# נספח א'

# מסמך אפיון והסבר קודים

הערה:

כל הקוד נכתב על ידי חברי הצוות אלא אם נכתב אחרת, בוצע שימוש בספריות numpy וscipy

# מציאת שורשים

# הגדרת הבעיה

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

**שיטת החציה:**

בהינתן 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  
 it=0  
 while abs(a - b) > epsilon:  
 it+=1  
 if f(m) == 0:  
 print("took {} iterations".format(it))  
 return m  
 elif f(a) \* f(m) < 0:  
 b = m  
 else:  
 a = m  
 m = (a + b) / 2.0  
 print("{} + {} / 2={}".format(a,b,m))  
 print("took {} iterations".format(it))  
 return m

**הקלט-** a,b שתי נקודות שאחת מהן נותנת ערך חיובי בפונקציה והשני הערך שלילי, f זו הפונקציה שעבורה מחשבים את השורשים ואפסילון המורה על רמת הדיוק של האלגוריתם.

**הפלט-** אחד משורשי הפונקציה (אותו השורש שקיים בתחום שלנו).

**בדיקה-**  בדקנו על משוואה ריבועית שאנו יודעים את פתרונה במקרה זה

**X=-1 X=-2**

print(bisection(-1.5,1.5,lambda x: x\*x +3\*x + 2,0.001))

**בדקנו 2 תחומים (-1.5,1.5)**

**ו (1.5,(-5**

וקיבלנו בסופו של דבר את תוצאות הפולינום:

**-1.5 + 0.0 / 2=-0.75**

**-1.5 + -0.75 / 2=-1.125**

**-1.125 + -0.75 / 2=-0.9375**

**-1.125 + -0.9375 / 2=-1.03125**

**-1.03125 + -0.9375 / 2=-0.984375**

**-1.03125 + -0.984375 / 2=-1.0078125**

**-1.0078125 + -0.984375 / 2=-0.99609375**

**-1.0078125 + -0.99609375 / 2=-1.001953125**

**-1.001953125 + -0.99609375 / 2=-0.9990234375**

**-1.001953125 + -0.9990234375 / 2=-1.00048828125**

**-1.00048828125 + -0.9990234375 / 2=-0.999755859375**

**-1.00048828125 + -0.999755859375 / 2=-1.0001220703125**

**took 12 iterations**

**-1.0001220703125**

**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=10, h=0.00000001):  
 *"""* ***:param*** *func: the function to get it's roots* ***:param*** *x: starting guess(default=0)* ***:param*** *it: the number of iterations(default=10)* ***:param*** *h: accuracy factor of the method (default=0.00000001)* ***:return****: An estimated root of the funtion  
 """* print("initial guess: {}".format(x))  
 def der(f, h):  
 print ("f(x0+{0})-f(x0)/{0}".format(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)  
 print ("{})guess={}".format(i+1,x))  
  
 return x

**הקלט-** הפונקציה, ניחוש התחלתי בתור x, מספר איטרציות מקסימלי ודיוק.

**הפלט-** אחד משורשי הפונקציה, תלוי בקירוב שבקלט.

**בדיקה-** בדקנו על שני פולינומים, אחד ממעלה שלישית ואחת ממעלה שנייה , וראינו האם קיבלנו את אחד הפתרונות:

print (newton\_rapson(lambda x: pow(x,3) - 3\*(x\*\*2)+ x +1,3,10,0.00001))

וקיבלנו בסופו של דבר את תוצאת הפולינום באופן תקין (לאחר בדיקת מחשבון):

initial guess: 3

f(x0+1e-05)-f(x0)/1e-05

f(x0+1e-05)-f(x0)/1e-05

1)guess=2.6000023999990773

f(x0+1e-05)-f(x0)/1e-05

2)guess=2.442255494070455

f(x0+1e-05)-f(x0)/1e-05

3)guess=2.4150110244378618

f(x0+1e-05)-f(x0)/1e-05

4)guess=2.414214244450737

f(x0+1e-05)-f(x0)/1e-05

5)guess=2.414213562380823

f(x0+1e-05)-f(x0)/1e-05

6)guess=2.414213562373096

f(x0+1e-05)-f(x0)/1e-05

7)guess=2.4142135623730954

f(x0+1e-05)-f(x0)/1e-05

8)guess=2.414213562373095

f(x0+1e-05)-f(x0)/1e-05

9)guess=2.414213562373095

f(x0+1e-05)-f(x0)/1e-05

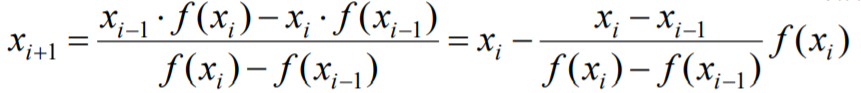
10)guess=2.414213562373095

2.414213562373095

**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 funtion 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]

**הקלט-** הפונקציה שעליה נרצה לבצע את השיטה, שני ניחושים, מספר איטרציות מקסימלי, דיוק.

**הפלט-**אחד השורשים של הפונקציה (תלוי בניחושים), ורמת השגיאה.

**בדיקה-** נבדוק על פונקציה שאנו יודעים את פתרונותיה האנליטים:

print (secant(lambda x: pow(x,3) - 3\*(x\*\*2)+ x +1, -10,10,100,0.0001))

ונקבל הדפסה של השורש וגם של מספר האיטרציות והשגיאה:

1 )Estimated root: 2.9603960396039604 Approximate error: 437.7926421404682

2 )Estimated root: 2.9247191826897483 Approximate error: 1.2198387156404356

3 )Estimated root: 2.5727397131324667 Approximate error: 13.681114640576103

4 )Estimated root: 2.4693817390890995 Approximate error: 4.185581046756007

5 )Estimated root: 2.4220972358754027 Approximate error: 1.9522132519426738

6 )Estimated root: 2.4146520182841456 Approximate error: 0.3083350120382041

7 )Estimated root: 2.414217203686974 Approximate error: 0.018010583161602717

8 )Estimated root: 2.4142135640658804 Approximate error: 0.00015075804177278407

9 )Estimated root: 2.4142135623731016 Approximate error: 7.011719395500813e-08

10 )Estimated root: 2.414213562373096 Approximate error: 2.3913210571048156e-13

11 )Estimated root: 2.4142135623730954 Approximate error: 1.8394777362344737e-14

12 )Estimated root: 2.414213562373094 Approximate error: 5.518433208703424e-14

13 )Estimated root: 2.414213562373095 Approximate error: 3.6789554724689486e-14

14 )Estimated root: 2.414213562373095 Approximate error: 0.0

number of iterations: 14

[2.414213562373095, 0.0]

**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 ייצוג מטריצה כקלאס בפייטון: (קונסטראקטור, str)

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  
 """* self.mat = np.array(A)  
 self.b = np.array(0.0 for i in range(len(A))) if b == 0 else np.array(b)  
  
 def \_\_str\_\_(self):  
 newstr = ''  
 for i in enumerate(self.mat):  
 newstr += str(i[1]).replace(']', '|{}]\n'.format(self.b[i[0]]))  
 return newstr

מייצג מחלקה של מטריצה המשתמשת בNUMPY כדי לעשות מערכים מהצורה NARRAY

הייצוג כולל גם ווקטור תוצאות: b

**חלוקת LU:**

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

לדוגמא (מטריצה 3\*3):

אלכסון ראשי

*חלק תחתון*

*חלק עליון*

קוד שמייצג את החלוקה הזאת:

