

## מסמך הסבר קוד להאקטון ומדריך למשתמש לכל אלגוריתם

שיטת החצייה – Bisection method:

שיטת החצייה מתאימה אך ורק לשורשים מריבוב אי זוגי. כלומר ש:

מספר קטן מאוד  $j$  כך ש-  $f(a-j)*f(a+j)<0$

שיטת החצייה שכתבנו, קודם כל בודקת אם השורש הוא מריבוב אי זוגי. אם כן, נמשיך באלגוריתם. אחרת, נעצור את התוכנית.

Input:

ax – point a

bx – point b

function f

epsilon – point of accuracy. If the answer is in this epsilon bounds, we will stop the function and return the value.

Output:

The function bisection method prints the a and b values along the algorithm. In the end of the algorithm the function prints the result value of the root.

האלגוריתם האיטרטיבי בודק כל פעם איפה השורש. בין נקודת האמצע לנקודה השמאלית או בין האמצע לנקודה הימנית. ולפי זה קובע גבולות חדשים.

אם התוצאה מספיק קרובה לרמת הדיוק שהגדרנו (אפסילון) נעצור ונחזיר את השורש אליו הגענו.

User guide:

First enter a function f in python format. (Example shown in picture)

Then enter a and b values:

```
Run bisection_method
C:\Python34\python.exe C:/Users/רן יאלי/PycharmProjects/Numerical/bisection_method.py
Enter a function f(x):
x**2-2
Enter range: a and b
a=1
b=2
aX: 1.0, bX: 2.0
aX: 1.0, bX: 1.5
aX: 1.25, bX: 1.5
aX: 1.375, bX: 1.5
aX: 1.375, bX: 1.4375
aX: 1.40625, bX: 1.4375
aX: 1.40625, bX: 1.421875
aX: 1.4140625, bX: 1.421875
aX: 1.4140625, bX: 1.41796875
aX: 1.4140625, bX: 1.416015625
aX: 1.4140625, bX: 1.4150390625
aX: 1.4140625, bX: 1.41455078125

The root is: 1.4141845703125

Process finished with exit code 0
|
```

## שיטת המיתר – Secant method

עבור שיטה זו דרושים שני ניחושים התחלתיים. שיטה זו היא שיטה איטרטיבית למציאת שורשים של פונקציה רציפה בין שני נקודות (שהם הניחושים ההתחלתיים). נעצור את האלגוריתם כאשר אנו מתקרבים לדיוק מספיק טוב (שאנו הגדרנו – אפסילון).

האלגוריתם:

$$x_n = x_{n-1} - f(x_{n-1}) \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

Input:

Function f

x0 – first point (guess)

x1 – second point (guess)

epsilon – point of accuracy. If the answer is in this epsilon bounds, we will stop the function and return the value.

nMax – number of maximum iterations.

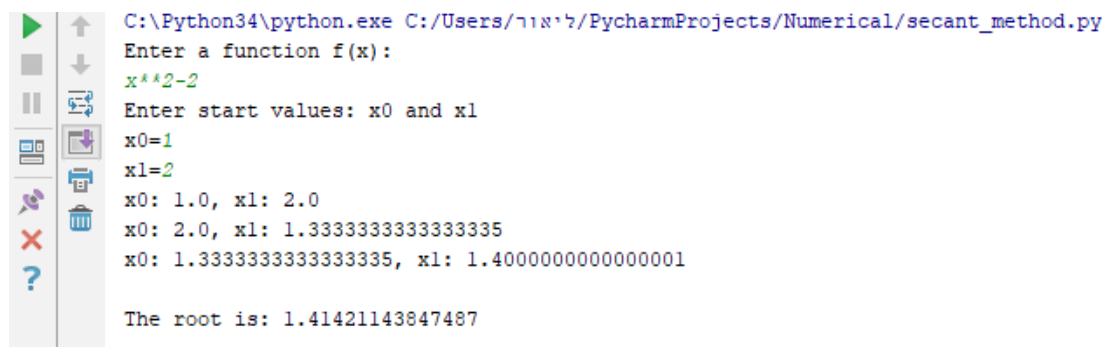
Output:

The function secant method prints the a and b values along the algorithm calculated according to the secant method algorithm. In the end of the algorithm the function prints the result value of the root.

User guide:

First enter a function f in python format. (Example shown in picture)

Then enter x0 and x1 values:



```
C:\Python34\python.exe C:/Users/ליאור/PycharmProjects/Numerical/secant_method.py
Enter a function f(x):
x**2-2
Enter start values: x0 and x1
x0=1
x1=2
x0: 1.0, x1: 2.0
x0: 2.0, x1: 1.3333333333333335
x0: 1.3333333333333335, x1: 1.4000000000000001

The root is: 1.41421143847487
```

## שיטת ניוטון-רפסון – Newton-Rapson method:

שיטת ניוטון-רפסון היא שיטה איטרטיבית למציאת שורשים של משוואה. שיטה זו עושה שימוש בנגזרת של נקודה. כלומר, הנגזרת צריכה להיות מוגדרת וצרי לחשב אותה.

חישבנו נגזרת לפונקציה נתונה והחזרנו פונקציה שהיא פונקציית הנגזרת שלה, לפי פתרון נומרי:

```
def derive(f):  
    """  
    This function return the numerical calculated derived function of f.  
    :param f: the function  
    :return: derived function  
    """  
    h = 10**-10  
    return lambda x: (f(x + h) - f(x))/float(h)
```

לפי הגזרה של נגזרת של פונקציה:

$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

בפיתרון שלנו המשתנה ששואף לאפס שווה ל10 בחזקת מינוס 10 (מספר ממש קטן).

