**קודים להגשה עבור האקטון מספר 2 בקורס אנליזה נומרית**

**מרצה:** פרופ' שלמה מארק

**חברי הצוות (קבוצה 24):**

בנימין יעקובי, דניס סמוילוב, זוהר כחלון, סמי סליבה, אלכסנדר בוריסנקו.

**תנאים מקדימים להרצת הקודים:**

1. כלל הקודים רשומים בשפת Python3.x ועל כן מחייבים עבודה בסביבה תומכת (PyCharm).
2. יש לייבא את הספריות הבאות: Numpy, Sympy, Scipy.

**המסמך מכיל את התוכן הבא:**

שיטת גאוס-זיידל...............................................................................................................2

שיטת יעקובי......................................................................................................................3

אינטרפולציית לגרנז'...........................................................................................................4

שיטת החצייה....................................................................................................................5

שיטת המיתר.....................................................................................................................6

שיטת ניוטון-רפסון.............................................................................................................7

שיטת רומברג.....................................................................................................................8

שיטת תרבועי גאוס........................................................................................................9-10

שיטת סימפסון.................................................................................................................11

שיטת נוויל.......................................................................................................................12

שיטת קירוב לינארי...........................................................................................................13

שיטת ספליין-קובי.......................................................................................................14-15

שיטת הטרפז....................................................................................................................16

שיטת SOR......................................................................................................................17

שיטת Vander-Monde.....................................................................................................18

References................................................................................................................19-20

מציאת שורשים: חציה, מיתר, ניוטון

מטריצות: יעקובי, גאוס-זיידל, SOR

קירוב פולינומי: לגרנז', נוויל, ספליין-קובי, וונדר-מונדה

אינטגרלים: גאוס, רומברג, טרפז, סימפסון

שיטת גאוס-זיידל

שיטת גאוס זיידל *(Gauss-Seidel Method)* היא שיטה איטרטיבית לפתרון מערכת משוואות לינאריות. את השיטה אפשר לממש על מטריצות בעלת איברים שונים מאפס על האלכסון הראשי והתכנסותה מובטחת רק אם המטריצה היא מטריצה אלכסונית-דומיננטית או שהמטריצה היא סימטרית וחיובית.

השיטה פותרת מערכת ריבועית של n משוואות לינאריות עם וקטור נעלמים x, כלומר Ax = b.

בקוד הנתון המימוש מבוצע עבור הנוסחה הבאה:

במימוש הקוד אנו משתמשים בספריית Numpy ובפונקציה Solve מתוך ספריית Scipy על מנת לוודא את נכונות התוצאה החוזרת מהפונקציה 'gauss'.

הפונקציה ממומשת עפ"י העיקרון הבא:

1. הפונקציה מקבלת את המטריצה A – המטריצה אותה אנו מעוניינים לפתור.
2. הפונקציה מקבלת את וקטור b – וקטור b הוא שיעזור לנו לגלות את ערכי הנעלמים.
3. הפונקציה מקבלת את וקטור x – וקטור המשתנים שעלינו לגלות.
4. הפונקציה תקבל גם משתנה שיגדיר עבורה את מספר האיטרציות המקסימלי.

הלולאה היחידה בפונקציה היא זו שתריץ את הנוסחה האיטרטיבית של גאוס כמספר הפעמים שהגדרנו עבורו ע"י שליחת המשתנה iterations\_number לפונקציה.

קוד בשפת פייטון:

import numpy as np  
from scipy.linalg import solve  
  
def gauss(A, b, x, iterations\_number):  
  
 L = np.tril(A)  
 U = A - L  
 for i in range(iterations\_number):  
 x = np.dot(np.linalg.inv(L), b - np.dot(U, x))  
 print(str(i).zfill(3),)  
 print(x)  
 print("solution: ")  
 return x

#Example:  
A = np.array([[4.0, -2.0, 1.0], [1.0, -3.0, 2.0], [-1.0, 2.0, 6.0]])

b = [1.0, 2.0, 3.0]

x = [1, 1, 1]

iterations\_number = 20

שיטת יעקובי

שיטת יעקובי *(Jacobi Method)* היא שיטה איטרטיבית למציאת פתרון מערכת משוואות לינאריות בעלות אלכסון דומיננטי, כאשר בכל איטרציה עבור כל אלכסון שנפתר מכניסים למטריצה את הערך המשוער. על התהליך חוזרים עד שמגיעים להתכנסות לתוצאה הרצויה.

בדומה לשיטת גאוס-זיידל השיטה פותרת מערכת ריבועית של n משוואות לינאריות עם וקטור נעלמים x, כלומר את המשוואה: Ax = b, כאשר בקוד הנתון המימוש מבוצע עבור הנוסחה הבאה:

במימוש הקוד נשתמש בספריית Numpy והפונקציה ממומשת עפ"י העיקרון הבא:

1. נאפס את המטריצה A עפ"י ערכי המשוואה הרלוונטיים עבור הפתרון.
2. נאפס את וקטור b המייצג עבורנו את הפתרון האפשרי.
3. לבסוף x יהיה הפתרון הרלוונטי עבורנו אם התוצאה אכן תתכנס כמצופה.
4. המשתנה iteration\_number יהווה את החסם העליון למספר האיטרציות האפשריות.

