אנליזה נומרית תשע"ט

דוח סיכום האקתון 1

קבוצה מס' : 20

חברי הקבוצה: אלמוג מכלוף, אלון גבאי, אלעד מטודי, יריב גארלה, טום צייגר.

1. הגדרת הבעיה – הערכת רמת הקרינה של אזור מזוהם ע"י ניטור מידע מהאוויר.
עקב הגבלות מתמטיות לא ניתן לספק תמונה תלת-ממדית של רמת הזיהום בענן הרדיואקטיבי, אך ניתן לספק תמונה דו ממדית מספיק מדויקת של התפלגות הזיהום בקרקע בעזרת מדידת השדה הרדיואקטיבי בגובה בין 100 ל500 מטר.
לפתרון בעיה זו אנו נעזר בשיטת המשלים ובשיטת העודף.

1.1 שיטת משלים – בשיטה זו אנו נמפה את האזור המזוהם בעזרת איזוטופים רדיואקטיביים. מסוק מרחף מעל השטח המזוהם, מחלק את השטח לרשת דמיונית בגודל NxN ובעצם יוצר מעין מטריצה שמרכז כל תא שלה מייצג את נקודות הרשת ומהווה אזור פעילות הומוגנית ואיזוטופית. בנוסף, כל מרכז תא מייצג נקודת ייחוס שתשמש בחישוב כמרכז הזיהום.
היחס בין הקרינה שנמדדה ובין השטח המזוהם ניתן ע"י מספר משוואות לינאריות וקבועים c, k ו- u.
מציאת הקבועים הנ"ל תתואר בהמשך בצורה מפורטת.

כל איבר ב-D מתקבל ע"י הנוסחא הבאה:

$$D_{ij} = (c(1 + k * R_{ih}) * e^{-u*Rij}) / R^2_{ij}$$

הגדלים אותם אנו נדרשים לספק ליישום הם:

D – מקדם המטריצה ממשוואה 3.

C – וקטור העוצמות(נעלם).

M – וקטור הערכים שנמדדו.

2. השיטה והצגת הכלים לפתרון – לפתרון הבעיה השתמשנו בשיטת החציה ושיטת המיתר. את השיטות ועוד ניתן למצוא בקישור הבא:

https://github.com/yariv1025/Numerical-Analysis-Project

למעט הקוד Plot_it המשמש להדפסת גרפים שאותו שכתבנו כך שיתאים לצרכינו, כל הקודים הקיימים בקישור הנ"ל נכתבו על ידינו.

3. הצגת הנתונים – בעזרת שיטת החצייה ושיטת המיתר השגנו את השורשים עבור הפונקציה הנתונה:

$$g(x) = 16X^3 - 16X^2 + 1$$

- **שיטת החצייה** היא אלגוריתם למציאת שורש של פונקציה, אשר עושה שימוש איטרטיבי בחלוקת האינטרוול לשניים וכך מאפשר לבחור מרווח קטן יותר שבו השורש נמצא. תהליך זה נמשך עד שהפער מספיק קטן.

<u>הדרישות עבור שיטת החצייה הן:</u>

פונקציה רציפה בקטע סגור [a,b] כך שיתקיים f(a)*f(b)<0 ובעצם יהיה שורש C עבורו f(c) = 0

