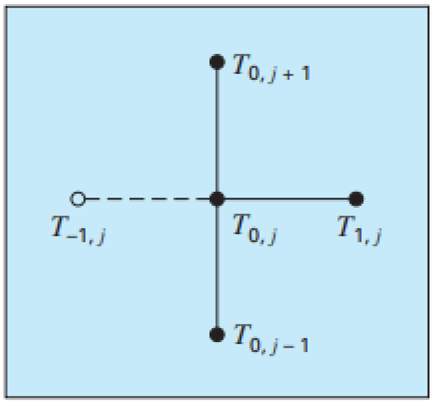


FIGURE 29.7

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

$$\begin{bmatrix} \frac{Td - 2\cdot T[0.25, 1] + 100}{detaX} + \frac{Te - 2\cdot T[0.25, 1] + 0}{detaY} & fun(0.25, 1) \\ \frac{Tr - 2\cdot T[0.25, 0.75] + TL}{detaX} + \frac{Td - 2\cdot T[0.25, 0.75] + Tup}{detaY} & fun(0.25, 0.75) \\ \frac{Tr - 2\cdot T[.25, 0.25] + TL}{detaX} + \frac{Td - 2\cdot T[.25, 0.25] + Tup}{detaY} & fun(.25, 0.25) \\ \frac{Tr - 2\cdot T[0.75, 1] + TL}{detaX} + \frac{Td - 2\cdot T[0.75, 1] + Tup}{detaY} & fun(0.75, 1) \\ \frac{Tr - 2\cdot T[0.75, 0.75] + TL}{detaX} + \frac{Td - 2\cdot T[0.75, 0.75] + Tup}{detaY} & fun(0.75, 0.75) \\ \frac{Tr - 2\cdot T[0.75, 0.25] + TL}{detaX} + \frac{Td - 2\cdot T[0.75, 0.25] + Tup}{detaY} & fun(0.75, 0.25) \\ \frac{Tr - 2\cdot T[1, 1] + TL}{detaX} + \frac{Td - 2\cdot T[1, 1] + Tup}{detaY} & fun(1, 1) \\ \frac{Tr - 2\cdot T[1, 0.75] + TL}{detaX} + \frac{Td - 2\cdot T[1, 0.75] + Tup}{detaY} & fun(1, 0.75) \\ \frac{Tr - 2\cdot T[1, 0.25] + TL}{detaX} + \frac{Td - 2\cdot T[1, 0.25] + Tup}{detaY} & fun(1, 0.25) \end{bmatrix}$$

$$\begin{bmatrix} 2\,Td - 8\,T_{0.25,\,1} + 200 + 2\,Te & 6.2500 \\ 2\,Tr - 8\,T_{0.25,\,0.75} + 2\,TL + 2\,Td + 2\,Tup & 4.687500 \\ 2\,Tr - 8\,T_{0.25,\,0.25} + 2\,TL + 2\,Td + 2\,Tup & 1.562500 \\ 2\,Tr - 8\,T_{0.75,\,1} + 2\,TL + 2\,Td + 2\,Tup & 56.2500 \\ 2\,Tr - 8\,T_{0.75,\,0.75} + 2\,TL + 2\,Td + 2\,Tup & 42.187500 \\ 2\,Tr - 8\,T_{0.75,\,0.25} + 2\,TL + 2\,Td + 2\,Tup & 14.062500 \\ 2\,Tr - 8\,T_{1,\,1} + 2\,TL + 2\,Td + 2\,Tup & 100 \\ 2\,Tr - 8\,T_{1,\,0.75} + 2\,TL + 2\,Td + 2\,Tup & 75.00 \\ 2\,Tr - 8\,T_{1,\,0.25} + 2\,TL + 2\,Td + 2\,Tup & 25.00 \end{bmatrix}$$

I think I was supposed to set the right Colum to 0

(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 $\lambda = 0.8$ in your iteration.

How are we suppose to use the relaxation factor on the system of equations?

29.2.2 The Liebmann Method

Most numerical solutions of the Laplace equation involve systems that are much larger than Eq. (29.10). For example, a 10-by-10 grid involves 100 linear algebraic equations. Solution techniques for these types of equations were discussed in Part Three.

