# 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 .

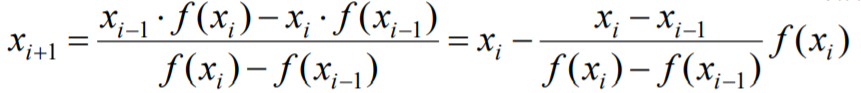
יתרונות וחסרונות: השיטה הכי יעילה באזורים קרובים לשורש. באזורים רחוקים היא עלולה להתבדר. אסור שהנגזרת תהיה שווה לאפס.

.****

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))