הלולאה היחידה שהמהווה את מימוש האלגוריתם תרוץ עד להתכנסות התוצאה בטווח הטעויות האפשרי שהגדרנו עבורו ע"י הפונקציה np.allclose(). במקרה זה טווח הטעות היא 1e-10.

קוד בשפת פייטון:

import numpy as np

iteration\_number = 1000  
# Example - initialize the matrix  
A = np.array([[10., -1., 2., 0.], [-1., 11., -1., 3.], [2., -1., 10., -1.], [0.0, 3., -1., 8.]])  
# Example - initialize the RHS vector  
b = np.array([6., 25., -11., 15.])  
  
x = np.zeros\_like(b)  
for it\_count in range(iteration\_number):  
 print("Current solution:", x)  
 x\_new = np.zeros\_like(x)  
 for i in range(A.shape[0]):  
 s1 = np.dot(A[i, :i], x[:i])  
 s2 = np.dot(A[i, i + 1:], x[i + 1:])  
 x\_new[i] = (b[i] - s1 - s2) / A[i, i]  
 if np.allclose(x, x\_new, atol=1e-10, rtol=0.):  
 break  
 x = x\_new

#Example:  
print("Solution:")  
print(x)  
error = np.dot(A, x) - b  
print("Error:")  
print(error)

אינטרפולציית לגרנז'

את פולינום האינטרפולציה נכתוב כך:  כאשר:  .

בקוד הבא נממש את הנוסחה הנתונה באמצעות פונקציה מסדר גבוה. בשיטת לגרנז' נייצר אינטרפולציה עבור הנקודות שהפונקציה מסדר גבוה *(interpolate\_lagrange)* מקבלת באמצעות המשתנים: x\_values & y\_values כאשר בפועל הפונקציה basis היא זו שתממש נוסחת האינטרפולציה.

אופן השימוש בפונקציה מחייב אותנו לייבא את הפונקציות הבאות:

1. Symbol & Simplify from Sympy library
2. Numpy library
3. Reduce from functools library
4. Operator library

השימוש בפונקציה יחייב אותנו לפעול עפ"י הסעיפים הבאים:

1. נשלח את ערכי ה-x בגרף למשתנה x\_values של הפונקציה מסדר גבוה.
2. נשלח את ערכי ה-y בגרף למשתנה y\_values של הפונקציה מסדר גבוה.
3. המשתנה x בחתימת הפונקציה *interpolate\_lagrange* ישמש לסימן 'x' בהצגת הפונקציה שתתקבל מהאלגוריתם. לפני שליחת ערך למשתנה x, יש להגדיר את 'x' בתור Symbol באמצעות הפונקציה Symbol בספריית Sympy.

קוד בשפת פייטון:

def interpolate\_lagrange(x, x\_values, y\_values):  
def basis(j):  
 p = [(x - x\_values[m])/(x\_values[j] - x\_values[m])

for m in range(k) if m != j]  
 return reduce(operator.mul, p)  
 assert len(x\_values) != 0 and (len(x\_values) == len(y\_values)), 'x and y cannot be empty and must have the same length'  
 k = len(x\_values)  
 return sum(basis(j)\*y\_values[j] for j in range(k))  
  
x = Symbol('x')  
poly = simplify(interpolate\_lagrange(x,[-1, 0, 1, 2],[3,-4, 5, -6]))  
x1 = np.linspace(-1, 2, 100)  
print('polynom: ', poly)  
print('x1: ', x1)

שיטת החצייה

שיטת החצייה *(Bisection Method)* מתבססת על כך שנניח שאנו רוצים לפתור את המשוואה f(x) = 0 כאשר f היא פונקציה רציפה. בהינתן שתי נקודותa & b שלערכי הפונקציה בהן יש סימנים הפוכים, ממשפט ערך הביניים נובע שלפונקציה יש לפחות שורש אחד במרווח [a, b].

במימוש האלגוריתם נשתמש בנוסחה הבאה: כאשר ידוע לנו כי אם

f  היא פונקציה רציפה במרווח [a, b] וכמו כן מתקיים: אז שיטת החצייה תתכנס עבורנו.

בתפעול השיטה יש להתייחס להנחיות הבאות:

1. השיטה func תחזיק את הפונקציה המתמטית שעבורה אנחנו נדרשים למצוא שורש.
2. הפונקציה bisection היא מימוש הנוסחה של שיטת החצייה ובחתימת הפונקציה עלינו לשלוח לה את המשתמשים הבאים:
   1. a – חסם צד שמאל
   2. b – חסם צד ימין
   3. epsilon – רמת דיוק התוצאה שאנו נדרשים לספק.