29.2 SOLUTION TECHNIQUE **857**

Notice that there are a maximum of five unknown terms per line in Eq. (29.10). For larger-sized grids, this means that a significant number of the terms will be zero. When applied to such sparse systems, full-matrix elimination methods waste great amounts of computer memory storing these zeros. For this reason, approximate methods provide a viable approach for obtaining solutions for elliptical equations. The most commonly employed approach is *Gauss-Seidel,* which when applied to PDEs is also referred to as *Liebmann's method.* In this technique, Eq. (29.8) is expressed as

$$T_{i,j} = \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4} \tag{29.11}$$

and solved iteratively for $j = 1$ to n and $i = 1$ to m. Because Eq. (29.8) is diagonally dominant, this procedure will eventually converge on a stable solution (recall Sec. 11.2.1). Overrelaxation is sometimes employed to accelerate the rate of convergence by applying the following formula after each iteration:

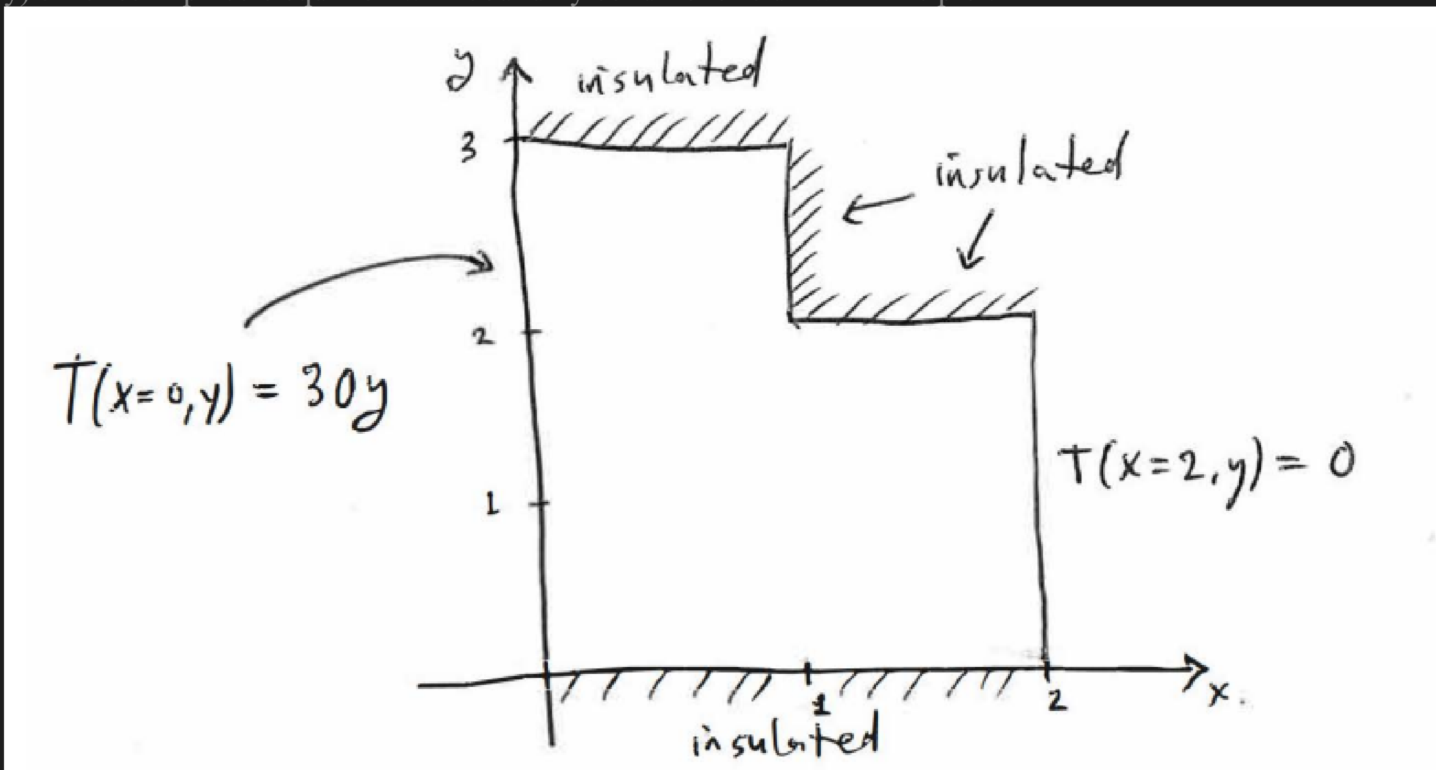
$$T_{i,j}^{\text{new}} = \lambda T_{i,j}^{\text{new}} + (1 - \lambda) T_{i,j}^{\text{old}} \tag{29.12}$$