<u>תנאי עצירה:</u>

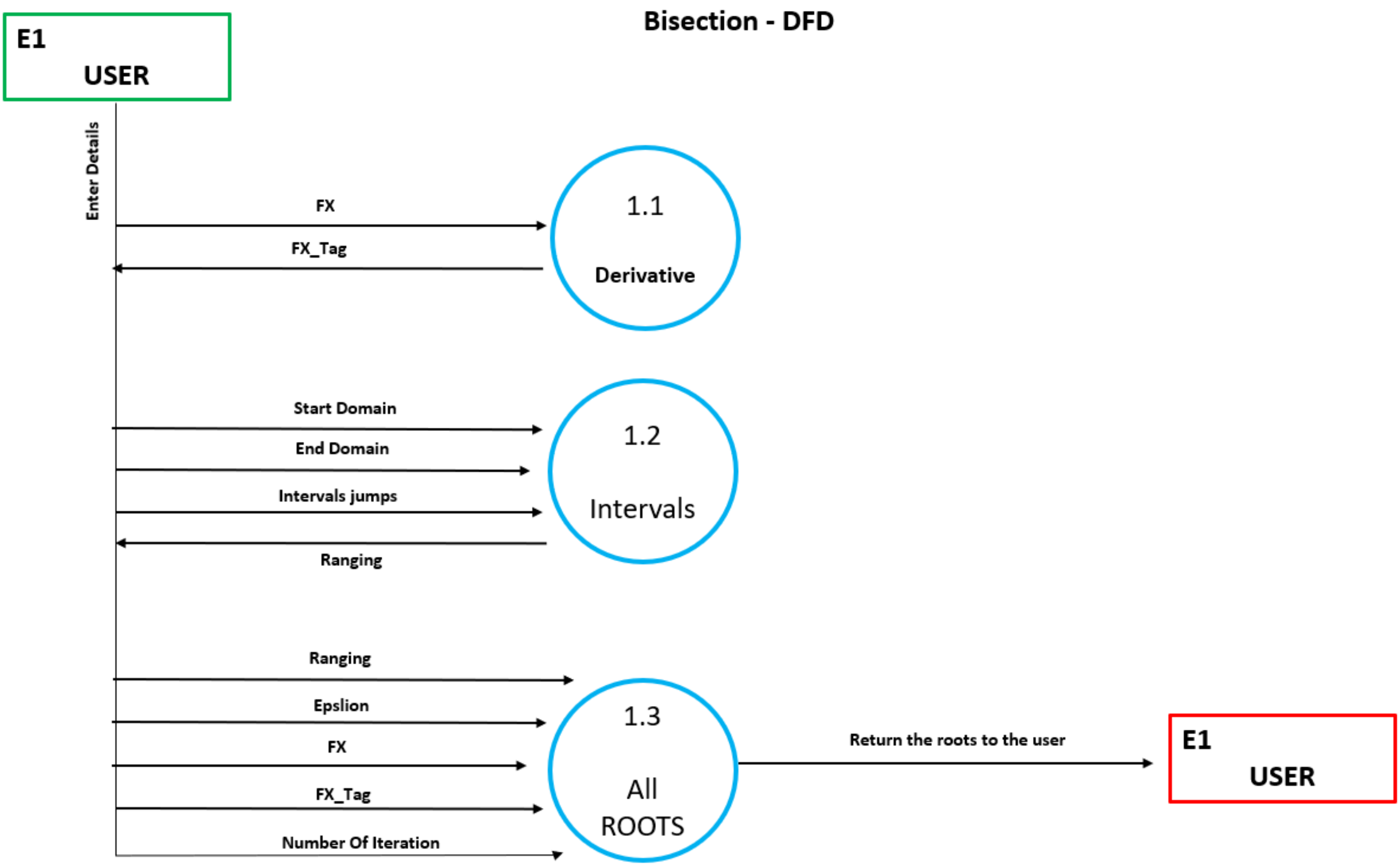
1. אורך תת קטע או מרחק בין שני ניחושים אחרונים.
2. מספר איטרציות

<u>יתרונות השיטה:</u>

1. מתכנסת בקצב קבוע
2. ניתן לחשב מראש מספר איטרציות לקבלת דיוק רצוי

<u>חסרונות השיטה:</u>

1. התכנסות איטית
2. צריך למצוא מראש קטע בו הפונקציה מחליפה סימן
3. שימושי לפונקציות רציפות בלבד ומחליפות סימן


Bisection - DFD
E1
USER
Enter Details
FX
FX_Tag
1.1
Derivative
Start Domain
End Domain
Intervals jumps
Ranging
1.2
Intervals
Ranging
Epslion
FX
FX_Tag
Number Of Iteration
1.3
All
ROOTS
Return the roots to the user
E1
USER

הפונקציה Bisection מכילה בתוכה מתודות שונות כאשר כל מתודה כזו אחראית על פעולה שונה:

ב-main אנו מגדירים את האפסילון, מספר האיטרציות, הטווח, הדילוגים והפונקציה הנתונה.

Bisection.py

```python

if __name__ == '__main__':
    x = Symbol('x')
    epsilon = 0.00000001
    iteration = 100
    Start_Domain = -10
    End_Domain = 10
    interval_jump = 0.5
    # fx=ln(x**2-2*x)+cos(x**3-1)+exp(2*(x**2)-3*x+4) syntx example
    # fx = (sin(x)+(ln(x)*cos(x))) example of syntx writing
    fx = 16*x**3-16*x**2+1
    tmp1 = f(fx)
    fx_tag = derivative(fx)
    fx = tmp1


```

אנו משתמשים במתודה **def** intervals(Start_Domain, End_Domain, interval_jump): ליצירת רשימת נקודות (אינטרוולים) כך שנוכל לבדוק את הפונקציות שלנו בנקודות האלו.

