# מציאת שורשים

### הגדרת הבעיה

שיטות למציאת שורש (ערך שמאפס את הפונקציה( עבור פונקציה מכל סוג ,כאשר השורש המדויק מסומן באחד מהסימונים הבאים .

**שיטת החציה:** .

בהנתן ששורש מסוים של פונקציה נתון בקטע שבו פונקציה מחליפה סימן אזי על פי משפט ערך הביניים קיים ערך שבו הפונקציה מתאפסת , לכן נמצא את השורש המקורב על פי האלגוריתם הבא:

1. חצה את הקטע לשתיים כך ש
2. אם  אזי השורש המקורב הוא c

אחרת בדוק

אם  אזי a=c

אחרת b=c.

3. חזור ל 1

בצורה זו הקטע מצטמצם עד לבחירת הקטע המינימאלי במסגרת ההתניה על השורש או מגבלות המחשב.

יתרון: תמיד מתכנס ,קצב קבוע (אחרי *n* איטרציות גודל הקטע קטן פי ).

חסרונות: התכנסות איטית מדי, צריכים לדעת מראש קטע שבו פונקציה מחליפה סימן.

שימוש אופייני: שלב ראשון של פתרון. אם שיטה מהירה יותר לא מתכנסת מהניחוש הראשוני הקיים, מתקרבים לשורש בשיטת חציה ואז חוזרים לשיטה מהירה יותר.

****

import sys  
def bisection(a, b,f, epsilon):  
 *"""* ***:param*** *a: first value to check in the function* ***:param*** *b: second value to check in the function* ***:param*** *f: The function to check* ***:param*** *epsilon: error tolerance* ***:return****: m: the root of the function  
 """* m = (a + b) / 2.0  
 while abs(a - b) > epsilon:  
 if f(m) == 0:  
 return m  
 elif f(a) \* f(m) < 0:  
 b = m  
 else:  
 a = m  
 m = (a + b) / 2.0  
 return m

**שיטת Newton – Raphson:** פותרים משוואה . בנקודה מקרבים את הפונקציה ע"י המשיק ומוצאים את הנקודה הבאה כחיתוך בין משיק לציר *x*: . השיטה מתכנסת עם סדר 2 ,השיטה איננה מתכנסת אם הערך של הנגזרת בנקודה שווה לאפס .

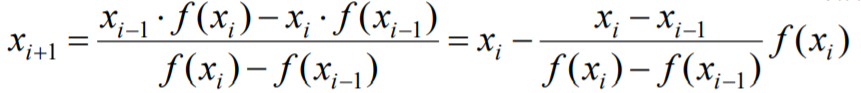
יתרונות וחסרונות: השיטה הכי יעילה באזורים קרובים לשורש. באזורים רחוקים היא עלולה להתבדר. אסור שהנגזרת תהיה שווה לאפס.

.

def newton\_rapson(func, x=0, it=1, h=0.00000001):  
 *"""* ***:param*** *func: the function to get it's roots* ***:param*** *x: starting guess(default=0)* ***:param*** *it: the maximum number of allowed iterations(default=1)* ***:param*** *h: accuracy factor of the method (default=0.00000001)* ***:return****: An estimated root of the funtion  
 """* def der(f, h):  
 return lambda x0: (f(x0 + h) - f(x0)) / h  
  
 if der(func, h)(x) == 0:  
 x = 1  
 for i in range(it):  
 fd = func(x)  
 ft = der(func, h)(x)  
 x = x - (fd / ft)  
 return x

**שיטת המיתר (Secant):** זוהי שיטה איטרטיבית למצוא שורשים של פונקציה רציפה בעלת משתנה אחד. נתונה פונקציה ונרצה לחשב את השורשים של המשוואה מהצורה .

בשיטה זו מחברים את שתי הנקודות האחרונות ע"י ישר ומוצאים חיתוך בין ישר זה לציר *x*.  
בהינתן שתי נקודות xi וxi-1 מוצאים את xi+1 , כך מוצאים את הישר העובר ב- xi וxi-1 ולוקחים את xi+1 להיות חיתוכו עם ציר ה-X , הנוסחה:



חסרונות: השיטה לא תמיד מתכנסת, אבל אם היא מתכנסת אז מהר יותר: סדר התכנסות בערך 1.6.