@property  
def D(self):  
 *"""  
 Defines the diagonal(D) attribute of the matrix  
 """* diag = np.zeros\_like(self.mat)  
 for i in enumerate(self.mat):  
 diag[i[0]][i[0]] = self.mat[i[0]][i[0]]  
 return diag  
  
@property  
def U(self):  
 *"""  
 Defines the upper(U) attribute of the matrix  
 """* upper = np.zeros\_like(self.mat)  
 for i in enumerate(self.mat):  
 for j in enumerate(self.mat):  
 if i[0] < j[0]:  
 upper[i[0]][j[0]] = self.mat[i[0]][j[0]]  
 return upper  
  
@property  
def L(self):  
 *"""  
 Defines the lower(L) attribute of the matrix  
 """* lower = np.zeros\_like(self.mat)  
 for i in enumerate(self.mat):  
 for j in enumerate(self.mat):  
 if i[0] > j[0]:  
 lower[i[0]][j[0]] = self.mat[i[0]][j[0]]  
 return lower

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

de @staticmethod  
def check\_invertible(A):  
 *"""  
 Checks if a matrix is invertible or not* ***:return****: boolean:True if it's invertible or False if it's singular  
 """* try:  
 inv(A)  
 except LinAlgError:  
 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):  
 return max(reduce(lambda x, y: abs(x + y), i) for i in A)  
 return normal(self.mat) \* normal(inv(self.mat)) if Matrix.check\_invertible(self.mat) else "can't be inverted"

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 למטריצה יש פתרון אין עמודת 0ים.

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])  
 elemental\_mat = elemental(index, i, len(A), multiplier)  
 print("elemental matrix:", elemental\_mat)  
 A = elemental\_mat.dot(A)  
 b[index] = b[index] + (multiplier \* float(b[i]))  
 index = index + 1  
 return [A, b]

*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

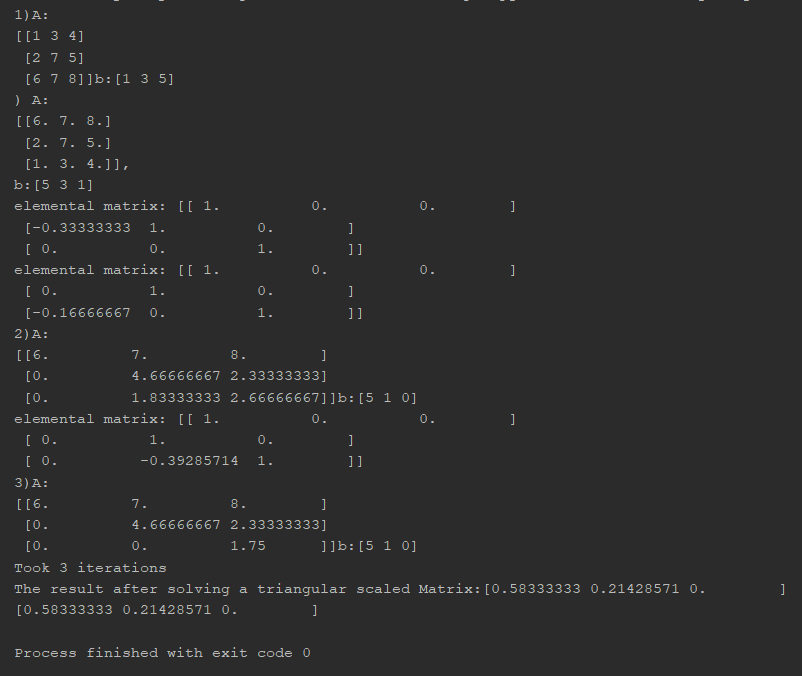
**הקלט-**מטריצה כלשהי, מסדר n\*n ווקטור הפתרונות שלה בגודל n.

**הפלט-**וקטור פתרונות מערכת המשוואות המיוצגות ע"י המטריצה

**בדיקה-** נבדוק על מטריצה שרירותית 3\*3 ווקטור שרירותי, נבדוק את הפתרון בעזרת הצבה

B = Matrix([[1, 3,4], [2, 7,5],[6,7,8]], (1, 3,5))  
print (B.gauss\_elemination())

הבדיקה:

****

הפתרון הוא לא מדויק אבל הוא די קרוב השגיאה בפתרון מדויק יכולה לנבוע מחלוקה וכפל במחשב ובבחירה במטריצה השרירותית הספציפית הזאת (COND)

**שיטות איטרטיביות על מטריצות**

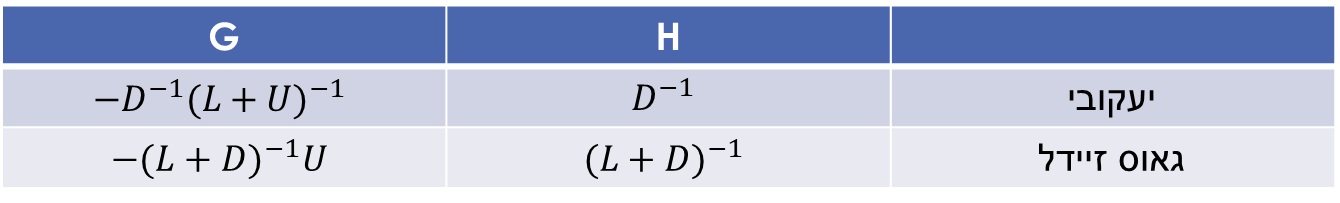
השיטות האיטרטיביות עליהן נדון הן 3 הבאות: גאוס סיידל, יעקובי וSOR

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:*



*יעקובי + זיידל + SOR*

**יעקובי-**

*יעקובי פותר את המערכת המשוואות ע"י שינוי סדר המשתנים במשוואה:*

*כפי שמתואר בטבלה לעיל, לאחר מכן בעזרת פתירת מערכות המשוואות, כאשר לא מציבים את התוצאה שהתקבלה בכל שלב עד לאיטרציה הבאה. להלן הקוד:*

def jacobi(self, guess,max\_itr=1000, tol=0.0001):  
 *"""  
 "Jacobi's method: x[i](r+1)=b- sum((row members except i)\*x[i](r)) i being the row, and r the guess number* ***:param*** *guess: the initial guess for this iterative method* ***:param*** *max\_itr: max number of tolerable iterations (default 1000)* ***:param*** *tol: tolerance of how close the results are* ***:return****: the solution vector of the Matrix this method is used on.  
 """* x = guess  
 n = len(self.mat)  
 for k in range(max\_itr):  
 print("iteration number {}:".format(k + 1))  
 old\_x = x.copy()  
 for i in range(n):  
 sum1 = 0  
 for j in range(n):  
 if (j!=i):  
 sum1 += self.mat[i][j] \* x[j]  
 print("x[{0}] = (1/{1})\*({2} - {3})".format(i + 1, self.mat[i][i], self.b[i], sum1), end="" )  
 x[i] = (1 / self.mat[i][i]) \* (self.b[i] - sum1)  
 print(" = {0}".format(x[i]))  
 if len(list("ok" for index in range(n) if abs(x[index] - old\_x[index]) < tol)) == n:  
 print("took {} iterations".format(k + 1))  
 return x  
 print("The method didn't converge after {} iterations".format(max\_itr))  
 return x