אנו גוזרים את הפונקציה ע"י המתודה: **def** derivative(symbolic_fuction): מכיוון שאנו לוקחים בחשבון שקיימת אפשרות לריבוב זוגי כך שיתכנו שורשים של הפונקציה שיתפספסו במהלך הבדיקה, כלומר הבדיקה עלולה שלא להביא את כל השורשים הקיימים ולכן נציב את השורשים של הנגזרת בפונקציה המקורית ע"י שימוש במתודה נוספת והיא: **def** all_roots(ranging, fx, fx_tag, iteration, epsilon): כדי לראות האם פספסנו שורש כלשהו, זאת כדי להוסיף אותו לרשימת השורשים המקורית.

המתודה: **def** all_roots(ranging, fx, fx_tag, iteration, epsilon): משתמשת במתודה **def** roots(ranging, function, iteration, epsilon): האחראית למעבר על האינטרוולים וקריאה למתודה **def** bisection(a, b, iteration, function, epsilon): האחראית למציאת השורשים הקיימים ע"י ביצוע הא לגוריתם של שיטת החציה.

להלן תוצאות היישום:

```
number iteration: 1
Approximation: -0.1875
number iteration: 2
Approximation: -0.21875
number iteration: 3
Approximation: -0.234375
number iteration: 4
Approximation: -0.2265625
number iteration: 5
Approximation: -0.22265625
number iteration: 6
Approximation: -0.224609375
number iteration: 7
Approximation: -0.2255859375
number iteration: 8
Approximation: -0.22607421875
number iteration: 9
Approximation: -0.225830078125
number iteration: 10
Approximation: -0.2257080078125
number iteration: 11
Approximation: -0.22576904296875
number iteration: 12
Approximation: -0.225799560546875
number iteration: 13
Approximation: -0.2258148193359375
number iteration: 14
Approximation: -0.22580718994140625
number iteration: 15
Approximation: -0.22580337524414062
number iteration: 16
Approximation: -0.2258014678955078
number iteration: 17
Approximation: -0.22580242156982422
number iteration: 18
Approximation: -0.22580289840698242
number iteration: 19
Approximation: -0.22580313682556152
number iteration: 20
Approximation: -0.22580301761627197
number iteration: 21
Approximation: -0.2258029580116272
number iteration: 22
Approximation: -0.22580298781394958
number iteration: 23
Approximation: -0.2258029729127884
number iteration: 24
Approximation: -0.225802980363369
```

```
number iteration: 25
Approximation: -0.2258029840886593
number iteration: 0
Approximation: 0.375
number iteration: 1
Approximation: 0.3125
number iteration: 2
Approximation: 0.28125
number iteration: 3
Approximation: 0.296875
number iteration: 4
Approximation: 0.3046875
number iteration: 5
Approximation: 0.30078125
number iteration: 6
Approximation: 0.298828125
number iteration: 7
Approximation: 0.2978515625
number iteration: 8
Approximation: 0.29833984375
number iteration: 9
Approximation: 0.298583984375
number iteration: 10
Approximation: 0.2984619140625
number iteration: 11
Approximation: 0.29852294921875
number iteration: 11
Approximation: 0.29852294921875
number iteration: 12
Approximation: 0.298492431640625
number iteration: 13
Approximation: 0.2984771728515625
number iteration: 14
Approximation: 0.29848480224609375
number iteration: 15
Approximation: 0.2984809875488281
number iteration: 16
Approximation: 0.29848289489746094
number iteration: 17
Approximation: 0.29848384857177734
number iteration: 18
Approximation: 0.29848432540893555
number iteration: 19
Approximation: 0.29848408699035645
number iteration: 20
Approximation: 0.298484206199646
number iteration: 21
Approximation: 0.2984841465950012
number iteration: 22
Approximation: 0.29848411679267883
number iteration: 23
Approximation: 0.29848413169384
```

```
number iteration: 12
Approximation: 0.298492431640625