יתרונות: לעומת Newton – Raphson לא צריכים לחשב נגזרת (זה יכול להיות קשה). כתוצאה מזה בכל איטרציה מחשבים רק את הפונקציה



xI ו xI\_1 הינם ניחושים התחלתיים.   
נסמן את eS גודל קטן כלשהו המהווה רמת מובהקות מהו הפער בין שני הקבוצות.  
amtIterations כמות האיטרציות ככל שנרבה באיטרציות כך נקבל ערך קרוב יותר לשורש.

def secant(f,xI,xI\_1,amtIterations,eS):  
 *"""* ***:param*** *f: the function to use this method on* ***:param*** *xI: the first value guess* ***:param*** *xI\_1: the second value guess* ***:param*** *amtIterations: the number of iterations* ***:param*** *eS: the error tolerance* ***:return****: the estimated root and error margin, or an error if there is no success.  
 """* count=0  
 eA=0  
 while count<amtIterations or eS < eA:  
 fXI = f(xI)  
 fXI\_1 = f(xI\_1)  
 temp = xI  
 if (fXI-fXI\_1) == 0: break #avoiding dividing by 0  
 xI = xI - ((fXI \* (xI - xI\_1))/(fXI - fXI\_1))  
 xI\_1 = temp  
 eA = float(abs((xI - xI\_1)/xI) \* 100)  
 count += 1  
 print(count,")Estimated root: ", xI,"Approximate error:", eA)  
 print("number of iterations:",count)  
 return [xI , eA]

**מטריצות:**

מערכות משוואות ליניאריות נהוג לפתור במטריצות

למשל:

הערה: בכל הניסויים נתייחס אך ורק למטריצות ריבועיות מסדר n\*n ולשם פשטות נדגים כל דבר על מטריצות 2\*2 או 3\*3

להלן הצגת מטריצה בפייטון:

import numpy as np  
from scipy.linalg import solve\_triangular as solve  
from numpy.linalg import inv  
class Matrix(object):  
 def \_\_init\_\_(self,A,b=0):  
 *"""  
 creates a matrix and a solution vector for it.* ***:param*** *A: The matrix to construct* ***:param*** *b: The solution vector by default the vector is a vector of 0's  
 """*ח self.D = np.zeros\_like(A)  
 self.diag()  
 self.U= np.zeros\_like(A)  
 self.upper()  
 self.L= np.zeros\_like(A)  
 self.lower()  
 self.b=b  
 if b==0:  
 self.b=np.array( [0.0 for i in range(len(A))])

מייצג מחלקה של מטריצה המשתמשת בNUMPY כדי לעשות מערכים מהצורה NARRAY

הייצוג כולל גם ווקטור תוצאות: b כמו בייצוג למעלהץ

**חלוקת LU:**

יצירת מטריצה כולל גם פירוק המטריצה לרכיבים L,D,U

לדוגמא (מטריצה 3\*3):

חלק עליון

*חלק תחתון*

*חלק עליון*

קוד שמסדר את החלוקה הזאת:

def LU(self):  
 *"""  
 An algorithm to update the attributes of L,D,U in case the attribute of A is changed.  
 """* self.diag()  
 self.upper()  
 self.lower()  
  
def diag(self):  
 *"""  
 Defines the diagonal(D) attribute of the matrix  
 """* for i in range (len(self.mat)):  
 self.D[i][i]=self.mat[i][i]  
def upper(self):  
 *"""  
 Defines the upper(U) attribute of the matrix  
 """* for i in range(0,len(self.mat)):  
 for j in range (0,len(self.mat)):  
 if (i<j):  
 self.U[i][j] = self.mat[i][j]  
def lower(self):  
 *"""  
 Defines the lower(L) attribute of the matrix  
 """* for i in range(0,len(self.mat)):  
 for j in range (0,len(self.mat)):  
 if (i>j):  
 self.L[i][j] = self.mat[i][j]

הפיכות מטריצה: במהלך האלגוריתם משתמשים בהפיכות מטריצה על מנת לבצע איטרציות ולשם כך צריך לבדוק האם מטריצה הפיכה נעזר בNUMPY.LINALG.INV:

def check\_invertible(self):  
 *"""  
 Checks the matrix is invertible or not* ***:return****: boolean: True if it's invertible or False if it's singular  
 """* try:  
 inv(A)  
 except str:  
 return False  
 return True

**שיטת האלמינציה של גאוס:**

זוהי שיטה שמביאה למשוואה ליניארית פתרון על ידי דירוג מטריצה למטריצה משולשית עליונה מהצורה:

אנו מעוניינים לבחור מטריצה המניבה תוצאות כך ששינוי קטן בווקטור הפתרונות לא יגרום לשינוי גדול בווקטור הנעלמים (x) לשם כך יש לנו חישוב של cond(A) הקובע בעזרת נורמות מקסימום והכפלת נורמת המקסימום של המטריצה ונורמת המקסימום של המטריצה ההופכית לה האם המטריצה שבחרנו טובה מספיק ולא תניב תוצאות השונות יותר מדי בסדרי גודל