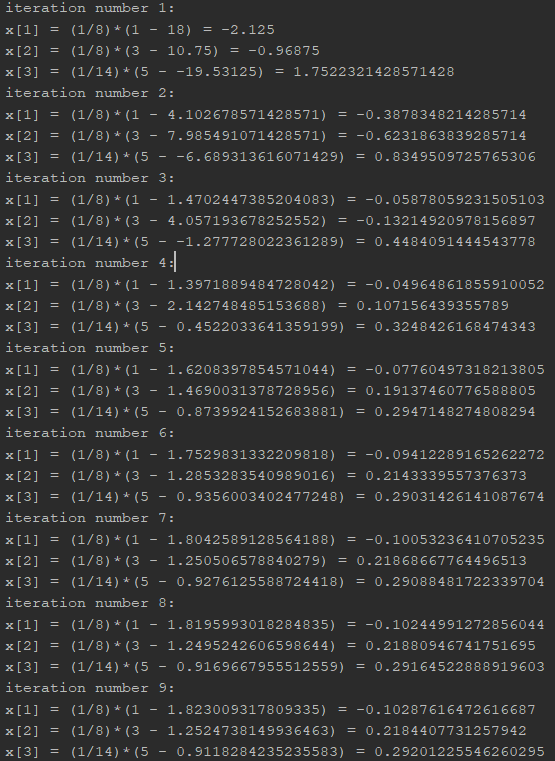
***הקלט-****מטריצה כלשהי ווקטור הפתרונות שלה, ניחוש, מספר מקסימלי של איטרציות ודיוק.*

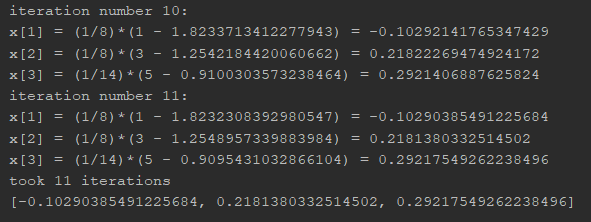
***הפלט-*** *פתרונות המטריצה כוקטור*

***בדיקה-***  *נבדוק על מטריצה שרירותית 3\*3, אך בעלת אלכסון דומיננטי, את הפתרון נבדוק מול פתרון אנליטי*

B = Matrix([[8, 3,4], [2, 8,5],[6,7,14]], (1, 3,5))  
  
print(B.jacobi([1, 2,3]))

*הפלט שלנו יצא תקין:*

**

******

**גאוס זיידל-**

גאוס זיידל *פותר את המערכת המשוואות ע"י שינוי סדר המשתנים במשוואה:*

*כפי שמתואר בטבלה לעיל, לאחר מכן בעזרת פתירת מערכות המשוואות כאשר בכל שלב מציבים את התוצאה של החישוב הקודם, מעדכנים את הניחוש הקודם, ולא מחכים לסוף האיטרציה כמו בשיטת יעקובי.*

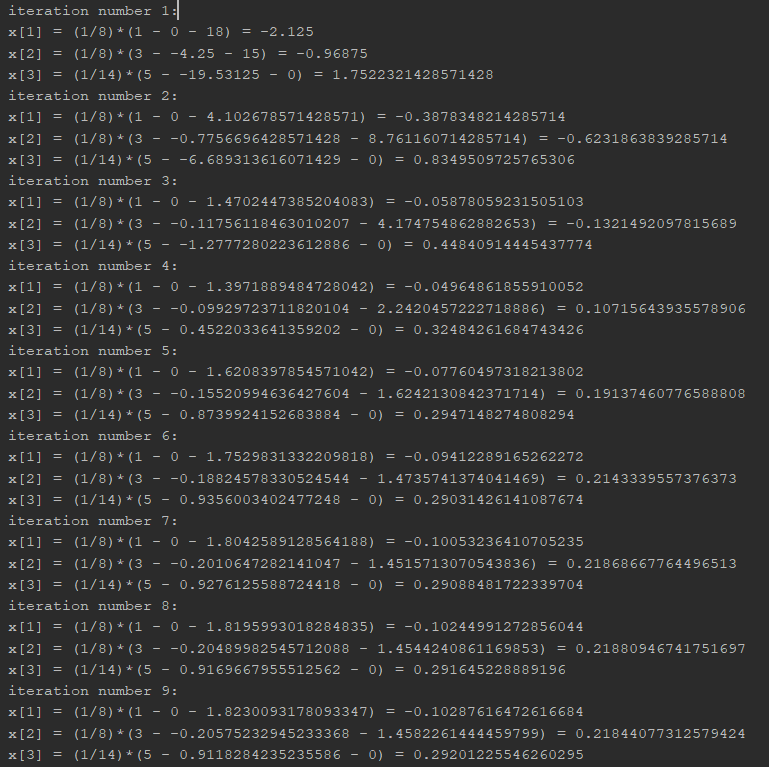
*להלן הקוד:*

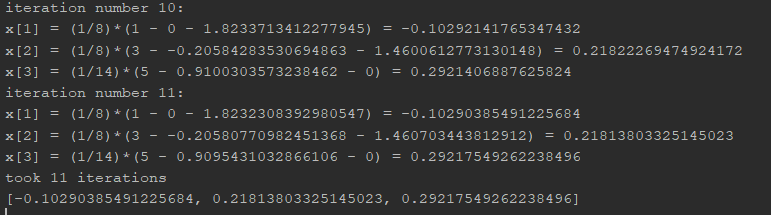
def gauss\_seidel(self, guess,max\_itr=1000,tol=0.0001):  
 *"""  
 Gauss\_seidel's method: x[i](r+1)= b- sum((row members until i)\*x[i](r+1))-sum((row after i)\*x[i](r)  
 i being the row, and r the guess number* ***:param*** *guess: the initial guess for this iterative method* ***:param*** *max\_itr: max number of tolerable iterations (default 1000)* ***:param*** *tol: tolerance of how close the results are* ***:return****: the solution vector of the Matrix this method is used on.  
 """* x = guess  
 n = len(self.mat)  
 for k in range(max\_itr):  
 print("iteration number {}:".format(k+1))  
 old\_x=x.copy()  
 for i in range(n):  
 sum1 = 0  
 for j in range(i):  
 sum1 += self.mat[i][j] \* x[j]  
 sum2 = 0  
 for j in range(n - 1, i, -1):  
 sum2 += self.mat[i][j] \* x[j]  
 print("x[{0}] = (1/{1})\*({2} - {3} - {4})".format(i + 1, self.mat[i][i], self.b[i], sum1, sum2), end="")  
 x[i] = (1 / self.mat[i][i]) \* (self.b[i] - sum1 - sum2)  
 print(" = {0}".format(x[i]))  
 if len(list("ok" for index in range(n) if abs(x[index] - old\_x[index]) < tol)) == n:  
 print("took {} iterations".format(k+1))  
 return x  
 print("The method didn't converge after {} iterations".format(max\_itr))  
 return x

***הקלט-*** *מטריצה כלשהי ווקטור הפתרונות שלה, ניחוש, מספר מקסימלי של איטרציות ודיוק.*

***הפלט-*** *פתרונות המטריצה כוקטור*

***בדיקה-*** *נבדוק על מטריצה שרירותית 3\*3, אך בעלת אלכסון דומיננטי, את הפתרון נבדוק מול פתרון אנליטי,*

**

******

*הבדיקה אכן יצאה טובה, כמו בבדיקה של שיטת יעקובי.*

**SOR-**

שיטה זו היא שיטה איטרטיבית הלוקחת מטריצה ובעזרת קבוע w גורמת להם להתכנס יותר מהר מאשר גאוס זיידל. הקבוע אמור לגרום למטריצה שיהיה לה אלכסון דומיננטי בעזרת הנוסחה:



להלן הקוד המבצע זאת:

def sor(self,w, guess,max\_itr=1000,tol=0.00001):  
 *"""  
 A variation of Gauss Seidel method that uses a w constant in a a way that would make the  
 main diagonal of the matrix a dominant diagonal for faster convention.  
 w should be a number between 0 and 2 (not including)  
 if w is 1 this methoid is essentially the same as gauss seidel method* ***:param*** *w: the constant on which this method is based should be between 0 and 2 not including.* ***:param*** *guess: the initial guess for this iterative method* ***:param*** *max\_itr: max number of tolerable iterations (default 1000)* ***:param*** *tol: tolerance of how close the results are* ***:return****: the solution vector of the Matrix this method is used on.  
 """* if (w<=0 or w>=2):  
 return "w out of range"  
 a=self.D +w\*self.L  
 b=w\*self.b  
 c=w\*self.U + (1-w)\* self.D  
 print ("SOR of this equation list can be represented as:\n{}x={}-{}x".format(a,b,c))  
 x = guess  
 n = len(self.mat)  
 for k in range(max\_itr):  
 print("iteration number {}:".format(k + 1))  
 old\_x = x.copy()  
 for i in range(n):  
 sum1 = 0  
 for j in range(i):  
 sum1 += self.mat[i][j] \* x[j]  
 sum2 = 0  
 for j in range(n - 1, i, -1):  
 sum2 += self.mat[i][j] \* x[j]  
 print("x[{0}] = (1-{5})\*x[{0}] ({5}/{1})\*({2} - {3} - {4})".format(i + 1, self.mat[i][i], self.b[i], sum1, sum2,w), end="")  
 x[i] = (1-w)\*x[i] + (w / self.mat[i][i]) \* (self.b[i] - sum1 - sum2)  
 print(" = {0}".format(x[i]))  
 if len(list("ok" for index in range(n) if abs(x[index] - old\_x[index])<tol))==n:  
 print("took {} iterations".format(k + 1))  
 return x  
 print("The method didn't converge after {} iterations".format(max\_itr))  
 return x

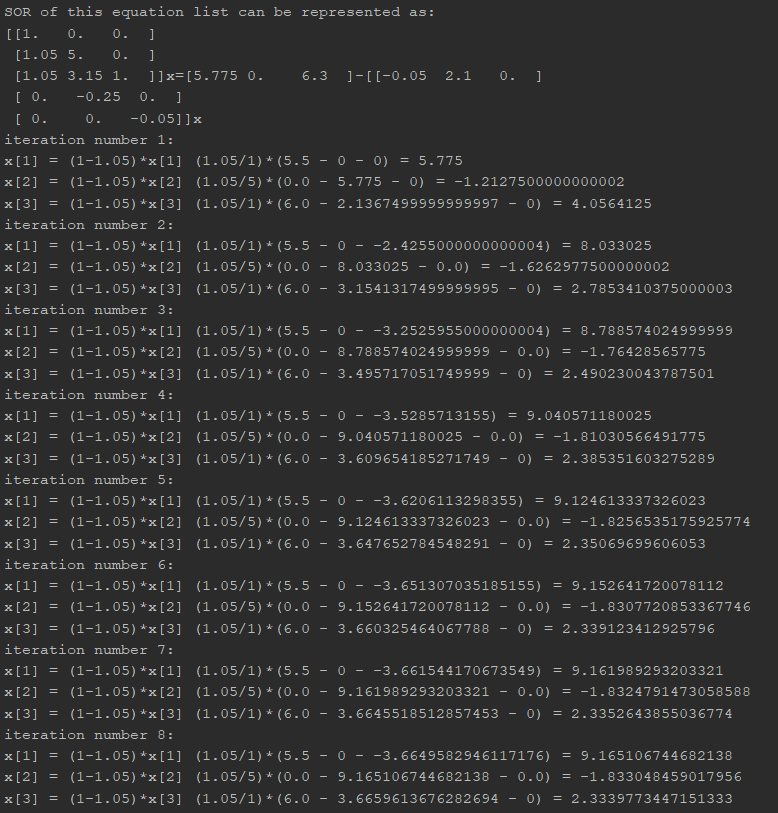
***הקלט-*** *מטריצה כלשהי ווקטור הפתרונות שלה, ניחוש, מספר מקסימלי של איטרציות ודיוק.*

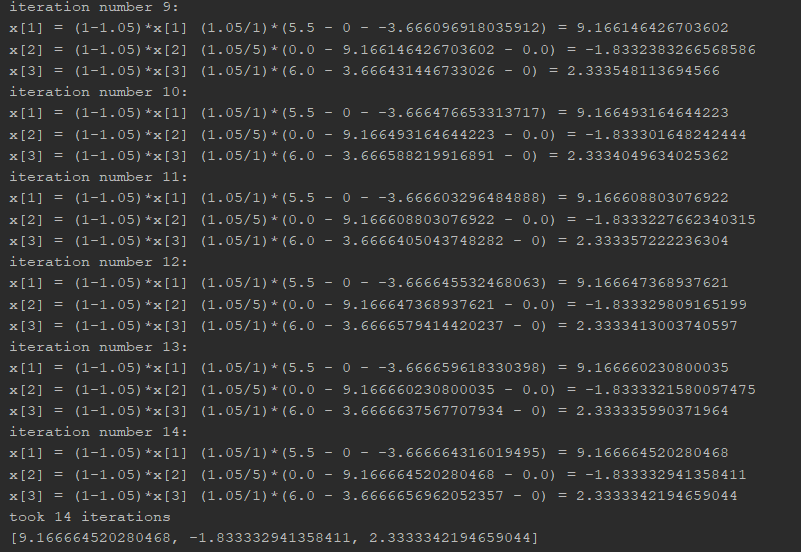
***הפלט-*** *ווקטור הפתרונות*

***הבדיקה-*** *נבצע בדיקה על מטריצה שרירותית שאין לה אלכסון דומיננטי:*

B = Matrix([[1, 2,0], [1,5,0],[1,3,1]], (5.5, 0,6))  
print(B.sor(1.05,[0,0,0],100))

*והתקבל פתרון כראוי-*

**

******

***אינטרפולציות***

**אינטרפולציה פולינומית:**

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, ווקטור התוצאות (הyים של הטבלה).

הקוד הבא מקבל מערך של זוגות של הצבות ותוצאות מהניסוי.

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  
 print("a{2}\*({0})^{1}".format(results[i][0],str(temp\_deegre),j+1), end=" ")  
 temp\_deegre-=1  
 print("\n")  
 b[i]=results[i][1]  
 return [A,b]

**הקלט-** מספר נקודות טבלה המיוצגות x,y)) עבור תוצאה כלשהי של ניסוי שערכנו וקיבלנו נקודות טבלה.

**הפלט-** מטריצת מקדמים ווקטור פתרונות המייצגים פולינומים במעלה n-1 כאשר n-1 מייצג את מספר הנקודות.

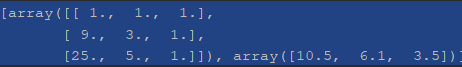
לאחר שנפתור את מטריצת המקדמים עם וקטור הפתרונות נקבל פולינום ממעלה n-1 המתארת בקירוב את הנקודות שקיבלנו.

**בדיקת הקוד-**

בוצעה ההדפסה הבאה:

print (results\_into\_matrix([(1,10.5),(3,6.1),(5,3.5)]))

שהביאה לנו את התוצאות:



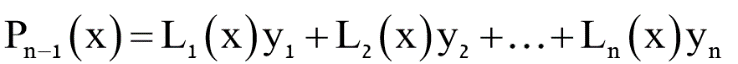
הפונקציה מציבה את ערכי הX שבטבלה ומעלה בחזקות הפולינום, ניתן לראות כי הקוד תקין ועובד על פי הנקודות שהובאו בדוגמא זאת.