האלגוריתם:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Input:

For derive function:

function f

For newton-Raphson method:

function f

start value x0

epsilon – point of accuracy. If the answer in in this epsilon bounds, we will stop the function and return the value.

nMax – number of maximum iterations.

Output:

For derive function:

the derived function of the function f

For newton-Raphson method:

the algorithm returns false if we finished the while loop without getting close to epsilon.

otherwise, the algorithm returns the value of the root of the function that was

given.

Because, in this algorithm we divide from the value of  $f(x_n)$  the value of  $f'(x_n)$ , we check first that the  $f'(x_n)$  is different than zero.

User guide:

First enter a function  $f$  in python format. (Example shown in picture)

Then enter  $x_0$  start value:

```
C:\Python34\python.exe C:/Users/רור/PycharmProjects/Numerical/newtonraphson_method.py
Enter a function f(x):
x**2-2
Enter x0 start value:
1
x0: 1.0, x1: 1.499999958629818
x0: 1.499999958629818, x1: 1.416666673561693

The root is: 1.416666673561693
```

### קירוב (אינטרפולציה) לינארית:

נשתמש במתודה זו כשנרצה לחשב ערך בין שני נקודות קרובות (יחסית) אחת לשנייה, ושהנקודות הן ידועות בגרף.

נפתור לפי האלגוריתם שלמדנו בהרצאה.

#### Input:

$x_1$  – value  $x_1$  of the known point  $x$  in the graph.

$x_2$  – value  $x_2$  of the known point  $x$  in the graph.

$y_1$  – value  $y_1$  of the known point  $y$  in the graph.

$y_2$  – value  $y_2$  of the known point  $y$  in the graph.

$x_f$  – the value of the  $x$  point in which we want to approximate its  $y$  value.

#### Output:

The linear approximation of the given values.

שיטת דירוג גאוס – Gauss elimination method:

הרעיון של האלגוריתם הוא לבצע רצף של פעולות אלמנטריות כדי להביא אותה לצורה מדורגת קנונית.

URL: [https://github.com/WaizungTaam/Gaussian-Elimination/blob/master/gaussian\\_eliminate.py](https://github.com/WaizungTaam/Gaussian-Elimination/blob/master/gaussian_eliminate.py)

השתמשנו בספריית

numpy

כדי לייצג מטריצות.

Input:

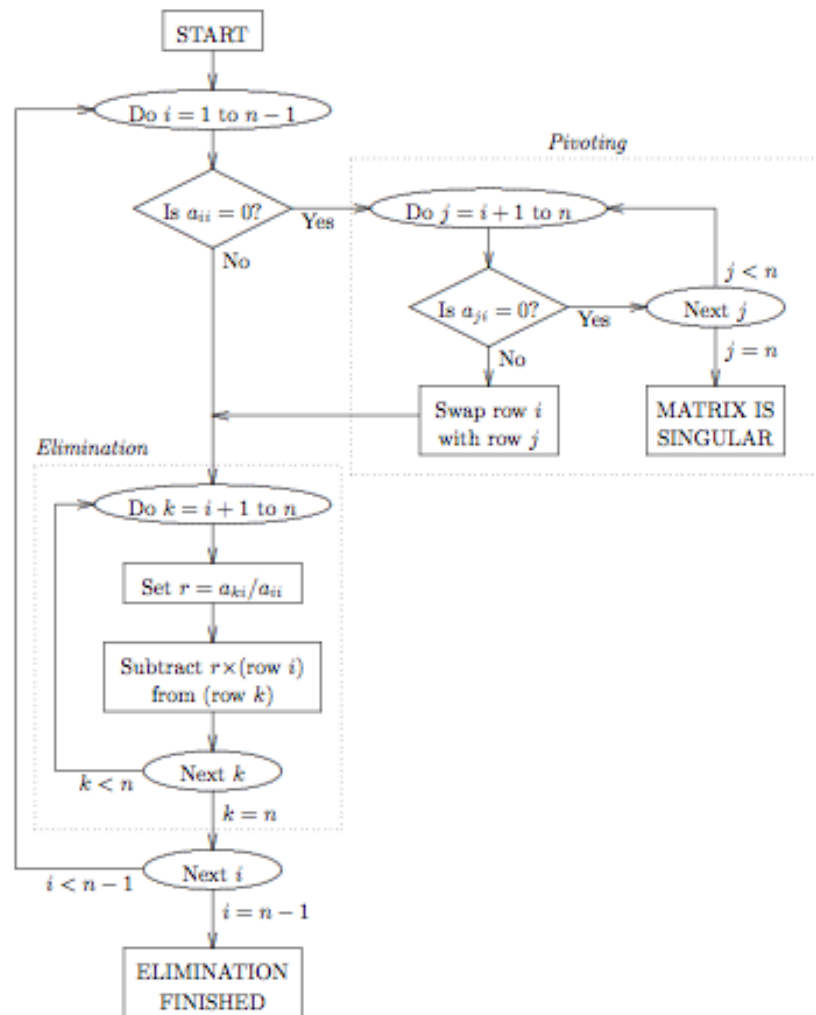
Matrix A, size of  $N \times N$

Output:

The algorithm prints the canonical eliminated matrix A with the result vector as vector b.

The algorithm returns and prints the result vector b.

gauss flowchart:





### קירוב (אינטרפולציה) פולינומיאלית:

קירוב פולינומיאלי מקבל מילון (בפייטון) בטור טבלה.  
מהמילון אנו יוצרים מטריצה בגודל של המילון. מערכי ה  
value  
אנו יוצרים וקטור פיתרונות.