where $T_{i,j}^{\text{new}}$ and $T_{i,j}^{\text{old}}$ are the values of $T_{i,j}$ from the present and the previous iteration, respectively, and λ is a weighting factor that is set between 1 and 2.

As with the conventional Gauss-Seidel method, the iterations are repeated until the absolute values of all the percent relative errors $(\varepsilon_a)_{i,j}$ fall below a prespecified stopping criterion ε_s. These percent relative errors are estimated by

$$|(\varepsilon_a)_{i,j}| = \left| \frac{T_{i,j}^{\text{new}} - T_{i,j}^{\text{old}}}{T_{i,j}^{\text{new}}} \right| 100\% \tag{29.13}$$

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.

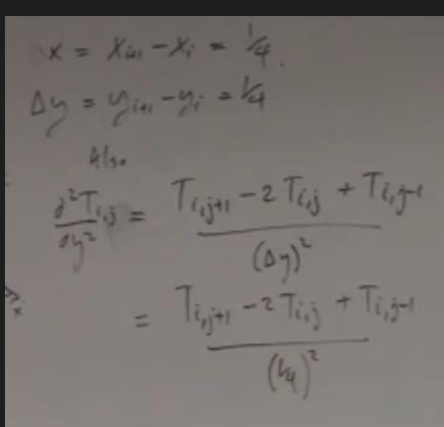


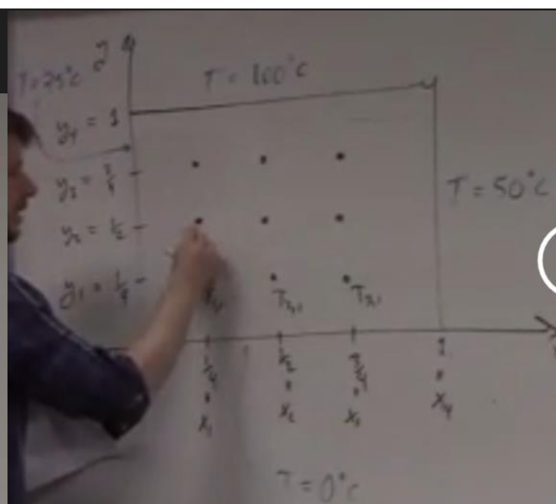
(a) Using **n = 2** subintervals in the x direction and **m = 3** subintervals in the y direction, set up a system of equations for the temperature Ti,j , at any nodes where the temperature is not specified in the problem.

n = 2
m = 3

Liebmann's method. In this technique, Eq. (29.8) is expressed as

$$T_{i,j} = \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4} \tag{29.11}$$





```python
import numpy as np
#          0       1      2   x values

#   0_     |   0   |   1  |  i/m values
#   1_ 0-| [[(0, 0), (0, 1)],
#   2_ 1-|  [(1, 0), (1, 1)],
#   3_ 2-|  [(2, 0), (2, 1)]]
#   ^   ^
#   |   |_  j/n values
#   |
#   |_  y values

#            0|.5|1.5|2  x values
#            0| 1| 2 |3  i/m values
#  0   0   [[0| 0| 0 |0]
#  .5  1    [0| 0| 0 |0]
# 1.5  2    [0| 0| 0 |0]
# 2.5  3    [0| 0| 0 |0]
# 1.5  4    [0| 0| 0 |0]]



def main():
    # n= 4
    # m=4
    # meand ther 4 rect  by 4 rect
    ymax =3
    xmax =2
    n = 2 #
    m = 3

    stepY = ymax/n
    stepX = xmax/m
    print("stepX: ",stepX)
    print("stepY: ",stepY)



    sumVal= 0
    mat = [[0 for j in range(n+1)] for i in range(m+1)]
    print(np.matrix(mat))

    y =np.arange(n).tolist()
    err =0.000001
    while(a >= err):
        for i in range( n):
            y[i] = 0
            a = 0
```

```
        for i in range( m):

            sumVal = 0
            for t in range(t ,a):

                a = err

                y[i] = y[t]


        print("\n")

        # for i in range( n):
    return
```