**שיטת אינטרפולצית לגראנז':**

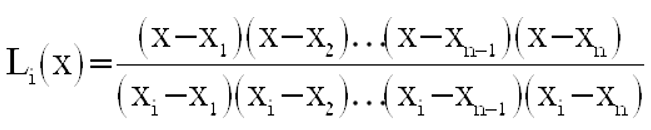
ניתן לבטא את פולינום האינטרפולציה (**שהוא, כזכור, יחיד**) באופן הבא:

 כאשר .

כאשר ניתן לבטא את P(x) כך:



כאשר ניתן לבטא את L(x) כך:



**יתרון של השיטה:** לכל רשת של נקודות הדגימה מחשבים פעם אחת ולתמיד את הפונקציות  וכל מה שצריך לעשות כדי לקרב פונקציה חדשה ברשת ישנה הוא להכפיל את ערכי הפונקציות הידועות בערכים של הפונקציה המקורבת ולחבר.

**חיסרון של השיטה:** אם רוצים להוסיף עוד נקודת דגימה אחת, צריך לחשב את הכל מהתחלה.

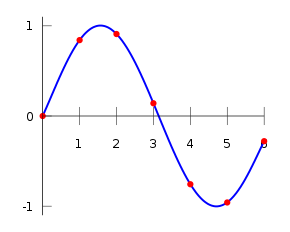
**דוגמה:** פונקציה  מוגדרת ע"י טבלה. מצא פולינום אינטרפולציה.

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 1 | 0 |  |
| 7 | 3 | 1 |  |

**פתרון:** נבנה פולינום ממעלה 2 שעובר דרך 3 נקודות:

.

צורת כתיבת השיטה הזו מאפשרת חישוב ערכו לכל X בלי לדעת מהם המקדמים a1,a2,…an המופיעים בניסוח המקובל של p(x).



הפולינום שהוגדר בצורת לגראנז' אכן עובר דרך הנקודות הנתונות.

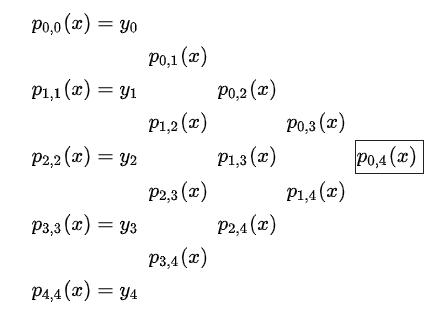
**from** functools **import** reduce  
  
**def** lagrangian\_interpolate(samples):  
 *"""  
 Takes some samples as a list of tuples and returns a function that's  
 a lagrangian interpolation of all the samples.  
 """* X = 0 *# the tuple index of the X variable in the samples* Y = 1 *# the tuple index of the Y variable in the samples* n = len(samples)  
 *# define the L function as a function generator that generates L functions  
 # for a given i* **def** L(i):  
 *"This function generates an L function for a given x\_i"* **def** L\_gen(x):  
 ret = []  
 **for** j **in** range(n):  
 **if** j != i:  
 *#print("({0}-{1})/({2}-{1})".format(x,samples[j][X], samples[i][X]), end=" ")* ret.append((x - samples[j][X])/(samples[i][X] - samples[j][X]))  
 **return** reduce(**lambda** a,b: a\*b, ret)  
 **return** L\_gen  
 **return lambda** x: sum(L(i)(x) \* samples[i][Y] **for** i **in** range(n))  
  
prob\_1 = lagrangian\_interpolate([(2,1.4142),(2.5,1.5811),(3.0,1.7321)])  
print(prob\_1(2.2))  
  
prob\_1\_b = lagrangian\_interpolate([(2,1.4142),(2.5,1.5811),(2.7,1.6432)])  
print(prob\_1\_b(2.2))  
  
prob\_2 = lagrangian\_interpolate([(2.0,1.4142),(2.5,1.5811),(3.0,1.7321),(3.5,1.8708)])  
print(prob\_2(2.8))

**שיטת נוויל:**

בשיטת נוויל נבנה פולינום בשלבים ובכל פעם נעשה חישובים על מספרים בסדרי גודל דומים,

הנוסחה שעליה מסתמכת שיטת נוויל:



אפשר לתאר את שיטת נוויל בעץ כדוגמא: 

def nevilleInter(points, x):  
 *"""* ***:param*** *points: a list of points ie ((0,1),(1,2)) etc..* ***:param*** *x: the x in question of what the value is at this x* ***:return****: the value y at the given x  
 """* if len(points) < 2:  
 return "Invalid points"  
  
 p = [[0 for j in range(len(points))] for y in range(len(points))]  
 for i in range(len(points)):  
 p[i][i] = points[i][1]  
 if i + 1 < len(points):  
 if points[i + 1][0] <= points[i][0]:  
 return "Invalid points"  
 i = 0  
 j = 1  
 t = j  
 while True:  
 print("P{0},{1}=(({2} - x{1}) \* p{0}{3} - ({2} -x{0}) \* p{4}{1}/ (x{0} - x{1})=".format(i,j,x,j-1,i+1),end="")  
 p[i][j] = ((x - points[j][0]) \* p[i][j - 1] - (x - points[i][0]) \* p[i + 1][j]) / (  
 points[i][0] - points[j][0])  
 print(p[i][j])  
 if i == 0 and j == len(points) - 1:  
 break  
 if j == len(points) - 1:  
 i = 0  
 j = t + 1  
 t = j  
 else:  
 i += 1  
 j += 1  
 return p[0][-1]

**קלט-** נקודות וערך x אותו נרצה לחשב עבור אותן נקודות.

**פלט-** הערך y של אותו הx אותו אנו רוצים לחשב

**בדיקה-**נבדוק עבור נקודות שרירותיות אם ערכים שרירותיים:

print(nevilleInter(((8.1, 16.9446), (8.3, 17.56492), (8.6, 18.50515), (8.7, 18.82091)),8.5))

הבדיקה אכן יצאה טובה:

P0,1=((8.5 - x1) \* p00 - (8.5 -x0) \* p11/ (x0 - x1)=18.185239999999997

P1,2=((8.5 - x2) \* p11 - (8.5 -x1) \* p22/ (x1 - x2)=18.19174

P2,3=((8.5 - x3) \* p22 - (8.5 -x2) \* p33/ (x2 - x3)=18.18939

P0,2=((8.5 - x2) \* p01 - (8.5 -x0) \* p12/ (x0 - x2)=18.19044

P1,3=((8.5 - x3) \* p12 - (8.5 -x1) \* p23/ (x1 - x3)=18.190565

P0,3=((8.5 - x3) \* p02 - (8.5 -x0) \* p13/ (x0 - x3)=18.190523333333335

18.190523333333335

**אינטרפולציה לפני שיטת ניוטון:**

רושמים את הפולינום בצורה:

 ומטרתנו היא לקבוע את המקדמים *Ai* ובזה נפתור את בעיית האינטרפולציה.

**שאלה:** איך ייתכן שבפולינום הרשום בצורה זו לא מופיעה נקודה *xN*? האם זה אומר שהפולינום אינו תלוי בבחירת נקודה זו? **תשובה:** בודאי שהמקדמים של הפולינום המתקבל תלויים בכל נקודות האינטרפולציה, כולל *xN*, אבל התלות הזאת מתבטאת במקדמים *Ai* .