את המטריצה מדרגים לקבלת מטריצה מדורגת קנונית ואת ווקטור התוצאות אנו לוקחים ויוצרים  
ממנו פולינום. באמצעות ספרייה מוכנה בפייטון.

Input:

Dictionary in python with keys and values.

Output:

Polynomial of the approximation.

User guide:

Enter a dictionary as:

{key1:val1, key2:val2, ..., keyn:valn}

## הסבר על האלגוריתמים - המשך:

### פתרון מטריצה שיטות איטרטיביות:

### מבנה שיטה איטרטיבית:

כל שיטה איטרטיבית ניתן לפרק לתהליך:

$$M * X = b$$

$$M = (D + U + L)$$

$$x^{i+1} = G * x^i + H * b$$

נתחיל עם ניחוש  $x^0$  ווקטור 0, ונתקדם לווקטור  $x^N$  כאשר N מוגדר על ידנו. אם ייתכנס לפני שהגענו ל N נחזיר את התוצאה. תנאי עצירה הוא  $x^i - x^{i-1} < e$ , כאשר e הוא טווח השגיאה המירבי. אם הניחוש שהתקבל  $x^i$  לא מקיים  $M * x^i = b$  לא נעצור ונקטין את טווח השגיאה (במימוש שלי באופן שרירותי  $e = \frac{e}{10}$ ). בסוף התהליך נוכל לדעת אם הניחוש שהתקבל מהווה פתרון למטריצה או לא.

```
def iterSolver(self,g,h,result=None,startingGuess=None,maxError=0.001,**kwargs):
    """
    performs x[i+1] = g*x[i] + h*b, stops when reaches n = max iteration or reaches valid guess
    """
    matrixDegree = self.shape[0]
    startingGuess = Matrix([[0] for _ in range(matrixDegree)]) if startingGuess==None else startingGuess
    result = Matrix([[i] for i in result]) if not isinstance(result, np.matrix) else result

    def guessError(guess):
        return abs(result - (self * guess)).max()

    def iteration(n=25):
        currGuess=startingGuess
        i=1
        err = maxError
        while(i<=n):
            nextGuess = g*currGuess + h*result
            if(abs(nextGuess - currGuess).max())<err:
                currGuess = nextGuess
                if(guessError(currGuess)<maxError):
                    break
            else:
                print("close, but not enough; decreasing range")
                err = err/10
            currGuess = nextGuess
            i+=1
            print("guess #{}: {}".format(i-1,currGuess.tolist()))
        print("done in iteration number : " , i-1)
        return currGuess

    guess = iteration(**kwargs)
    print("guess error:",guessError(guess)," , result guess:\n",guess,'')
    return guess
```

## : Jacobi+Gauss Seidel+SOR

נציב בפונקציה של פתרון מטריצה בשיטה איטרטיבית את G,H בהתאם לשיטה:  
\* נעזרנו בNumpy לצורך חישוב מטריצה הופכית  
\* בנינו פונקציה לפירוק מטריצה לגורמיה (D,L,U)

:Jacobi

$$G = -D^{-1} * (L + U)$$
$$H = D^{-1}$$

:Gauss Seidel

$$G = -(D + L)^{-1} * U$$
$$H = (D + L)^{-1}$$

:SOR

$$0 < w < 2$$

$$G = (D + w * L)^{-1} * ((1 - w) * D - w * U)$$

$$H = w * (D + w * L)^{-1}$$

```
def iterJacobi(self,result,**kwargs):
    lu,d = Matrix.decompose(self,'LU','D')
    dInver = d**-1
    g = -dInver*lu
    h = dInver
    return Matrix.iterSolver(self, g, h, result,**kwargs)

def iterGaussSeidel(self,result,**kwargs):
    dl,u = Matrix.decompose(self,'DL','U')
    dlInver = dl**-1
    g = -dlInver*u
    h = dlInver
    return Matrix.iterSolver(self, g, h, result,**kwargs)

def iterSOR(self,result,w=None,**kwargs):
    w = 2*random_sample() if (w==None or w>2 or w<=0) else w
    d,l,u = Matrix.decompose(self,'D','L','U')
    dwlInver = (d+w*l)**-1
    g = dwlInver * ((1-w)*d -w*u)
    h = w * dwlInver
    print("w: ", w)
    return Matrix.iterSolver(self, g, h, result, **kwargs)
```

## קירוב ( אינטרפולציה ) פולינומיאלית:

נשתמש במתודה זאת על מנת לייצר פולינום שיהווה קירוב פולינומי לפונקציה נקודתית. ייצגנו פונקציה נקודתית ע"י מילון (מפתח = x, ערך = f(x) כאשר f היא הפונקציה הנקודתית)

נפתור לפי האלגוריתם שלמדנו בהרצאה.  
ייצרנו מטריצה כך שכל שורה במטריצה מהווה פתרון של הפולינום הסופי לכל  $x \rightarrow f(x)$  בפונקציה הנקודתית:

$$\text{matrix}A = \begin{pmatrix} (x_1)^{n-1} & (x_1)^{n-2} & \dots & (x_1) & 1 \\ (x_2)^{n-1} & (x_2)^{n-2} & \dots & (x_2) & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ (x_n)^{n-1} & (x_n)^{n-2} & \dots & (x_n) & 1 \end{pmatrix} \quad | \quad \text{vector}B = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix}$$