המספר הזה הוא מספר הגדול מ1 וכל חזקה של 10 שלו גורמת למספר נוסף בתוצאה להיות שונה לכן נרצה שcond(A) של המטריצה שבחרנו יהיה קטן ככל האפשר(!) והיא גם חייבת להיות הפיכה

קוד לחישוב נורמות וcond(A)

def cond(self):  
 *"""  
 Calculates the cond(A) of a matrix the formula is: normal(A)\*normal(A^-1)* ***:return****: The value of condition A the bigger this value the less accurate the matrix will be.  
 """* def normal(A):  
 sum = 0  
 maxSum = 0  
 for i in A:  
 for j in i:  
 sum += abs(j)  
 if sum > maxSum:  
 maxSum = sum  
 sum = 0  
 return maxSum  
 print("the inverted matrix is:\n",inv(self.mat))  
 return normal(self.mat) \* normal(inv(self.mat))

השיטה פועלת כך:

בכל שלב בודקים אם האיבר באלכסון הוא האיבר הגדול ביותר במידה ולא אז מחליפים שורה עם השורה שבו האיבר הכי גדול (בערך מוחלט) באותו טור נשים לב כי אם המטריצה הפיכה תמיד יהיה איבר הגדול מ0 בכל טור (אם יש עמודת ים0 דרגת המטריצה אינה n ואז היא אינה הפיכה)

להלן הקוד שעושה זאת:

def biggest\_value\_swap(A, b, i, j):  
 def switch\_lines(A, l1, l2): #switches 2 lines by multiplying by an #adequate elemental matrix  
 B = np.identity(len(A))  
 B[l1][l1] = 0  
 B[l1][l2] = 1  
 B[l2][l2] = 0  
 B[l2][l1] = 1  
 return B.dot(A)  
 maximum = A[i][j]  
 index = i  
 biggest\_index = i  
 while index < len(A):  
 if abs(A[index][j]) > maximum:  
 maximum = abs(A[index][j])  
 biggest\_index = index  
 index = index + 1  
 if A[biggest\_index][j] == 0:  
 return 'error'  
 b[biggest\_index], b[i] = b[i], b[biggest\_index]  
 return [switch\_lines(A, i, biggest\_index), b]

לאחר מכן כופלים את המטריצה שאותה רוצים לדרג במטריצה אלמנטרית משולשית תחתונה מתאימה מהסוג:

כאשר בדוגמא לעיל (שהיא הדגמה על מטריצה 3\*3) רק אחד מ x y z אינו 0 ואותו איבר הוא בעצם הכופל של איבר האלכסון על מנת לאפס את אותו איבר מתחת לאלכסון לפי הנוסחא:

*כאשר איבר האלכסון לא 0, זאת ניתן להבטיח על ידי החלפת השורה שאנו עושים בכל שלב כפל המטריצה באותה מטריצה אלמנטרית* ***משמאל*** *מבטיח את איפוס האיבר בשורה מתחתיו, נשים לב כי גם את וקטור הפתרונות יש לשנות באותה פעולה.*

def gauss\_scalling(A, b, i):  
 def elemental(i, j, n, val):  
 *"""Makes an elementary matrix in size of n\*n"""* A = np.identity(n)  
 A[i][j] = val  
 return A  
 index = i + 1  
 while index < len(A):  
 multiplier = -1 \* (A[index][i] / A[i][i])  
 A = elemental(index, i, len(A), multiplier).dot(A)  
 b[index] = b[index] + (multiplier \* float(b[i]))  
 index = index + 1  
 matrix = [A, b]  
 return matrix

*אחר כך יש לעשות את הפעולה על כל איבר אחר באלכסון עד הגעה לאיבר שהוא האיבר האחרון ואין מתחתיו איברים.*

*אחרי הדירוג למשולשית תחתונה אפשר להגיע לתוצאה זאת נעשה באמצעות פונקציית Scipy*

*Solve\_triangle*

for i in range(0, len(A)):  
 print('{2})A:\n{0},\nb:{1}'.format(A, b,i+1))  
 [Ai, b] = biggest\_value\_swap(A, b, i, i)  
 if not np.array\_equal(Ai, A): #So it doesn't print twice the same values  
 print(') A:\n{0},\nb:{1}'.format(Ai, b))  
 A = Ai  
 [A, b] = gauss\_scalling(A, b, i)  
print("Took {0} iterations".format(i+1))  
x = solve(A,b) # solving triangular matrix.  
print ("The result after solving a triangular scaled Matrix:{0}".format(x))  
return x