### שיטת הפרשים מחולקים

זוהי דרך נוחה (וכמו שנראה בהמשך, קצת מעבר לזה) לבניית מקדמים עבור אינטרפולצית Newton. מגדירים הפרש מחולק כפונקציה של נק' דגימה אחת או רצף של נקודות באופן הבא:

עבור נקודת דגימה אחת ; עבור 2 נקודות , וכו'. באופן כללי עבור *k*+1 נקודות: , כלומר, הפרש מחולק של סיפא פחות הפרש מחולק של רישא חלקי נקודה אחרונה פחות נקודה ראשונה. אפשר להוכיח ש-

א' 

ב'  עבור איזשהו .

ניתן לבנות טבלה של הפרשים מחולקים. למשל, עבור 4 נק' זה נראה כך:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

כאן כל משבצת מחושבת לפי שכנותיה השמאלית והשמאלית-עליונה. המקדמים של הפולינום רשומים באלכסון הטבלה.

### תנאי על הנגזרת

**הבעיה:** יש להרחיב את בעיית האינטרפולציה באופן הבא: ברצוננו לבנות פולינום שגם יעבור דרך הנקודות הנתונות וגם הנגזרת שלו בנקודה  תהיה שווה לערך נתון מראש.

**הפתרון:** בשיטת הפרשים מחולקים ניתן לעשות זאת באופן הבא: רושמים את הנקודה  פעמיים ברציפות (בשתי שורות עוקבות): . במקרה זה  אינו מוגדר באופן רגיל (כי בהגדרה רגילה גם מונה וגם מכנה שווים לאפס) וקובעים .

**הערה:** ניתן להמשיך ולקבוע בדרך זו נגזרות מסדר גבוה יותר עפ"י תכונה ב' של הפרשים מחולקים. כדי לקבוע נגזרת מסדר *k* יש לקבוע גם נגזרות מסדרים נמוכים יותר.

**שגיאת האינטרפולציה:** איבר השגיאה אינו משתנה, רק היות ש-  ניתן לרשום את המכפלה בצורה קצרה יותר: במקום  נכתוב 

**וכל זה מדוע?** לא נביא את ההוכחה הפורמלית של השיטה אלא נסמן את הכוון שלה. ההוכחה היא פשוטה מאד ומבוססת על תכונה ב' של הפרש מחולק. אם נגדיר , נקבל פולינום אינטרפולציה רגיל ואם נשאיף את *ε* לאפס נראה שערך הפונקציה ב-  וגם .

**שם:** פולינום אינטרפולציה שנגזרתו בכל נקודות הדגימה מקבלת ערכים נתונים מראש (או, במילים אחרות, פולינום שמתלכד עם הפונקציה בנקודות הדגימה כך שגם נגזרותיהם מתלכדות) נקרא פולינום אינטרפולציה של Hermite.

### דוגמאות

**דוגמה 1:** פונקציה  מוגדרת ע"י טבלה. מצא פולינום אינטרפולציה.

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 1 | 0 |  |
| 7 | 3 | 1 |  |

**פתרון:** נבנה פולינום ממעלה 2 שעובר דרך 3 נקודות. בדרך ישירה מקבלים:



ואם נבנה טבלת הפרשים מחולקים נקבל

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  | 1 | 0 |
|  |  | 3 | 1 |
|  |  | 7 | 2 |

קיבלנו אותם מקדמים כמו בשיטה ישירה ואם נציב אותם בפולינום נקבל . קיבלנו פולינום זהה לזה שבנינו בשיטת Lagrange. האם זה מוזר? בכלל לא! הרי קיים פולינום יחיד ממעלה 2 שעובר דרך 3 נקודות נתונות, אז אין זה חשוב באיזו שיטה אנו בונים אותו.

**def** newton\_interpolation(x, y, u):

*'''*

*Parameters*

*----------*

*x : list of floats*

*y : list of floats*

*u : float*

*Returns*

*-------*

*float*

*an estimate at the point u*

*'''*

g = y[:]

s = g[0]

**for** i **in** range(len(y) - 1):

g = [(g[j + 1] - g[j]) / (x[j + i + 1] - x[j]) **for** j **in** range(len(g) - 1)]

s += g[0] \* product(u - x[j] **for** j **in** range(i + 1))

**return** s

**def** product(a):

p = 1

**for** i **in** a: p \*= i

**return** p

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

**from** math **import** \*

x = [1, 2.5, 4, 5.5, 7]

y = [log(i) **for** i **in** x]

u = 6

estim = newton\_interpolation(x, y, u)

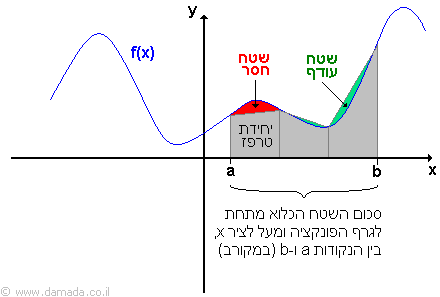
exact = log(u)

print(estim, exact)

***חישוב אינטגרלים***

***שיטת הטרפז-***

בחישוב שטחו של העיגול שהובא בתחילת הפרק חולק שטחו של העיגול לאינספור חלקים. לולא חולק העיגול לאינספור חלקים לא ניתן היה לקרב כל חלק למשולש שגובהו כרדיוס העיגול.  
  
למרות זאת, בסופו של תהליך החישוב התקבלה נוסחה מדויקת לחישוב שטחו של העיגול.  
  
העיגול הנו צורה גיאומטרית סימטרית ופשוטה עבור צורות אחרות נצטרך לבצע חישוב מקורב בעזרת שיטת הטרפז.  
  
שיטת הטרפז היא שיטת חישוב הנותנת קירוב לערכו של שטח כלוא או לערכו של אינטגרל מסוים על פונקציה בין שני ערכים ללא ביצוע פעולת האינטגרל כלל.



**ציור של שטח מתחת לגרף המחולק לטרפזים**

בשיטה זו נחלק את השטח הכלוא או את השטח המוגדר בין הפונקציה לציר x בין שתי נקודות על ציר a x ו- b, לצורות טרפז. כל הטרפזים בעלי רוחב זהה המונח על ציר x , רוחב כל טרפז מתקבל מחלוקת כל המרחק על ציר b-a, x במספר הטרפזים.

נקבל שרוחבו של כל טרפז הוא:

(b-a)/n

השטח, המסומן על-ידי s, של כל טרפז מתקבל מהנוסחה הבאה,

s = (b-a)/n • (yn + yn+1)/2

כאשר yn ו- yn+1 הן שתי נקודות עוקבות מתוך n+1 נקודות שעל הגרף.  
  
נשים לב שכדי לקבל n טרפזים יש להגדיר n+1 נקודות על הגרף!

השטח הכולל, המסומן על-ידי S, מחושב בעזרת סכום כל שטחי הטרפזים.