ופתרנו את מטריצה זאת ע"י פונקציה פתרון מטריצה כלשהיא, כלומר matrixSolver יקבל פעם את שיטת גאוס-סיידל ופעם אחרת את שיטת SOR.  
ווקטור הפתרון:

$$\text{vectorResults} = \begin{pmatrix} z_n \\ \vdots \\ z_1 \end{pmatrix}$$

פתרון המטריצה במיקום i:  $z_i$  מהווה מקדמי חזקת  $x^i$  בפולינום התוצאה, ממנו ייצרנו פולינום מספריית NumPy.

```
def polynomial_approximation(dict, matrixSolver = gauss_elim):
    """
    :param dict: the dictionary that was given
    :return: result vector as a polynom.
    """

    def createMatrix():
        """
        create the matrix
        :return: the matrix from the dictionary
        """
        size = len(dict.keys())
        return np.matrix([[x ** i for i in range(size - 1, -1, -1)] for x in dict.keys()])

    def createResultsVector():
        """
        :return: the resut vector
        """
        return np.matrix([[y] for y in dict.values()])

    # def unpackVector(vector):
    #     return vector.flatten()

    matrixA , vectorB = createMatrix(), createResultsVector()
    vectorX = matrixSolver(matrixA,vectorB)
    if isinstance(vectorX, np.matrix):
        vectorX = vectorX.flatten().tolist()[0]
        print(vectorX)
    return np.poly1d(vectorX)
```

## האקטון שני – אנליזה נומרית

### מסמך הסבר לאלגוריתמים

#### שיטת תרבועי גאוס:

באנליזה נומרית, כלל תרבווע הוא שיטת קירוב של האינטגרל המסוים של פונקציה, שבדרך כלל מנוסח כסכום משוקלל של ערכים של הפונקציה בנקודות מסוימות בתוך תחום האינטגרציה. כלל תרבווע גאוסיאני בעל  $n$  נקודות הוא כלל תרבווע שנבנה במיוחד לצורך חישוב תוצאה מדויקת לאינטגרל של פולינומים ממעלה  $2n-1$  או פחות באמצעות בחירה מתאימה של הצמתים  $x_i$  והמשקלים  $w_i$  בעבור  $i = 1, \dots, n$ . תחום האינטגרציה השכיח ביותר של פונקציות הוא הקטע  $[-1, 1]$ , כך שהכלל מנוסח למקרים אלו כך:

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i),$$

#### קלט:

האלגוריתם שיטת תרבועי גאוס מקבל כקלט:

1. פונקציה
2. גבולות של האינטגרל (השימוש הנהוג הוא  $[-1, 1]$ )

#### פלט:

האלגוריתם לשיטת קירוב של האינטגרל המסוים (טווחים סופיים) של פונקציה מחזיר כפלט את ההערך של האינטגרל שהוא מצא (כלומר, מחזיר חישוב אינטגרל).

#### מדריך למשתמש:

בשלב ראשון, נכניס את הפונקציה ולאחר מכן את גבולות האינטגרל. לאחר מכן התכנית תדפיס את ערך האינטגרל המצופה ואת הערך שהתקבל בפועל.

## שיטת הטרפז:

שיטת הטרפז זו שיטה לחישוב (נומרי) לשטח הכלוא מתחת לגרף פונקציה  $f$  עבור קטע כלשהו  $a$  ו- $b$ .  
מניחים כי הפונקציה  $f$  רציפה. השיטה מחלקת את הקטע  $[a,b]$  למספר קטעים שווים.

## דרישות:

1. הפונקציה  $f$  רציפה בקטע  $[a,b]$ .
2.  $b \geq a$ .

## קלט:

האלגוריתם של שיטת הטרפז מקבל כקלט:

1. גבולות  $a$  ו- $b$ .
2. פונקציה  $f$  רציפה בקטע של הגבולות.
3. מספר קטעים שווים (אינטרוולים) שרוצים לחלק.

## פלט:

האלגוריתם מדפיס את השטח הכלוא מתחת לגרף הפונקציה  $f$  בקטע שבין  $a$  ל- $b$  ומחזיר את ערך השטח.  
ערך שטח ולכן נחזיר את הערך המוחלט של התוצאה שחישבנו.

## מדריך למשתמש:

בעת הרצת התוכנית, התוכנית תבקש מהמשתמש להכניס פונקציה. הפונקציה  $f$  תיכתב באופן הבא:  
עבור הפונקציה  $x^2 - 4x$  נכתוב  $x**2 - 4*x$ .  
לאחר מכן המשתמש יתבקש להכניס תחום  $a$  ו- $b$  ולבסוף את מספר האינטרוולים (השווים) שנרצה לחלק את תחום הפונקציה.

```
C:\Python34\python.exe C:/Users/ליאור/PycharmProjects/Hackaton2/trapeze_method_integral.py
Enter a function f(x):
x**2+4*x
Enter range: a and b
a=0
b=5
Enter number of equals sections:
n=100
The area below the function f(x) is: 91.66875
```

## שיטת פולינום לגרנז:

פולינום לגרנז הוא פולינום שמורכב מפולינומים  $q_i(x)$ . הפולינום של לגרנז  $P_n(x)$  מחזיר עבור ערך  $x$  את הקירוב הנומרי שלו לפי הפולינום. מקבלים  $n$  נקודות טבלה ויוצרים פולינום לגרנז ממעלה  $n-1$ .

נשתמש בפולינום לגרנז שנקודות הטבלה הנתונות קרובות אחת לשנייה.

קלט:

האלגוריתם לקבלת פולינום לגרנז מקבל כקלט:

3.  $N$  נקודות טבלה, עם ערכי  $X_i$  ו  $Y_i$ .

4. נקודה  $x=x_0$  עבורה רוצים לחשב את ערך הפולינום (בנקודה  $x_0$ ).