number iteration: 13
Approximation: 0.2984771728515625
number iteration: 14
Approximation: 0.29848480224609375
number iteration: 15
Approximation: 0.2984809875488281
number iteration: 16
Approximation: 0.29848289489746094
number iteration: 17
Approximation: 0.29848384857177734
number iteration: 18
Approximation: 0.29848432540893555
number iteration: 19
Approximation: 0.29848408699035645
number iteration: 20
Approximation: 0.298484206199646
number iteration: 21
Approximation: 0.2984841465950012
number iteration: 22
Approximation: 0.29848411679267883
number iteration: 23
Approximation: 0.29848413169384
number iteration: 0
Approximation: 0.875
number iteration: 1
Approximation: 0.9375
number iteration: 2
Approximation: 0.90625
number iteration: 3
Approximation: 0.921875
number iteration: 4
Approximation: 0.9296875
number iteration: 5
Approximation: 0.92578125
number iteration: 6
Approximation: 0.927734375
number iteration: 7
Approximation: 0.9267578125
number iteration: 8
Approximation: 0.92724609375
number iteration: 9
Approximation: 0.927490234375
number iteration: 10
Approximation: 0.9273681640625
number iteration: 11
Approximation: 0.92730712890625
number iteration: 12
Approximation: 0.927337646484375
```

```
number iteration: 13
Approximation: 0.9273223876953125
number iteration: 14
Approximation: 0.9273147583007812
number iteration: 15
Approximation: 0.9273185729980469
number iteration: 16
Approximation: 0.9273204803466797
number iteration: 17
Approximation: 0.9273195266723633
number iteration: 18
Approximation: 0.9273190498352051
number iteration: 19
Approximation: 0.927318811416626
number iteration: 20
Approximation: 0.9273189306259155
number iteration: 21
Approximation: 0.9273188710212708
number iteration: 22
Approximation: 0.9273188412189484
number iteration: 23
Approximation: 0.9273188263177872
number iteration: 24
Approximation: 0.9273188337683678
number iteration: 25
Approximation: 0.9273188374936581
number iteration: 0
Approximation: 0.625
number iteration: 1
Approximation: 0.6875
number iteration: 2
Approximation: 0.65625
number iteration: 3
Approximation: 0.671875
number iteration: 4
Approximation: 0.6640625
number iteration: 5
Approximation: 0.66796875
number iteration: 6
Approximation: 0.666015625
number iteration: 7
Approximation: 0.6669921875
number iteration: 8
Approximation: 0.66650390625
number iteration: 9
Approximation: 0.666748046875
number iteration: 10
Approximation: 0.6666259765625
number iteration: 11
Approximation: 0.66668701171875
number iteration: 12
Approximation: 0.666656494140625
```

```
number iteration: 12
Approximation: 0.666656494140625
number iteration: 13
Approximation: 0.6666717529296875
number iteration: 14
Approximation: 0.6666641235351562
number iteration: 15
Approximation: 0.6666679382324219
number iteration: 16
Approximation: 0.6666660308837891
number iteration: 17
Approximation: 0.6666669845581055
number iteration: 18
Approximation: 0.6666665077209473
number iteration: 19
Approximation: 0.6666667461395264
number iteration: 20
Approximation: 0.6666666269302368
number iteration: 21
Approximation: 0.6666666865348816
number iteration: 22
Approximation: 0.6666666567325592
number iteration: 23
Approximation: 0.6666666716337204
number iteration: 24
Approximation: 0.6666666641831398
number iteration: 25
Approximation: 0.6666666679084301
[-0.2258029840886593, 0.2984841428697109, 0.9273188374936581]

Process finished with exit code 0
```