קוד בשפת פייטון:

def func(x):  
 return x\*\*3 - x\*\*2 + 2 # Change to required function  
  
# Prints root of func(x) with error of EPSILON  
def bisection(a, b, epsilon):  
 if (func(a) \* func(b) >= 0):  
 print("You have not assumed right a and b\n")  
 return  
 c = a  
 while ((b - a) >= epsilon):  
 # Find middle point  
 c = (a + b) / 2  
 # Check if middle point is root  
 if (func(c) == 0.0):  
 break  
 # Decide the side to repeat the steps  
 if (func(c) \* func(a) < 0):  
 b = c  
 else:  
 a = c  
 print("The value of root is : ", "%.8f" % c)  
  
# Driver code Initial values assumed  
a = -200  
b = 300  
epsi = 0.01  
bisection(a, b, epsi)

שיטת המיתר

שיטת המיתר *(Secant Method)* היא שיטה איטרטיבית למציאת שורשי פונקציה רציפה של משתנה יחיד. שיטת המיתר מקרבת את הנגזרת על ידי שיפוע המיתר המחבר את שתי הנקודות האחרונות שחושבו, ומכאן שמה. יתרונה של שיטת המיתר היא בכך שהיא אינה משתמשת בנגזרת: אם הנגזרת אינה ידועה, או שחישובה גוזל משאבי חישוב רבים, שיטת המיתר מתכנסת מהר יותר.

נממש את הנוסחה המתמטית המייצגת את שיטת המיתר:

בלולאת ה-while היחידה שמופיעה בגוף השיטה Secant\_method. ככל שהלולאה תרוץ ליותר איטרציות כך נקבל ערך שקרוב יותר ויותר לשורש האמיתי של המשוואה.

לפני תפעול השיטה Secant\_method יש לשים לב לנקודות הבאות:

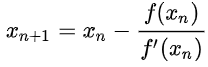
1. יש לייבא את הספריות Math & Sympy.
2. יש להגדיר את 'x' בתור Symbol (כלומר סימן מתמטי) ע"מ לכתוב את המשוואות המתמטיות בצורה אופיינית ולא ע"י וקטורי מקדמים.
3. הפונקציה Secant\_method תקבל את שלושת המשתנים הבאים:
   1. equation – מהווה את המשוואה המתמטית שאת שורשיה אנו נדרשים למצוא.
   2. x0, x1 – שני ערכים התחלתיים שמוטב שיהיו קרובים ככל הניתן לשורש המשוואה.

קוד בשפת פייטון:

def Secant\_method(equation,x0,x1):  
 sqr = x0  
 while sqr != x1:  
 sqr = x1  
 y1 = equation.evalf(subs= {x:x1})  
 print("y1:", y1)  
 y2 = equation.evalf(subs= {x:x0})  
 print("y2:", y2)  
 next\_x = x1-((y1\*(x1-x0))/(y1-y2))  
 print("next x:", next\_x, "\n")  
 x0=x1  
 x1=next\_x  
 print("x1: ", x1,"\n")  
 return x1  
x = sympy.Symbol('x')

# Example:  
equation = math.exp(1)\*\*x-3\*x\*\*2  
print(Secant\_method(equation,-3,1))

שיטת ניוטון-רפסון

שיטת ניוטון-רפסון (Newton-Raphson Method) היא שיטה נומרית למציאת שורשים של פונקציה ממשית באמצעות קירוב שנעשה עם המשיק לגרף הפונקציה. כדי לממש את השיטה נסתמך על הנוסחה הבאה:

השיטה ממומשת באמצעות שלוש פונקציות כאשר שתיים הן פונקציות עזר ופונקציה אחת היא העיקרית למימוש הנוסחה:

1. Func (עזר) – פונקציית עזר המקבלת את equation (הפונקציה) ואת x1 (ערך מספרי) ומחשבת את הערך המספרי של הפונקציה בנקודה x1.
2. Derivate (עזר) – מקבלת את אותם הפרמטרים בדומה ל-Func, אם כי פונקציה זו מחשבת את ערך נגזרת הפונקציה בנקודה x1.
3. newtonRaphson (עיקרית) – מקבלת את equation, ערך התחלתי x ורמת דיוק epsilon ומוציאה לפועל את הנוסחה המתמטית עד להגעה לרמת הדיוק הנדרשת עפ"י ה-epsilon הנתון.
4. יש לשים לב כי צריך להגדיר את 'x' בתור Symbol באמצעות הפונקציה Symbol של ספריית Sympy. זאת על מנת לכתוב את הפונקציה שאנו מעוניינים לחשב בכתיבה מתמטית אופיינית ולא ע"י וקטורי מקדמים.

קוד בשפת פייטון:

def func(equation, x1):  
 return equation.evalf(subs={x: x1})  
  
def derivate(equation, x1):  
 return equation.diff(x).evalf(subs={x: x1})  
  
def newtonRaphson(equation, x, epsilon):  
 h = func(equation, x) / derivate(equation, x)  
 while abs(h) >= epsilon:  
 h = func(equation, x) / derivate(equation,x)  
 # x(i+1) = x(i) - f(x) / f'(x)  
 x = x - h  
 print(x)  
 return("The value of the root is : ", "%.14f" % x)

x = sympy.Symbol('x')

# Example:

x0 = -1  
epsi = 0.0001  
equation = x\*\*3+2\*x\*\*2+10\*x-20  
print(newtonRaphson(equation, x0, epsi))

שיטת רומברג

שיטת רומברג *(Romberg's Method)* היא שיטה נומרית לחישוב אינטגרל סופי. שיטת רומברג מיישמת את אקסטרפולציית ריצ'רדסון באופן איטרטיבי על שיטת הטרפז *(הרחבה על שיטת הטרפז תופיע עבור קוד שיטת הטרפז אשר יוצג בהמשך המסמך)*.