פלט:

האלגוריתם למציאת פולינום לגרנז מחזיר כפלט את הפולינום שהוא מצא (כלומר, מחזיר פונקציה), פולינום ממעלה  $n-1$  עבור  $n$  נקודות טבלה.

את הערך  $x_0$  שקיבלנו מהמשתמש נציב בפולינום (פונקציה) שהוחזרה לקבלת ערך הפולינום בנקודה הזאת.

התוכנית תדפיס את ערך הפולינום בנקודה  $x_0$  שקיבל.

מדריך למשתמש:

בשלב ראשון, ניצור  $n$  נקודות טבלה עם ערכי  $(x,y)$ . לקבלת פולינום לגרנז ממעלה  $n-1$ . לאחר מכן, נכניס את הערך  $x_0$ .

לקבלת ערך הפולינום בנקודה הזאת.

התוכנית תבקש מהמשתמש את מספר נקודות הטבלה  $n$ , ואז המשתמש יכניס  $n$  נקודות טבלה. לאחר מכן המשתמש יכניס ערך נקודה  $x_0$ . לקבלת ערך פולינום נויל בנקודה.

```
C:\Python34\python.exe C:/Users/ליאור/PycharmProjects/Hackaton2/lagrange_approx.py
Enter number of table points=
3
Enter table points:
x=0
y=1
Enter table points:
x=1
y=0
Enter table points:
x=2
y=1
Enter the value of X0:
3
The value of the polynomial at point x is: 4.0
```

## שיטת פולינום נויל:

הרעיון הוא לבנות את הפולינום בשלבים ובכל פעם לעשות חישובים על מספרים בסדרי גודל דומים. בפולינום נויל מעניין אותנו לשערך נקודות בתוך הקטע אז ככל שנוסיף נקודות דגימה, המרחק דלתא יקטן והשגיאה תקטן.

נקרא גם פולינום ממעלה הולכת וגדלה, כי בכל איטרציה מחשבים פולינומים מסדר גבוה יותר.

### קלט:

האלגוריתם לקבלת פולינום נויל מקבל כקלט:

1.  $N$  נקודות טבלה, עם ערכי  $Y_i$  ו- $X_i$ .
2. נקודה  $x_0$  עבורה רוצים לחשב את ערך הפולינום (בנקודה  $x_0$ ).

### פלט:

האלגוריתם למציאת פולינום נויל מחזיר כפלט את הפולינום שהוא מצא (כלומר, מחזיר פונקציה), פולינום ממעלה  $n-1$  עבור  $n$  נקודות טבלה. את הערך  $x_0$  שקיבלנו מהמשתמש נציב בפולינום (פונקציה) שהוחזרה לקבלת ערך הפולינום בנקודה הזאת. התוכנית תדפיס את ערך הפולינום בנקודה  $x_0$  שקיבל.

### מדריך למשתמש:

בשלב ראשון, ניצור  $n$  נקודות טבלה עם ערכי  $(x, y)$ . לקבלת פולינום נויל ממעלה  $n-1$ . לאחר מכן, נכניס את הערך  $x_0$ . לקבלת ערך הפולינום בנקודה הזאת.

התוכנית תבקש מהמשתמש את מספר נקודות הטבלה  $n$ , ואז המשתמש יכניס  $n$  נקודות טבלה. לאחר מכן המשתמש יכניס ערך נקודה  $x_0$ . לקבלת ערך פולינום נויל בנקודה.

```
C:\Python34\python.exe C:/Users/ליאור/PycharmProjects/Hackaton2/lagrange_approx.py
Enter number of table points=
4
Enter table points:
x=0
y=5
Enter table points:
x=1
y=2
Enter table points:
x=2
y=-5
Enter table points:
x=3
y=-10
Enter the value of X0:
3
The value of the polynomial at point x is: -10.0
```



## שיטת פולינום ניוטון – טבלת הפרשים מחולקים:

נתונים  $n$  נקודות טבלה.

פולינום ניוטון הוא פולינום ממעלה  $n-1$  שעובר דרך כל נקודות הטבלה הנתונות.

נקרא גם טבלת הפרשים מחולקים כי מתחילים בנקודות ה $X_i$  של הטבלה ובנקודות  $f(X_i)$  כלומר ערכי ה $y$  של נקודות הטבלה ומחשבים לפי הנוסחה של  $f[x_0, \dots, x_N]$  את המקדמים של הפולינום של ניוטון. אחרי מציאת המקדמים נמצא את הפולינום.

קלט:

האלגוריתם לקבלת פולינום ניוטון מקבל כקלט:

1.  $N$  נקודות טבלה, עם ערכי  $X_i$  ו $Y_i$ .
2. נקודה  $x=x_0$  עבורה רוצים לחשב את ערך הפולינום (בנקודה  $x_0$ ).

פלט:

האלגוריתם למציאת פולינום ניוטון מחזיר כפלט את הפולינום שהוא מצא (כלומר, מחזיר פונקציה), פולינום ממעלה  $n-1$  עבור  $n$  נקודות טבלה. את הערך  $x_0$  שקיבלנו מהמשתמש נציב בפולינום (פונקציה) שהוחרה לקבלת ערך הפולינום בנקודה הזאת. התוכנית תדפיס את ערך הפולינום בנקודה  $x_0$  שקיבל.

מדריך למשתמש:

בשלב ראשון, ניצור  $n$  נקודות טבלה עם ערכי  $(x,y)$ . לקבלת פולינום ניוטון ממעלה  $n-1$ . לאחר מכן, נכניס את הערך  $x_0$ . לקבלת ערך הפולינום בנקודה הזאת.