נקבל,

(b-a)/n • [(y1 + y2)/2 + (y3 + y4)/2 + … + (yn + yn+1)/2] = S

השטח הכולל הוא ערכו של האינטגרל על הפונקציה בין שתי הנקודות a ו- b. לכן נקבל,

b                                                                                      
∫f(x)dx ≈ (b-a)/n • [(y1 + yn+1)/2 + y2 + y3 + … + yn]  
a

def trapezium(f=lambda x:x,a=0,b=1,n=1000):  
 *"""  
 Calculates and prints the process of trapezium method for calculating an integral* ***:param*** *f: the function to calculate its' integral* ***:param*** *a: the initial value to calculate the integral from* ***:param*** *b: the end value of the integral* ***:param*** *n: the amount of parts to calculate the integral* ***:return****: the calculation of the integral  
 """* def delta\_x(a, b, n):  
 return (b - a) / n  
 integral=f(a)  
 h=delta\_x(a,b,n)  
 print("h={}".format(h))  
 print('h/2 \* ({}'.format(integral), end="")  
 multiplier=2  
 for i in range(1,n-1):  
 fxi= multiplier\*f(a+i\*h)  
 print (" + {}\*f({}) ".format(multiplier,a+i\*h),end="")  
 integral+=fxi  
  
 print("+ {})".format(f(b)))  
 return (h/2) \* (integral + f(b))

**הקלט-** הפונקציה, a וb שהם הגבולות ומספר הטרפזים שנחלק את האינטגרל.

**הפלט-**האינטגרל של הפונקציה

**בדיקה-** נזין פונקציה שאנו יודעים את האינטגרל המסוים שלה ונבדוק האם מתקבל אינטגרל תקין:

print(trapezium(lambda x:x\*\*3 + e(x),0,1,10000))

וקיבלנו-

h=0.01

h/2 \* (1.0 + 2\*f(0.01) + 2\*f(0.02) + 2\*f(0.03) + 2\*f(0.04) + 2\*f(0.05) + 2\*f(0.06) + 2\*f(0.07) + 2\*f(0.08) + 2\*f(0.09) + 2\*f(0.1) + 2\*f(0.11) + 2\*f(0.12) + 2\*f(0.13) + 2\*f(0.14) + 2\*f(0.15) + 2\*f(0.16) + 2\*f(0.17) + 2\*f(0.18) + 2\*f(0.19) + 2\*f(0.2) + 2\*f(0.21) + 2\*f(0.22) + 2\*f(0.23) + 2\*f(0.24) + 2\*f(0.25) + 2\*f(0.26) + 2\*f(0.27) + 2\*f(0.28) + 2\*f(0.29) + 2\*f(0.3) + 2\*f(0.31) + 2\*f(0.32) + 2\*f(0.33) + 2\*f(0.34) + 2\*f(0.35000000000000003) + 2\*f(0.36) + 2\*f(0.37) + 2\*f(0.38) + 2\*f(0.39) + 2\*f(0.4) + 2\*f(0.41000000000000003) + 2\*f(0.42) + 2\*f(0.43) + 2\*f(0.44) + 2\*f(0.45) + 2\*f(0.46) + 2\*f(0.47000000000000003) + 2\*f(0.48) + 2\*f(0.49) + 2\*f(0.5) + 2\*f(0.51) + 2\*f(0.52) + 2\*f(0.53) + 2\*f(0.54) + 2\*f(0.55) + 2\*f(0.56) + 2\*f(0.5700000000000001) + 2\*f(0.58) + 2\*f(0.59) + 2\*f(0.6) + 2\*f(0.61) + 2\*f(0.62) + 2\*f(0.63) + 2\*f(0.64) + 2\*f(0.65) + 2\*f(0.66) + 2\*f(0.67) + 2\*f(0.68) + 2\*f(0.6900000000000001) + 2\*f(0.7000000000000001) + 2\*f(0.71) + 2\*f(0.72) + 2\*f(0.73) + 2\*f(0.74) + 2\*f(0.75) + 2\*f(0.76) + 2\*f(0.77) + 2\*f(0.78) + 2\*f(0.79) + 2\*f(0.8) + 2\*f(0.81) + 2\*f(0.8200000000000001) + 2\*f(0.8300000000000001) + 2\*f(0.84) + 2\*f(0.85) + 2\*f(0.86) + 2\*f(0.87) + 2\*f(0.88) + 2\*f(0.89) + 2\*f(0.9) + 2\*f(0.91) + 2\*f(0.92) + 2\*f(0.93) + 2\*f(0.9400000000000001) + 2\*f(0.9500000000000001) + 2\*f(0.96) + 2\*f(0.97) + 2\*f(0.98) + 3.718281828459045)

**1.931705812726924**

***שיטת השליש של סימפסון-***

שיטת סימפסון מחלקת כל קטע ל3 קטעים שווים בכל פעם בשונה מהשתי קטעים ששיטת הטרפז מחלקת.

שיטת סימפסון עם n קטעים שווים (n=2m זוגי)



השיטה יותר מדויקת משיטת הטרפז משום שהשגיאה בה יותר נמוכה בדרך כלל.

שגיאה בשיטת סימפסון:



כאשר a וb הם תחומי האינטגרציה וc היא נקודה בין b לa שאותה אנו לא יכולים לחשב, שיטת סימפסון מדויקת עד פולינומים ממעלה שלישית.

def simsoms\_one\_third(f=lambda x:x,a=-1,b=1,n=1000):  
 *"""  
 Calculates and prints the process of Simpson's one-third method for calculating an integral* ***:param*** *f: the function to calculate its' integral* ***:param*** *a: the initial value to calculate the integral from* ***:param*** *b: the end value of the integral* ***:param*** *n: the amount of parts to calculate the integral* ***:return****: the calculation of the integral  
 """* def delta\_x(a, b, n):  
 return (b - a) / n  
 integral=f(a)  
 h=delta\_x(a,b,n)  
 print("h={}".format(h))  
 print('h/3 \* ({}'.format(integral), end="")  
 multiplier=2  
 for i in range(1,n-1):  
 fxi= multiplier\*f(a+i\*h)  
 print (" + {}\*f({}) ".format(multiplier,a+i\*h),end="")  
 integral+=fxi  
 multiplier= 4 if multiplier==2 else 2  
 print("+ {})".format(f(b)))  
 return (h/3)\*(integral + f(b))

**הקלט-** הפונקציה עליה אנו רוצים לבצע אינטגרל, a וb הגבולות, ומספר חלוקות שוות.

**הפלט-** האינטגרל

**בדיקה-** ניקח פונקציה שאנו רוצים לחשב עליה את האינטגרל, נחשב את האינטגרל בעזרת מחשבון אינטגרלים ונבדוק האם הפלט שלנו תקין:

print(simsoms\_one\_third(lambda x: x\*\*3 + e(x),1,3,100))

וקיבלנו אכן תשובה קרובה מאוד לפתרון המחשבון:

המחשבון: 37.3672550947

הפלט:

h=0.02

h/3 \* (3.718281828459045 + 2\*f(1.02) + 4\*f(1.04) + 2\*f(1.06) + 4\*f(1.08) + 2\*f(1.1) + 4\*f(1.12) + 2\*f(1.1400000000000001) + 4\*f(1.16) + 2\*f(1.18) + 4\*f(1.2) + 2\*f(1.22) + 4\*f(1.24) + 2\*f(1.26) + 4\*f(1.28) + 2\*f(1.3) + 4\*f(1.32) + 2\*f(1.34) + 4\*f(1.3599999999999999) + 2\*f(1.38) + 4\*f(1.4) + 2\*f(1.42) + 4\*f(1.44) + 2\*f(1.46) + 4\*f(1.48) + 2\*f(1.5) + 4\*f(1.52) + 2\*f(1.54) + 4\*f(1.56) + 2\*f(1.58) + 4\*f(1.6) + 2\*f(1.62) + 4\*f(1.6400000000000001) + 2\*f(1.6600000000000001) + 4\*f(1.6800000000000002) + 2\*f(1.7000000000000002) + 4\*f(1.72) + 2\*f(1.74) + 4\*f(1.76) + 2\*f(1.78) + 4\*f(1.8) + 2\*f(1.82) + 4\*f(1.8399999999999999) + 2\*f(1.8599999999999999) + 4\*f(1.88) + 2\*f(1.9) + 4\*f(1.92) + 2\*f(1.94) + 4\*f(1.96) + 2\*f(1.98) + 4\*f(2.0) + 2\*f(2.02) + 4\*f(2.04) + 2\*f(2.06) + 4\*f(2.08) + 2\*f(2.1) + 4\*f(2.12) + 2\*f(2.14) + 4\*f(2.16) + 2\*f(2.1799999999999997) + 4\*f(2.2) + 2\*f(2.2199999999999998) + 4\*f(2.24) + 2\*f(2.26) + 4\*f(2.2800000000000002) + 2\*f(2.3) + 4\*f(2.3200000000000003) + 2\*f(2.34) + 4\*f(2.3600000000000003) + 2\*f(2.38) + 4\*f(2.4000000000000004) + 2\*f(2.42) + 4\*f(2.44) + 2\*f(2.46) + 4\*f(2.48) + 2\*f(2.5) + 4\*f(2.52) + 2\*f(2.54) + 4\*f(2.56) + 2\*f(2.58) + 4\*f(2.6) + 2\*f(2.62) + 4\*f(2.64) + 2\*f(2.66) + 4\*f(2.6799999999999997) + 2\*f(2.7) + 4\*f(2.7199999999999998) + 2\*f(2.74) + 4\*f(2.76) + 2\*f(2.7800000000000002) + 4\*f(2.8) + 2\*f(2.8200000000000003) + 4\*f(2.84) + 2\*f(2.8600000000000003) + 4\*f(2.88) + 2\*f(2.9000000000000004) + 4\*f(2.92) + 2\*f(2.94) + 4\*f(2.96) + 47.08553692318767)