השיטה ממומשת באמצעות שתי פונקציות – אחת עיקרית שתכיל את האלגוריתם שלנו והשנייה משנית שתכיל את האלגוריתם של שיטת הטרפז למימוש שיטת רומברג.

יש לשים לב לדגשים הבאים במימוש הקוד:

1. יש לייבא את ספריית Numpy.
2. הפונקציה trapezcomp מקבלת ארבעה משתנים:
   1. f שמייצגת עבורנו את הפונקציה עליה מבצעים אינטגרציה
   2. a שמייצגת עבורנו את החסם השמאלי של האינטגרל
   3. b שמייצגת עבורנו את החסם הימני של האינטגרל
   4. n מספר הריבועים הקטנים שניצור בקטע [a, b]
3. הפונקציה הראשית romberg תקבל את אותם המשתנים בדיוק עם שינוי אחד:
   1. הפונקציה תשלח את n לפונקציית העזר.
   2. הפונקציה תקבל את p (בנוסף למשתנים a, b, f ב-trapezcomp) כאשר p מייצג את מספר האיטרציות שיבצע האלגוריתם עד להתכנסות לתוצאה הרצויה.

קוד בשפת פייטון:

import numpy as np  
def trapezcomp(f, a, b, n):  
 h = (b - a) / n  
 x = a  
 In = f(a)  
 for k in range(1, n):  
 x = x + h  
 In += 2\*f(x)  
 return (In + f(b))\*h\*0.5  
def romberg(f, a, b, p):  
I = np.zeros((p, p))  
 for k in range(0, p):  
 I[k, 0] = trapezcomp(f, a, b, 2\*\*k)  
 for j in range(0, k):  
 I[k, j+1] = (4\*\*(j+1) \* I[k, j] - I[k-1, j]) / (4\*\*(j+1) - 1)  
 print(I[k,0:k+1])  
 return I

def func(x):  
 return np.sin(x)  
p\_rows = 4  
I = romberg(func, 0, np.pi/2, p\_rows)  
solution = I[p\_rows-1, p\_rows-1]  
print(solution)

שיטת תרבועי-גאוס

שיטת תרבועי-גאוס *(Gaussian quadrature / Gauss–Legendre Method)* היא שיטה נומרית לחישוב מקורב של אינטגרל מסוים באופן הבא:

1. ערך n המתקבל ע"י המשתמש קובע לאלגוריתם לכמה חלקים שווים לחלק את השטח הכלוא מתחת לגרף הפונקציה (הפונקציה המתמטית מיוצגת בפונקציה func).
2. החלוקה ל-n חלקים שווים מתבצעת ע"י האלגוריתם כך:
   1. באמצעות הפונקציה LegendereRoots אשר תפקידה לחשב את השורשים של פולינום לז'נדר.
   2. באמצעות הפונקציות GaussLegendreWeights & GaussLegendreQuadrature שתפקידן לחשב את הקבועים הנוספים הדרושים למימוש התהליך המתמטי.
3. בסוף התהליך האלגוריתם מבצע מעבר מתמטי בין קטע שהמשתמש בחר להכניס למערכת ע"י המשתנים a, b המהווים את הגבול השמאלי והימני של האינטגרל לקטע: [-1, 1].

* להפעלת האלגוריתם יש לייבא את ספריית Numpy.

קוד בשפת פייטון:

from numpy import \*  
# Recursive generation of the Legendre polynomial of order n  
def Legendre(n, x):  
 x = array(x)  
 if (n == 0):  
 return x \* 0 + 1.0  
 elif (n == 1):  
 return x  
 else:  
 return ((2.0 \* n - 1.0) \* x \* Legendre(n - 1, x) - (n - 1) \* Legendre(n - 2, x)) / n  
# Derivative of the Legendre polynomials  
def DLegendre(n, x):  
 x = array(x)  
 if (n == 0):  
 return x \* 0  
 elif (n == 1):  
 return x \* 0 + 1.0  
 else:  
 return (n / (x \*\* 2 - 1.0)) \* (x \* Legendre(n, x) - Legendre(n - 1, x))  
# Roots of the polynomial obtained using Newton-Raphson method  
def LegendreRoots(polyorder, tolerance=0.0001):  
 if polyorder < 2:  
 err = 1 # bad polyorder no roots can be found  
 else:  
 roots = []  
 # The polynomials are alternately even and odd functions. So we evaluate only half the number of roots.  
 for i in range(1, int(polyorder / 2 + 1)):  
 x = cos(pi \* (i - 0.25) / (polyorder + 0.5))  
 error = 10 \* tolerance  
 iters = 0  
 while (error > tolerance) and (iters < 1000):  
 dx = -Legendre(polyorder, x) / DLegendre(polyorder, x)  
 x = x + dx  
 iters = iters + 1  
 error = abs(dx)  
 roots.append(x)  
 # Use symmetry to get the other roots  
 roots = array(roots)  
 if polyorder % 2 == 0:  
 roots = concatenate((-1.0 \* roots, roots[::-1]))  
 else:  
 roots = concatenate((-1.0 \* roots, [0.0], roots[::-1]))  
 err = 0 # successfully determined roots  
 return [roots, err]  
# Weight coefficients  
def GaussLegendreWeights(polyorder):  
 W = []  
 [xis, err] = LegendreRoots(polyorder)  
 if err == 0:  
 W = 2.0 / ((1.0 - xis \*\* 2) \* (DLegendre(polyorder, xis) \*\* 2))  
 err = 0  
 else:  
 err = 1 # could not determine roots - so no weights  
 return [W, xis, err]  
# The integral value  
# func : the integrand  
# a, b : lower and upper limits of the integral  
# polyorder : order of the Legendre polynomial to be used  
#  
def GaussLegendreQuadrature(func, polyorder, a, b):  
 [Ws, xs, err] = GaussLegendreWeights(polyorder)  
 if err == 0:  
 ans = (b - a) \* 0.5 \* sum(Ws \* func((b - a) \* 0.5 \* xs + (b + a) \* 0.5))  
 else:  
 # (in case of error)  
 err = 1  
 ans = None  
 return [ans, err]  
# The integrand - change as required  
def func(x):  
 return exp(x)  
# Integrating the function  
[ans, err] = GaussLegendreQuadrature(func, 5, -3, 3)  
if err == 0:  
 print("Integral : ", ans)  
else:  
 print("Integral evaluation failed")

שיטת סימפסון:

שיטת השליש של סימפסון *(Simpson's Method)* היא שיטה נומרית לביצוע אינטגרציה מתמטית. בשיטה זו, עפ"י האלגוריתם, דרך כל שלוש נקודות אנו מקרבים את הפונקציה הנתונה לפולינום ממעלה שנייה (אינטרפולציה ריבועית) ועבור פולינום זה מחשבים שטח מתחת לגרף בתחום נתון באמצעות שיטת לגרנז'. במצב זה אנו מקבלים כי מתחת לכל קירוב קיימים שלוש נקודות ושתי קטעים שעל פיהם אנחנו מבצעים את החישוב.

יש לשים לב לדגשים הבאים במימוש הקוד:

1. יש לייבא את ספריית Numpy.
2. הפונקציה שעליה אנו מעוניינים לבצע אינטגרל נכתוב בפונקציית העזר f
3. הפונקציה הראשית simpsonsRule תקבל שלושה משתנים:
   1. a שמייצגת עבורנו את החסם השמאלי של האינטגרל
   2. b שמייצגת עבורנו את החסם הימני של האינטגרל
   3. n שמייצגת עבורנו את כמות המקטעים שנרצה לחלק את השטח מתחת לגרף הפונקציה על מנת לקרב את התוצאה לסכום הרצוי.

קוד בשפת פייטון:

def simpsonsRule(a, b, n):  
sum = float()  
 sum += f(a) #evaluating first point  
 sum += f(b) #evaluating last point  
 width=(b-a)/(2\*n) #width of segments  
 oddSum = float()  
 evenSum = float()  
 for i in range(1,n): #evaluating all odd values of n (not first and last)  
 oddSum += f(2\*width\*i+a)  
 print(oddSum)  
 sum += oddSum \* 2  
 for i in range(1,n+1): #evaluating all even values of n (not first and last)  
 evenSum += f(width\*(-1+2\*i)+a)  
 print(evenSum)  
 sum += evenSum \* 4  
 return sum \* width/3  
  
def f(x):return x #Enter equation here  
print(simpsonsRule(1,2,3))

שיטת נוויל

שיטת נוויל *(Neville's Method)* היא שיטה נומרית המשמשת לאינטרפולציה מתמטית. בהינתן n+1 נקודות קיים פולינום בדרגה קטנה או שווה ל-n שעובר בנקודות הנתונות. באמצעות שיטת נוויל אנו יכולים להעריך את הפולינום הנ"ל.

יש לשים לב לדגשים הבאים במימוש הקוד:

1. אין צורך להשתמש בספריות חיצונית כגון Numpy.
2. פונקציה זו מקבלת כקלט:
   1. שתי מערכים באורך זהה של נקודות: מערך x ומערך y.
   2. הנקודה הרצויה x שתשמש אותנו עבור האינטרפולציה.