התוכנית תבקש מהמשתמש את מספר נקודות הטבלה  $n$ , ואז המשתמש יכניס  $n$  נקודות טבלה. לאחר מכן המשתמש יכניס ערך נקודה  $x_0$ . לקבלת ערך פולינום ניוטון בנקודה.

```
C:\Python34\python.exe C:/Users/ליאור/PycharmProjects/Hackaton2/newton_approx.py
Enter number of table points=
3
Enter table points:
x=2
y=7
Enter table points:
x=1
y=3
Enter table points:
x=0
y=1
Enter the value of X0:
5
The value of the polynomial at point x is: 31.0
```

## שיטת השליש של סימפסון:

שיטת סימפסון היא שיטה לחישוב אינטגרל של פונ'  $f$  חד מימדית בתחום  $[a, b]$  ע"י פישוט הפונ' לפולינום ריבועי כלשהוא העובר בין נקודות  $[(a, f(a)), (\frac{a+b}{2}, f(\frac{a+b}{2})), (b, f(b))]$ , עליו קל יותר לחשב אינטגרל (בצורה אנליטית). אם נייצר את הפולינום לפי לגרנז' ונחלק את חישוב האינטגרל ל- $n$  קטעים שווים בעלי מרחק  $h = \frac{b-a}{n}$  פתרון האינטגרל לפי שיטת סימפסון יהיה:

$$\int_a^b f(x) dx = \frac{h}{3} (f(a) + f(b) + \sum_{\substack{0 < i < n \\ i \text{ גוז'א } i}} 4f(a + i * h) + \sum_{\substack{0 < i < n \\ i \text{ גוז'א } i}} 2f(a + i * h))$$

קוד:

קלט:

פונ'  $f$  מוגדרת ורציפה בתחום  $[a, b] \rightarrow R$   
tuple  $dom$  המורכב מ-  $(a, b)$  מייצג את טווח האינטגרל  
מספר  $n$  זוגי מייצג את מספר החלקים אליו יחולק הקטע  $[a, b]$ . אם  $n$  אי-זוגי המספר יועגל למספר הזוגי הראשון הגדול מ- $n$

פלט:

מספר המהווה קירוב של האינטגרל  $\int_a^b f(x) dx$

```
simpson_third_integral.py x
1 def simpson_third(f,dom,n=2):
2     """
3     f = callable that returns a single numeric value, the function that we want to find the integral of.
4     dom = the integral domain
5     n = number of chunks to perform the integration on. must be positive and devisable by 2.
6     """
7     a,b = dom
8     if n%2!=0: n+=1
9     if b<a: a,b = b,a
10    h = (b-a)/n
11    x = lambda i: a+i*h
12    sum = f(a) + f(b)
13    for i in range(1,n):
14        sum+= 4*f(x(i)) if i%2!=0 else 2*f(x(i))
15    return h*sum/3
```

## דוגמאות נכונות:

```
17 if __name__=="__main__":
18     from numpy import log as ln
19     i=lambda x: x*ln(x) -x
20     print(f"integral of ln(x):
21         using simpson: {simpson_third(lambda x:ln(x),(1,3),n=4)}
22         analitic method: {i(3)-i(1)}
23     ")
24
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\Pavel\Desktop\Project\NumAnalytics> & python c:/Users/Pavel/Desktop/Project/NumAnalytics/integration/simpson\_third\_integral.py

integral of ln(x):

using simpson:	1.2953216682862132
analitic method:	1.2958368660043291

## שיטת ספליין קובי:

שיטת ספליין קובי היא שיטה לאינטרפולציה של פונ'  $f$  רציפה בנגזרת ראשונה ושנייה על סמך דגימות  $(x, f(x))$  ואופציאונלית- התנהגות (נגזרת ראשונה) הפונ' בנקודות הקצה. בהינתן  $n$  דגימות השיטה מייצרת  $n-1$  פולינומים  $c_i$  ממעלה שלישית ע"י אינטרפולצית לגרנז' שמקיימים:  $c'_i(x_i) = c'_{i-1}(x_i)$  כאשר כל  $x_i$  מהווה נקודת חיבור בין טווחי המקור של  $c_{i-1} - | c_i$  (פרט לקצוות, כנ"ל לגבי נגזרת שנייה).

קוד:

קלט:

רשימת דגימות data מורכבת מאוסף tuples  $(x, f(x))$  פונקצייה  $\text{solver}(\text{Matrix } A, \text{Vector } b) \rightarrow \text{Vector } x$  פותרת מטריצות מהצורה  $A\bar{x} = \bar{b}$  bounds tuple מכיל את המידע על נגזרות הקצוות:  $(f''(\min\{x\}), f''(\max\{x\}))$  כאשר  $x$  שייך לדגימות.

פלט:

פונ'  $f$  שעונה לדרישות רציפות בנגזרת הראשונה והשנייה, שמוגדרת רק לטווח  $(\min\{x\}, \max\{x\})$  כאשר  $x$  שייך לדגימות.

קוד:

```

def cubic_spline(data, solver = gauss, **kwargs):
    """given a list of tuples representing (x,y) point create a "smooth" function where the first and second derivatives are continuous along the graph
    possible kwargs:
        bounds= 2 item tuple containing edge case first derivatives, example bounds=(-5,5) for the dataset [(x=1,y=?),(x=2,y=?),...,(x=10,y=?)] the resulting smooth functions f'(1)=-5, f'(10)=5
    """
    def create_h():
        """creates an array h that each h[i]=x[i]-x[i-1]"""
        return [None] + [abs(data[i].x - data[i-1].x) for i in range(1,n)] # array of distances

    def create_matrix(bounds = None):
        """
        creates the matrix
        """
        lm = [0] * n
        mu = [0] * n
        matrix = numpy.zeros((n,n))
        for i in range(1,n-1):
            lm[i] = 2 * h[i]
            mu[i] = h[i] * h[i+1]
            matrix[i,i-1] = -h[i]
            matrix[i,i+1] = h[i]
        if bounds:
            lm[0] = bounds[0]
            lm[n-1] = bounds[1]
        matrix[0,0] = lm[0]
        matrix[n-1,n-1] = lm[n-1]
        return numpy.matrix(matrix)

    def create_d(bounds = None):
        """creates the results matrix d where each d[i] = 6*divided_difference(x[i-1],x[i],x[i+1]) except for edge cases.
        bounds represent the known first derivative of the edge cases, and are calculated accordingly
        """
        d0 = (0 if not bounds else (divided_diff(data[:2]) - bounds[0])/h[1])
        dn = (0 if not bounds else (divided_diff(data[-2:]) - bounds[1])/h[n-1])
        d = [d0] + [divided_diff(data[i-1:i+2]) for i in range(1,n-1)] + [dn]
        d = [6*x for x in d]
        return d

    def create_polynoms(m):
        """create a list containing polynomials for each range of points, ie: c[i] is the part function for calculating f(x) for x[i-1]<=x[i]"""
        def func(roots):
            """given n roots x1,x2,...xn creates the polynomial (x-x1)*(x-x2)*...*(x-xn)"""
            return numpy.poly1d(numpy.poly(roots))

        def create_c(i):
            """given index i create Lagrange polynomial compliant with found derivatives m[], returns a numpy poly1d
            """
            prod1 = (m[i-1]*(-1*func([data[i].x]))**3)/(6*h[i])
            prod2 = (m[i]*func(data[i-1].x)**3)/(6*h[i])
            prod3 = (data[i-1].y - (m[i-1]*h[i]**2))/6 * (-1*func([data[i].x])/h[i])
            prod4 = (data[i].y - (m[i]*h[i]**2))/6 * (func(data[i-1].x)/h[i])
            return prod1 + prod2 + prod3 + prod4

        return [None] + [create_c(i) for i in range(1,n)]

    data = [Point(p) for p in data]
    data.sort(key=lambda elem:elem.x)
    n = len(data)
    h = create_h()
    matrix = create_matrix(**kwargs)
    d = create_d(**kwargs)
    d = [[x] for x in d]
    m = solver(matrix,d)
    c = create_polynoms(m)

    ranges = [(data[i].x,data[i+1].x) for i in range(n-1)]
    functions = c[1:]

    return route_function(ranges,functions)

def graph(f, points = None, precision = 0.001):
    try:
        import matplotlib.pyplot as pyplot
    except:
        print("cannot import matplotlib")
        return

    getx = lambda elem:elem[0]
    rng = (getx(min(points, key = getx)),getx(max(points, key = getx)))
    x=numpy.arange(rng[0],rng[1]+precision,precision)
    y=[f(i) for i in x]
    pyplot.plot(x,y)
    if points!=None:
        px = [i[0] for i in points]
        py = [i[1] for i in points]
        pyplot.scatter(px, py)
    pyplot.show()

```

## דוגמאות נכונות:

```
data.sort(key=lambda elem:elem.x)
n = len(data)
h= create_h()
matrix = create_matrix(**kwargs)
d = create_d(**kwargs)
d = [[x] for x in d]
m = solver(matrix,d)
c = create_polynoms(m)

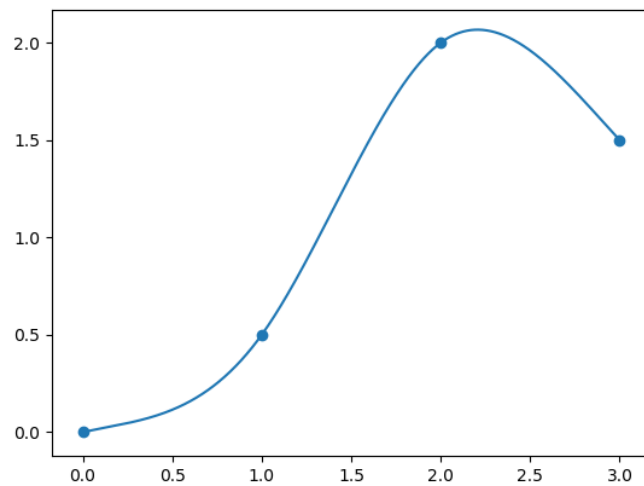
ranges = [(data[i].x,data[i+1].x) for i in range(n-1)]
functions = c[1:]

return route_function(ranges,functions)

def graph(f, points = None, precision = 0.001):[]

def examples():
    data = [(0,0),(1,0.5),(2,2),(3,1.5)]
    cubic = cubic_spline(data = data,bounds=[0.2,-1])
    graph(cubic,points=data)
#     data = [(5,10),(-5,-1.5),(1,3),(2,2.5)]
#     cubic = cubic_spline(data = data)
#     graph(cubic,points=data)