36.41596864830495

***שיטת תרבועי גאוס-***

שיטת תרבועי גאוס היא שיטת קירוב של אינטגרל מסוים של פונקציה. בשיטה זו ניקח n נקודות לצורך חישוב תוצאה מדויקת לאינטגרל של פולינומים ממעלה 2n-1 או פחות. באמצעות בחירה מתאימה של נקודות דגימה Xi ומשקלים Ci .

עבור i שרץ מ1 עד לn, תחום האינטגרציה בכלל זה הוא בקטע של -1 עד 1.

from numpy.polynomial.legendre import leggauss as compute\_quadrate  
  
def gauss\_quadrature(f,n,b=1,a=-1):  
 *"""  
 Calculates an integral of a function using gauss\_quadrature method.* ***:param*** *f: function to calculate it's integral* ***:param*** *n: Number of sample points and weights. It must be >= 1. it's not accurate after 100.* ***:param*** *b: higher integral limit(default 1)* ***:param*** *a: lower integral limit (default -1)* ***:return****: an estimate of the integral using Gauss\_quadrature method  
 """* print("calculating:integral in [{},{}}] for the given function")  
 sample\_points ,weights= compute\_quadrate(n)  
 print ("sample points:{} weights:{}".format(str(sample\_points),str(weights)))  
 sample\_points=list(((b-a)\*x/2) + ((a+b)/2) for x in sample\_points) #normalize the sample points to the range of 1 and -1  
 print("sample points after normalizing them to the range of 1,-1(the 'ci's in the formula):{}".format(str(sample\_points)))  
 print("({}-{})/2 \* (".format(b,a), end="")  
 for c,xi in zip(weights,sample\_points):  
 print("{} \* f({}) + ".format(c, xi), end="") if xi!=sample\_points[-1] else print("{} \* f({}))".format(c, xi))  
 return ((b-a)/2)\*sum((c\*f(xi) for c,xi in zip(weights,sample\_points))) # The calculation of the integral.

**הקלט-** הפונקציה שנרצה למצוא לה אינטגרל, נקודות ומשקלים וגבולות a וb שהם בברירת מחדל כפי שמצוין בקוד לעמלה.

**הפלט-**האינטגרל

**בדיקה-** ניקח פונקציה שאנו רוצים לחשב עליה את האינטגרל, נחשב את האינטגרל בעזרת מחשבון אינטגרלים ונבדוק האם הפלט שלנו תקין:

print (gauss\_quadrature(lambda x:x\*\*2 - x,5, 10, -10))

ואכן יצא פלט תקין:

מחשבון- 666.666666667

הפלט-

calculating:integral in [{},{}}] for the given function

sample points:[-0.90617985 -0.53846931 0. 0.53846931 0.90617985] weights:[0.23692689 0.47862867 0.56888889 0.47862867 0.23692689]

sample points after normalizing them to the range of 1,-1(the 'ci's in the formula):[-9.06179845938664, -5.384693101056831, 0.0, 5.384693101056831, 9.06179845938664]

(10--10)/2 \* (0.23692688505618942 \* f(-9.06179845938664) + 0.4786286704993662 \* f(-5.384693101056831) + 0.568888888888889 \* f(0.0) + 0.4786286704993662 \* f(5.384693101056831) + 0.23692688505618942 \* f(9.06179845938664))

**666.6666666666672**

***שיטת רומברג-***

שיטת Romberg היא אקסטרפולציה Richardson שמופעלת על שיטת הטרפז. חלוקת הקטע האופטימאלית במקרה זה (הערך q) היא 2, כלומר q=2.

הצורך היחיד באקסטרפולציה הוא חסכון במספר חישובי פונקציה (כי באינטגרציה ההתבטלות אינה קיימת). אם ניקח , כדי לחשב אינטגרל עם מרווח פי שתיים יותר קטן בין הנקודות, נצטרך לקחת נקודה חדשה בין כל שתי נקודות שכבר לקחנו. כלומר, החלק הכבד של החישוב החדש שהוא חישוב הסכום  אנו עושים רק למחצית כי את הסכום בנקודות הזוגיות כבר מחושב ונשארו רק נקודות אי-זוגיות.

import numpy as np  
from functools import reduce  
  
def romberg(f=lambda x:x,a=0,b=1,max\_steps=3):  
 *"""  
 Prints and executes romberg's method of calculating an integral* ***:param*** *f: the function to calculate it's integral* ***:param*** *a: lower limit of the integral* ***:param*** *b: higher limit of the integral* ***:param*** *max\_steps: maximum number of steps* ***:return****: the integral value.  
 """* def h(n):  
 return (b-a)/(2\*\*n)  
 def sum\_of\_f(hi,i):  
 return reduce(lambda x,y:x+y,(f(a + hi\*(2\*k-1)) for k in range(1,(2\*\*(i-1))+1)))  
  
 R=[[h(1)\*(f(a)+f(b))]]  
 i,j=1,0  
 while max\_steps>0:  
 if j==0:  
 hi = h(i)  
 R.append([0.5\*R[i-1][0] + hi\*(sum\_of\_f(hi,i))])  
 j += 1  
 while (j<=i):  
 R[i].append(((1/(pow(4,j)-1))\*(pow(4,j)\*R[i][j-1] - R[i-1][j-1])))  
 j+=1  
 if R[i][j-1]==R[i][j-2]:  
 print(str(R).replace('],', '\n').replace('[[', ' ').replace('[', '').replace(']', ''))  
 return R[i][j-1]  
 j=0  
 i+=1  
 max\_steps-=1  
 print(str(R).replace('],','\n').replace('[[',' ').replace('[','').replace(']',''))  
 return R[i-1][j-1]

**קלט-** הפונקציה עליה נרצה לבצע אינטגרל, הגבולות a וb ומספר איטרציות מקסימלי.

**פלט-** האינטגרל

**בדיקה-** ניקח פונקציה שאנו רוצים לחשב עליה את האינטגרל, נחשב את האינטגרל בעזרת מחשבון אינטגרלים ונבדוק האם הפלט שלנו תקין:

print(romberg(lambda x: x\*\*3 + 3\*x, 0,1, 5))

ואכן יצא פלט תקין-

מחשבון: 1.75

הפלט:

2.0

1.8125, 1.75

1.765625, 1.75, 1.75

**1.75**