קוד בשפת פייטון:

def neville(datax, datay, x):  
 n = len(datax)  
 p = n\*[0]  
 for k in range(n):  
 for i in range(n-k):  
 if k == 0:  
 p[i] = datay[i]  
 print('p[i] k == 0: ', p[i])  
 else:  
 p[i] = ((x-datax[i+k])\*p[i] +

(datax[i]-x)\*p[i+1]) / (datax[i]-datax[i+k])  
 print('p[i] k != 0: ', p[i])  
  
 return ('Result: ', p[0])  
  
print(neville((1,2,6),(1,-1,4),4))

שיטת קירוב ליניארי

שיטת הקירוב הליניארי *(Linear approx Method)* הוא מושג מתמטי המתאר [קירוב](https://he.wikipedia.org/wiki/%D7%A7%D7%99%D7%A8%D7%95%D7%91) של [פונקציה](https://he.wikipedia.org/wiki/%D7%A4%D7%95%D7%A0%D7%A7%D7%A6%D7%99%D7%94) מתמטית כלשהי באמצעות פונקציה ליניארית. לקירובים ליניאריים יש שימוש במדעים ובמתמטיקה כדי לקבל קירוב לערך הפונקציה בסביבה של ערך קבוע מראש. היות שפונקציות ליניאריות הן קלות לחישוב ולפתרון, קירובים ליניאריים מועדפים כמעט תמיד בניתוחים אנליטיים ונומריים אם הם מספקים את הדיוק הנדרש. כאשר לפונקציה קיים קירוב ליניארי, נאמר שהפונקציה דיפרנציאבילית.

הנוסחה שנממש ע"י להגיע לקירוב לינארי היא הנוסחה הבאה:

יש לשים לב לדגשים הבאים במימוש הקוד:

1. יש לייבא את ספריית Numpy.
2. יש לייבא את ספריית Sympy.
3. יש להגדיר את 'x' בתור "סימן" מתמטי ע"י פונקציית Symbol של ספריית Sympy.
4. ע"מ לייצג את המשוואה המתמטית ע"י וקטור המקדמים בלבד נשתמש בפונקציה Poly1d.
5. ע"י לייצג את נגזרת המשוואה המתמטית שאנו רוצים לקרב נשתמש בפונקציה Polyder.
6. הפונקציה Linear\_approx מקבלת שני משתנים: a & x.
7. הפונקציה func\_representation מחזירה את ערך הפונקציה בנקודה x.
8. הפונקציה func\_derivate\_value מחזירה את ערך נגזרת הפונקציה בנקודה x.

קוד בשפת פייטון:

from sympy import \*  
import numpy as np  
x = Symbol ('x')  
y = np.poly1d([1,0,1])  
yprime = np.polyder(y)  
def func\_representation(x):  
 return y(x)  
  
def func\_derivate\_value(x):  
 return yprime(x)  
  
def Linear\_approx(a, x):  
 fx = func\_representation(a) + func\_derivate\_value(a)\*(x-a)  
 return ('Liner approx f(x)={0}'.format(fx))

# Example:  
print(Linear\_approx(10,12))

שיטת ספליין-קובי

בשיטת ספליין קובי *(Cubic Spline)* בהינתן נקודות טבלה נעשה קירוב ע"י פולינום מדרגה שנייה בין כל 2 נקודות בטבלה. בכל נקודה בטבלה נדרוש שתהיה "החלקה" ע"י זה שנגדיר ששתי הנגזרות של שני הפולינומים בכל נקודת טבלה יהיו שוות.

בהתחלה בוחרים פולינום ממעלה 3 שעוברת דרך 2 הנקודות הראשונות בגלל שהגדרנו שהנגזרות שוות בנקודות, בעצם נקבעו לנו כל שאר הפולינומים שעוברים בכל נקודות הטבלה.

צורת כל פולינום מדרגה 3 יהיה ככה:

דגשים שיש לשים לב אליהם במימוש הקוד:

1. הפונקציה zeroV אחראית על החזרת וקטור אפסים בגודל m שהפונקציה מקבלת.
2. הפונקציה cubic\_spline מקבלת שלושה משתנים:
   1. xn – מערך של ערכים על ציר ה-x.
   2. a – מערך של ערכים על ציר ה-y.
   3. n – אורכם של שני המערכים.

קוד בשפת פייטון:

from numpy import poly1d  
  
def zeroV(m):  
 z = [0]\*m  
 return (z)  
  
def cubic\_spline(n, xn, a):h = zeroV(n-1)  
 alpha = zeroV(n-1)  
 l = zeroV(n+1)  
 u = zeroV(n)  
 z = zeroV(n+1)  
 b = zeroV(n)  
 c = zeroV(n+1)  
 d = zeroV(n)  
 for i in range(n-1):  
 h[i] = xn[i+1]-xn[i]  
 print('h=',h)  
 print("")  
  