if __name__ == "__main__":
    examples()
```



## Romberg Integration

### תנאי מוקדם: ריצ'רדסון אקסטרפולציה

ריצ'רד ריצ'רדסון אינו משמש רק כדי לחשב קרובים מדויקים יותר של נגזרת אלא משמש גם בבסיס לאינטגרציה נומרית הנקראת אינטגרציה של רומברג, היא בעיקר שיטה פשוטה להגברת הדיוק.

הנחות:

- יש לנו אמצעי משוער של חישוב כמות מסוימת  $G$ .
- התוצאה תלויה בפרמטר  $h$  כך שהקירוב יתקבל על ידי  $g(h)$  כאשר  $G = g(h) + E(h)$ .

השלבים:

- נבצע חישוב, נתונים  $h_1$  וגם  $h_2$ :  $G = g(h_1) + Ch_1^p$  וגם  $G = g(h_2) + Ch_2^p$ .
- נפתור עבור  $G$ : המטרה למחוק  $E(h) = Ch^p$ ,  $\frac{h_1^p g(h_2) - h_2^p g(h_1)}{h_1^p - h_2^p} = G$ .
- מבלי להיכנס לאלגברה, ריצ'רדסון אקסטרפולציה כללי:  $G = \frac{2^p g(\frac{h_1}{2}) - g(h_1)}{2^p - 1}$ .

### שיטה: אינטגרציה רומברג

אינטגרציה רומברג משלבת את הכלל טרפז משולב עם אקסטרפולציה ריצ'רדסון.

$$\text{Let } R_{i-1} = I_i$$

- Starts with the computation of one panel and two panels via Composite Trapezoidal Rule

$$R_{1,1} = I_1 \text{ (one panel)}$$

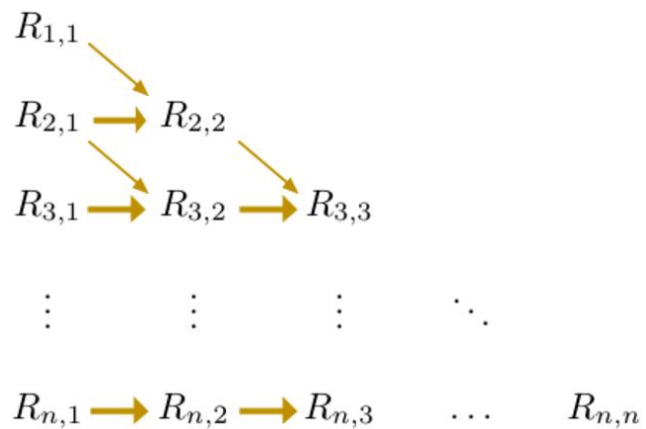
$$R_{2,1} = I_2 \text{ (two panels)}$$

- We get the leading error term  $C_1 h^2$  then eliminate by Richardson Extrapolation using  $p = 2$  (since that is the exponent of the error)

$$\begin{aligned} R_{2,2} &= \frac{2^2 R_{2,1} - R_{1,1}}{2^2 - 1} \\ &= \frac{4}{3} R_{2,1} - \frac{1}{3} R_{1,1} \end{aligned}$$

- Get the leading error term  $C_2 h^4$  then eliminate by Richardson Extrapolation using  $p = 4$
- We do this until it converges to the wanted solution

מה שנקבל יראה :



זה אומר לנו שאנחנו צריכים לחשב איפה שני החצים המצביעים עם חץ המדרגה לאותה נקודה.

(למשל  $R_{1,1}$ ,  $R_{2,1}$  מבצעים על  $R_{2,2}$ )

האומדן המדויק ביותר של האינטגרל הוא תמיד המונח האלכסון האחרון של המערך.

תהליך זה נמשך עד שההבדל בין שני מונחים אלכסוניים רצופים הופך להיות קטן למדי.

**נוסחה הכללית :**

### General Romberg Formula

$$R_{i,j} = \begin{cases} R_{i,j} = CTR(h_i) & : \text{if } i = 1 \\ R_{i,j} = \frac{4^{j-1}R_{i,j-1} - R_{i-1,j-1}}{4^{j-1} - 1} & : \text{if } i > 1 \end{cases}$$

\*note:  $h = \frac{b-a}{2^{i-1}}$  : where  $a$  and  $b$  are the boundaries  
:  $i$  is the index of the row

**:Romberg function:**

**קלט:** מקבל פונקציה וגבולות אינטגרל- $[a,b]$  .

**פלט:** מחזיר אינטגרל ומספר הפגמים (אלכסון) למשל מהדוגמא למעלה  $(R_{1,2})$  שהשתמש.

```
# module_romberg
...

I , nP = romberg(func,a,b,tol=1.0e-6).
Romberg integration of f(x) from x = a to bselfself.
Return the integral and the number of panelssself.

...

import numpy as np
from recursive_trapezoid import trapezoid

def romberg(f, a, b, tol=1.0e-6):

    def richardson(r, k):
        for j in range(k-1, 0, -1):
            const = 4.0**(k-j)
            r[j] = (const*r[j+1]-r[j])/(const-1.0)
        return r

    r = np.zeros(21)
    r[1] = trapezoid(f, a, b, 0.0, 1)
    r_old = r[1]
    for k in range(2, 21):
        r[k] = trapezoid(f, a, b, r[k-1], k)
        r = richardson(r, k)
        if abs(r[1]-r_old) < tol*max(abs(r[1]), 1.0):
            return r[1], 2**(k-1)
        r_old = r[1]
    print("not converge")
```

דוגמת הרצה:

$$\int_0^{\sqrt{\pi}} 2x^2 \cos(x^2) = -0.894831$$

```
31 if __name__=="__main__":
32     try:
33         import math
34         def f(x): return (2.0*(x**2)*math.cos(x**2))
35         I, n = romberg(f, 0, (math.pi)**0.5)
36         print(f"""
37         integral = {I}
38         numEvals = {n}
39         """)
40     except:
41         pass
42
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

[Running] python -u "d:\Programs\Programming\Programming Projects\Python Projects\NumAnalytics\integration\romberg\_method.py"

```

integral = -0.8948314695044126
numEvals = 64
```