**שיטות איטרטיביות על מטריצות**

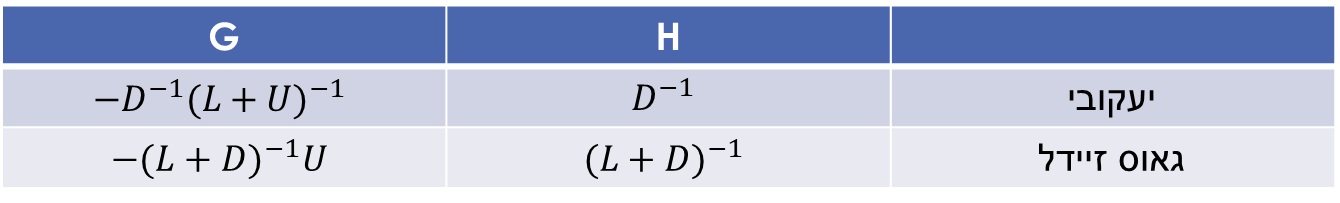
השיטות האיטרטיביות עליהן נדון הן 3 הבאות: גאוס סיידל, יעקובי וSOR

יעקובי וגאוס סיידל הן שיטות איטרטיביות מאוד דומות אשרלוקחות את מערכת המשוואות:

*ולאחר פישוט הביטוי מגיעים לביטוי מהצורה:*

*מכאן גאוס סיידל ויעקובי מנסות לפתור את מערכת המשוואות בעזרת ניחושים על האיברים a,b,c כך שבכל פעם מציבים את הabc שמצאנו באיטרציה הקודמת ניתן לתאר את 2 השיטות על ידי אותה נוסחא:*

*כאשר G וH שונות בין יעקובי לגאוס ההצבות של G וH בין הנוסחאות הן:*



*וזה מתבטא בקוד כך:*

def gauss\_seidel(self):  
 G = (-1 \* inv(self.D + self.L)).dot(self.U)  
 print("G:", G)  
 H = inv(self.L + self.D)  
 print("H:", H)  
 return [G,H]  
def Jacobi(self):  
 G =((-1 \*inv(self.D)).dot(inv(self.L+self.U)))  
 print("G:", G)  
 H = inv(self.D)  
 print("H:", H)  
 return[G,H]

*המעבר על איטרציות הוא כך: G H קבועים במשך עכל האיטרציות כל מה שמשתנה זה הxr ןxr+1 הניחוש ההתחלתי לabc (באופן כללי יכול להיות ווקטור בכל אורך כל עוד הוא תואם לגודל המטריצה) הוא לא משנה, לקחנו כניחוש את [1,0,0,.....] באופן שרירותי.*

*הניחוש ההתחלתי לא משנה (הוכח בהרצאה/ידוע).*

*להלן הקוד:*

def iterative(self,method,max\_iter=10000,tolerance=0.00001):  
 *"""  
 Calculates a solution for a matrix and a solution vector using 2 methods  
 that can be choosen when the function is called* ***:param*** *method: which method to use either "Jacobi" or "Gauss Seidel" are the only valid inputs* ***:param*** *max\_iter: max number of iterations the default is 10000* ***:param*** *tolerance: tolerance of solution the default is 0.0001* ***:return****: either an error (in string) if something gone wrong an exception if the matrix ins't invertable,  
 or the solution of the matrix and it's solution vector  
 """* [A,b]=self.mat,self.b  
 def gauss\_seidel(self):  
 G = (-1 \* inv(self.D + self.L)).dot(self.U)  
 print("G:", G)  
 H = inv(self.L + self.D)  
 print("H:", H)  
 return [G,H]  
 def Jacobi(self):  
 G =((-1 \*inv(self.D)).dot(inv(self.L+self.U)))  
 print("G:", G)  
 H = inv(self.D)  
 print("H:", H)  
 return[G,H]  
 iteration\_num = 0  
 x = np.zeros\_like(self.b)  
 x[0]=1  
 if (method=="Jacobi"):  
 [G,H]=Jacobi(self)  
 elif (method == "Gauss Seidel"):  
 [G,H]=gauss\_seidel(self)  
 else:  
 return "Wrong input"  
 xI=x  
 while (iteration\_num!=max\_iter):  
 xI= G.dot(x)+H.dot(self.b)  
 iteration\_num += 1  
 print(iteration\_num,")",x)  
 if (Matrix.iterative\_convergence(self,x,xI,tolerance)):  
 print("number of Iterations:",(iteration\_num))  
 return x  
 x = xI  
 if x[0]>10e+33: #putting an upper limit so the solution doesn't go too high..  
 return "does not converge"  
 return "does not converge"