 for i in range(1, n-1):  
 alpha[i] = (3./h[i])\*(a[i+1]-a[i])-(3./h[i-1])\*(a[i] - a[i-1])  
 print('alpha=', alpha)  
 print("")  
 l[0] = 1  
 u[0] = 0  
 z[0] = 0  
 for i in range(1, n-1):  
 l[i] = 2\*(xn[i+1] - xn[i-1]) - h[i-1]\*u[i-1]  
 u[i] = h[i]/l[i]  
 z[i] = (alpha[i] - h[i-1]\*z[i-1])/l[i]  
 l[n] = 1  
 z[n] = 0  
 c[n] = 0  
 print('l=', l)  
 print('u=', u)  
 print('z=', z)  
 print("")  
 for j in range(n - 2, -1, -1):  
 c[j] = z[j] - u[j] \* c[j + 1]  
 b[j] = (a[j + 1] - a[j]) / h[j] - h[j] \* (c[j + 1] + 2 \* c[j]) / 3.  
 d[j] = (c[j + 1] - c[j]) / (3 \* h[j])  
 print('a=', a)  
 print('b=', b)  
 print('c=', c)  
 print('d=', d)  
 print('')  
 polylist = list()  
 for j in range(n - 1):  
 root = poly1d(xn[j], True)  
 poly = d[j] \* (root) \*\* 3  
 poly = poly + c[j] \* (root) \*\* 2  
 poly = poly + b[j] \* root  
 poly = poly + a[j]  
 polylist.append([poly, xn[j], xn[j+1]])  
 print('S{}(x)=\n'.format(j), poly, 'when {}<=x<={}'.format(xn[j], xn[j+1]))  
 return polylist  
p = cubic\_spline(7,[-1.5,-0.2,1,5,10,15,20],

[-1.2,0,0.5,1,1.2,2,1])  
print(p)

שיטת הטרפז

שיטת הטרפז *(Trapezoidal rule)* היא שיטת אינטגרציה נומרית המשמשת לחישוב מקורב של אינטגרלים מסוימים (בעלי גבולות אינטגרציה מוגדרים). ניתן להשתמש בשיטה בשביל לבצע קירוב לאינטגרלים אשר אין להם פתרון אנליטי, פונקציה קדומה או שהפונקציה שלהם נתונה ע"י ערכים בדידים (כמו בניסויים).

השיטה מבצעת את הקירוב ע"י חלוקה של הפונקציה לקטעים ויצירת טרפזים עם ציר ה- x בעזרת שתי נקודות בקצה כל קטע כאשר הציר מהווה את בסיס הטרפזים וערך הפונקציה בקצה כל קטע מהווה את גודל הצלעות, לאחר החלוקה מתבצע חישוב שטח כל הטרפזים וסכום כל השטחים מהווה את הקירוב של האינטגרל.

הקוד הנתון מסתמך על הנוסחה הבאה:

כאשר עבור N נקודת מספר הטרפזים הוא N-1 ודלטא של Xk הוא אורך הבסיס של הטרפז, במקרה שלנו זהו גודל קבוע לכל הטרפזים.

הפונקציה ממומשת בצורה הבאה:

1. f – פונקציה עבורה נרצה לבצע קירוב של אינטגרל.
2. a – גבול תחתון של האינטגרל.
3. b– גבול עליון של האינטגרל.
4. n – מספר החלוקות של הקטע.

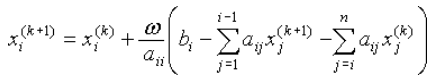
הפונקציה קודם כל מייצרת מערך של חלוקה שווה עבור הקטע בו נעשית האינטגרציה ומחשבת את ההפרש הנוצר בין הנקודות כתוצאה מחלוקה כזאת, לאחר מכן בעזרת לולאה הפונקציה מחשבת את שטח כל אחד מהטרפזים ומוסיפה אותו לסכום השטחים הכללי שזהו בעצם הקירוב של האינטגרל וסכום השטחים הוא הערך המוחזר.

קוד בשפת פייטון:

from math import exp

def g(t):  
 return exp(-t\*\*4)  
  
def trap(f, a, b, n):h = (b-a)/float(n)  
 s = 0.5\*(f(a) + f(b))  
 for i in range(1,n,1):  
 s = s + f(a + i\*h)  
 return h\*s  
  
a = -2; b = 2  
n = 1000  
result = trap(g, a, b, n)  
print (result)

שיטת SOR

שיטת SOR *(Successive Over Relaxation)*  היא שיטה נומרית איטרטיבית לפתרון מערכת משוואות לינאריות מהסוג Ax=b בצורה מהירה מאוד. השיטה הנתונה בשפת פייטון מממשת את הנוסחה הנתונה הבאה:

יש לשים לב לדגשים הבאים במימוש הקוד:

1. A היא זו שמייצגת עבורנו את מטריצת המקדמים.
2. b היא זו שמייצגת עבורנו את מטריצת התוצאות.
3. Eps בחתימת הפונקציה sor היא זו שמייצגת עבורנו את רמת דיוק התוצאה.

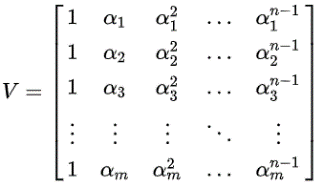
קוד בשפת פייטון:

def sor(A,b, w=1.25, x0=None, eps=1e-5, max\_iteration=100,pItr=0):for i in range(len(A)):  
 A[i].append(b[i][0])  
 m=A  
 n = len(m)  
 x0 = [0] \* n if x0 == None else x0  
 x1 = x0[:]  
 itr=0  
 for \_\_ in range(max\_iteration):  
 for i in range(n):  
 s = sum(-m[i][j] \* x1[j] for j in

range(n) if i != j)  
 x1[i] = w \* (m[i][n] + s) / m[i][i] +

(1 - w) \* x0[i]  
 itr+=1  
 if(pItr==1):  
 print("X[",itr,"]=",x1)  
 if all(abs(x1[i] - x0[i]) < eps for i in range(n)):  
 return x1  
 x0 = x1[:]  
 raise ValueError('Solution does not converge')  
  