This question was very difficult, and I could not compute the result even using code. I did not feel I had enough exposer to understand how to approach these finite difference problems. I watch the lecture and read the book many times and still was unsure on how to approach this problem.

My plan was to create a 2D array that represented the 2D service and iterate across it to develop a system of equations. From that system of equations, coefficient could be calculated and then be reapplied to repeated iterations within an error threshold.

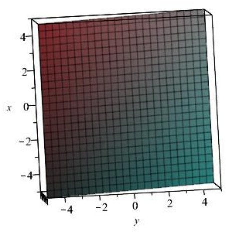
$solve(30 \cdot y + 2 = 0, y)$

$$-\frac{1}{15}$$

$f := 30\,y + x$

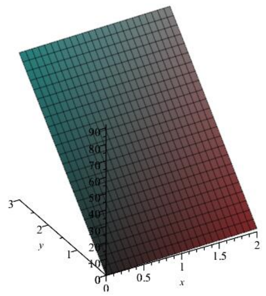
$$f := 30\,y + x$$

$smartplot3d[x, y](\ 30 * y + x\ )$



$with(plots):$

$plot1 := plot3d(30\,y + x, x = 0 ..2, y = 0 .. 3, orientation = [\,-120, 60\,],\ axes = normal);$



(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 = 1.2 in your iteration.

After reading the book and attempting to follow the examples provided, I was not user how I could attempt this portion of the problem without successfully creating a system of equations.

If I did have the system of equations I think it was unclear how to apply the relaxation factor to each iteration.

29.2.2 The Liebmann Method

Most numerical solutions of the Laplace equation involve systems that are much larger than Eq. (29.10). For example, a 10-by-10 grid involves 100 linear algebraic equations. Solution techniques for these types of equations were discussed in Part Three.

29.2 SOLUTION TECHNIQUE **857**

Notice that there are a maximum of five unknown terms per line in Eq. (29.10). For larger-sized grids, this means that a significant number of the terms will be zero. When applied to such sparse systems, full-matrix elimination methods waste great amounts of computer memory storing these zeros. For this reason, approximate methods provide a viable approach for obtaining solutions for elliptical equations. The most commonly employed approach is *Gauss-Seidel,* which when applied to PDEs is also referred to as *Liebmann's method.* In this technique, Eq. (29.8) is expressed as

$$T_{i,j} = \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4} \tag{29.11}$$

and solved iteratively for $j = 1$ to n and $i = 1$ to m. Because Eq. (29.8) is diagonally dominant, this procedure will eventually converge on a stable solution (recall Sec. 11.2.1). Overrelaxation is sometimes employed to accelerate the rate of convergence by applying the following formula after each iteration:

$$T_{i,j}^{\text{new}} = \lambda T_{i,j}^{\text{new}} + (1 - \lambda) T_{i,j}^{\text{old}} \tag{29.12}$$

where $T_{i,j}^{\text{new}}$ and $T_{i,j}^{\text{old}}$ are the values of $T_{i,j}$ from the present and the previous iteration, respectively, and λ is a weighting factor that is set between 1 and 2.

As with the conventional Gauss-Seidel method, the iterations are repeated until the absolute values of all the percent relative errors $(\varepsilon_a)_{i,j}$ fall below a prespecified stopping criterion ε_s. These percent relative errors are estimated by

$$|(\varepsilon_a)_{i,j}| = \left| \frac{T_{i,j}^{\text{new}} - T_{i,j}^{\text{old}}}{T_{i,j}^{\text{new}}} \right| 100\% \tag{29.13}$$