***תנאי העצירה:***

*כל השיטות האיטרטיביות המוצגות במסמך זה משתמשות באותו תנאי עצירה:*

*(להוסיף הסבר מקובץ משתמש בנורמלים)*

*הקוד:*

def iterative\_convergence(self,x,xI,tolerance):  
 *"""  
 checks if a loop using a vector as result needs to end or not calculating if 2 vectors are close enough, using their normals.* ***:param*** *x: Previous x (xr)* ***:param*** *xI: new x (xr+1)* ***:param*** *tolerance: the tolerance factor of the iteration* ***:return****: True if the iterative loop ended in success, False rhe loop needs to keep going.  
 """* if Matrix.check\_invertible(self) is False:  
 return  
 diff1norm = 0.0  
 oldnorm = 0.0  
 for i in range(len(self.b)):  
 diff1norm = diff1norm + abs(x[i] - xI[i])  
 oldnorm = oldnorm + abs(xI[i])  
 if oldnorm == 0.0:  
 oldnorm = 1.0  
 norm = diff1norm / oldnorm  
 if (norm < tolerance):  
 return True  
 return False

*השיטות הנ"ל לא תמיד מתכנסות יש דרך לבדוק האם המטריצה תתכנס וזה על ידי בדיקה האם האלכסון הראשי הוא דומיננטי כלומר האם כל איברי האלכסון הראשי גדולים מסכום שאר איברי אותה שורה, הקוד מכיל בדיקה לאותה תכונה*

def has\_dominant\_diag(self):  
 *"""  
 Checks if a matrix has a dominant diagonal* ***:return****: True if it does, False if it doesn't  
 """* summ=0  
 current=0  
 for i in range(len(self.mat)):  
 for j in range(len(self.mat)):  
 if i==j:  
 current=abs(self.mat[i][i])  
 else:  
 summ+=abs(self.mat[i][j])  
 if (summ>current):  
 return False  
 summ=0  
 return True

***שיטת SOR***

*שיטת SOR היא גם שיטה איטרטיבית לפתירת מערכת משוואות היא מתבססת על פירות מערכת המשוואות באופן הבא:*

*A=L+D+U*

*נסמן*

לבסוף הגיעו לנוסחא הבאה:

כאשר w קבוע בין 0 ל2,שהגיעו אליו אחרי בדיקות רבות (ניסוי וטעייה בעיקר)

להלן שיטת SOR בפייטון:

def sor(self,w=1.5,max\_iter=1000,tolerance=0.00001):  
 *"""* ***:param*** *w: a variable between 0 and 2 (default is 1.5)* ***:param*** *max\_iter maximum iterations the code can go up to 1000 by default.* ***:param*** *tolerance: tolerance of solution the default is 0.0001* ***:return****: the solution to a matrix.  
 """* if Matrix.check\_invertible(self) is False:  
 return  
 if (w<=0 or w>=2):  
 return "Error, cannot initiate the function"  
 x=np.zeros\_like(self.b)  
 x[0]=5 #don't want the initial vector to be all 0's but the actual guess doesn't matter much.  
 xI=x  
 iteration\_num=0  
 while(iteration\_num!=max\_iter):  
 iteration\_num+=1  
 xI=(inv((self.D + w\*self.L)).dot((1-w)\*self.D - w \* self.U)).dot(x) + (w \* inv((self.D + w\*self.L)).dot(self.b))#the sor formula  
 print(iteration\_num, ")", x)  
 if (Matrix.iterative\_convergence(self,x, xI, tolerance)):  
 print("number of Iterations:", (iteration\_num))  
 return x  
 x = xI  
 if x[0] > 10e+33: # putting an upper limit so the loop doesn't go too high.. needlessly  
 return "does not converge"  
 return "does not converge"

**פולינום אינטרפולציה:**

בקבלת קוד מטבלה של ניסויים ותוצאות אנו נרצה ליצור מטריצה המייצגת את הבעיה שלנו, על פי פולינומים בדרגה של מספר הניסויים שבוצעו וכך נקבל מטריצה

import numpy as np

:def results\_into\_matrix(results):

*"""*

*A function that makes a matrix based on data from a table*

***:param*** *results: a list of pairs of tuples*

***:return****: a matrix and a solution vector to represent the results numpy objects*

*"""*

A=np.array(np.identity(len(results)))

b=np.array([0.0 for i in range(len(results))])

deegre=len(results)

for i in range(len(results)):

temp\_deegre=deegre-1

for j in range(len(results)):

A[i][j]=results[i][0]\*\*temp\_deegre

temp\_deegre-=1

b[i]=results[i][1]

return [A,b]