A = [[4, 3.2, 0.5], [2.2, 3, -0.3], [-3.1, -0.2, 4]]  
b=[[9.2],[0.9],[7]]  
#print without iterations of Xi  
print("Vector x of Ax=b: ",sor(A,b))  
#print with iterations of Xi  
print("Vector x of Ax=b: ",sor(A,b,pItr=1))

שיטת Vander-Monde

שיטת *Vander-Monde* היא שיטה שמבוססת על מטריצת ואן-דר-מונדה. מטריצת ואן-דר-מונדה היא מטריצה nXm שבה כל שורה היא סדרה הנדסית מהצורה הבאה:

מטריצה זו מעריכה פולינום בנקודות ספציפיות, היא מעבירה את המקדמים של הפולינום (וקטור res המוחזרת מהפונקציה הראשית( לערכים שהפולינום מקבל בנקודות ספציפיות. מסיבה זו אפשר לבצע באמצעות שיטה זו אינטרפולציה פולינומית.

יש לשים לב לדגשים הבאים במימוש הפונקציה:

1. הפונקציה poly\_approx\_vandermonde מקבלת שני מערכים ופועלת כך:
   1. X – מערך נקודות על ציר ה-X.
   2. Y – מערך נקודות על ציר ה-Y.
   3. הפונקציה כופלת את המטריצה ההופכית שהיא ייצרה ממערך X במערך Y, מדפיסה את הפולינום שנוצר ומחזירה את המטריצה המוכפלת.
2. הפונקציה vecToFunc מקבלת את הערך המוחזר של הפונקציה הראשית (כלומר res) ומחזירה את ערכו בנקודה ספציפית.

קוד בשפת פייטון:

import numpy as np  
def poly\_approx\_vandermonde(X, Y):matX = np.vander(X, len(X), True)  
 invMatX = np.linalg.inv(matX)

res = np.dot(invMatX, Y)  
 print("Polynom: ", end='')  
 for i in range(len(res) - 1):  
 print("{}x^{} + ".format(res[i], i), end='')  
 print("{}x^{}".format(res[len(res) - 1], len(res) - 1), end='')  
 return res  
  
def vecToFunc(vector):def f(x):  
 f = 0  
 for i in range(len(vector)):  
 f += vector[i] \* x \*\* i  
 return f  
 print('')  
 return f  
  
x = [1, 3, 5]  
y = [10.5, 6.1, 3.5]  
t = poly\_approx\_vandermonde(x, y)  
print(vecToFunc(t)(4))

***References***

*Gauss-Seidel Method:*

[*https://www3.nd.edu/~zxu2/acms40390F12/Lec-7.3.pdf*](https://www3.nd.edu/~zxu2/acms40390F12/Lec-7.3.pdf)

*Jacobi Method:*

[*https://www3.nd.edu/~zxu2/acms40390F12/Lec-7.3.pdf*](https://www3.nd.edu/~zxu2/acms40390F12/Lec-7.3.pdf)

[*https://www.maa.org/press/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi-jacobis-method*](https://www.maa.org/press/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi-jacobis-method)

*Lagrange Interpolation:*

[*http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html*](http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html)

*Bisection Method:*

<http://www.mathcs.emory.edu/~cheung/Courses/170/Syllabus/07/bisection.html>

<http://mathworld.wolfram.com/Bisection.html>

*Secant Method:*

<https://mat.iitm.ac.in/home/sryedida/public_html/caimna/transcendental/iteration%20methods/secant/secant.html>

*Newton-Raphson Method:*

<https://nptel.ac.in/courses/122104019/numerical-analysis/Rathish-kumar/ratish-1/f3node6.html>

*Romberg's Method:*

<https://webcourse.cs.technion.ac.il/234107/Spring2017/ho/WCFiles/tut7_slides.pdf>

<https://learnche.org/3E4/Numerical_differentiation_and_integration>

*Gaussian quadrature / Gauss–Legendre Method:*

<https://rosettacode.org/wiki/Numerical_integration/Gauss-Legendre_Quadrature>

<https://www.asc.ohio-state.edu/physics/ntg/6810/readings/hjorth-jensen_notes2013_05.pdf>

*Simpson's Method:*

<https://www.intmath.com/integration/6-simpsons-rule.php>

*Neville's Method:*

<https://github.com/gisalgs/geom/blob/master/neville.py>

Spline-Cubic Method:

<http://mathworld.wolfram.com/CubicSpline.html>

<https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/05Interpolation/splines/>

Trapezoidal Method:

<http://www.mathwords.com/t/trapezoid_rule.html>

Successive Overrelaxation Method:

<http://mathworld.wolfram.com/SuccessiveOverrelaxationMethod.html>

Vander-Monde Method:

<https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/05Interpolation/vandermonde/>

Linear Approximation:

<http://tutorial.math.lamar.edu/Classes/CalcI/LinearApproximations.aspx>