**שיטת המיתר - שיטת אינטרפולציה ליניארית** היא שיטה איטרטיבית למציאת שורשים של פונקציה רציפה של משתנה אחד.

שיטה זו דומה לשיטת ניוטון-רפסון למציאת שורשים. בשיטת ניוטון-רפסון יש צורך בנגזרת הפונקציה בנקודה האחרונה. שיטת המיתר מקרבת את הנגזרת על ידי שיפוע המיתר המחבר את שתי הנקודות האחרונות שחושבו, ומכאן שמה.

סדר ההתכנסות של השיטה נמוך יותר מזה של שיטת ניוטון-רפסון, בשיטת ניוטון-רפסון הסדר הוא 2, ואילו בשיטת המיתר הסדר הוא יחס הזהב (1.618 לערך).

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.

<u>הדרישות עבור שיטת המיתר הן:</u>

1. שני ניחושים, oX ו- X1

<u>תנאי עצירה:</u>

1. אם הניחוש הבא פחות הניחוש העכשווי קטן מאפסילון

<u>יתרונות השיטה:</u>

1. במידה והשיטה מתכנסת, השיטה מתכנסת בצורה מהירה יותר משיטות ליניאריות.

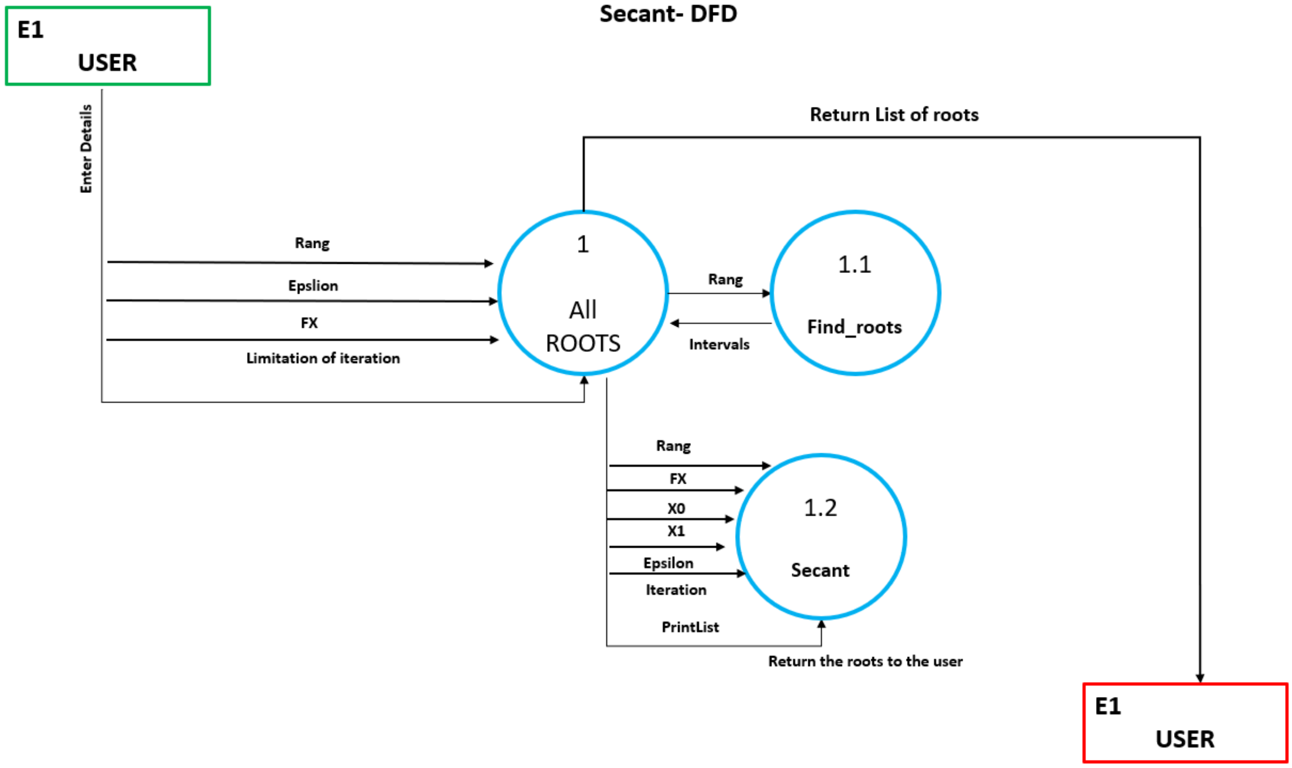
אפשר להראות כי סדר ההתכנסות של השיטה הוא: $\frac{1}{2}\left(1+\sqrt{5}\right)\approx 1.618$

<u>חסרונות השיטה:</u>

1. השיטה יכולה שלא להתכנס

2. חלוקה באפס מתאפשרת כאשר שיפוע של ישר משיק בנקודת האפס של פונקציה קרוב לאפס.

**Secant- DFD**


E1
USER
Enter Details
Return List of roots
Rang
Epslion
FX
Limitation of iteration
1
All
ROOTS
Rang
Intervals
1.1
Find_roots
Rang
FX
X0
X1
Epsilon
Iteration
PrintList
1.2
Secant
Return the roots to the user
E1
USER

הפונקציה Secant מכילה בתוכה מתודות שונות כאשר כל מתודה כזו אחראית על פעולה שונה:

ב-main אנו מגדירים את האפסילון, מספר האיטרציות, הטווח, הדילוגים והפונקציה הנתונה.

Secant.py

```python
    if len(roots) == 0:
        return None
    return roots

if __name__ == '__main__':
    x = Symbol('x')
    #x*exp(-x)-0.25
    gx = 16*x**3-16*x**2+1
    fx = func(gx)
    print(all_roots(fx, 0.0001, [-1,3], 100))
```

if __name__ == '__main__'

אנו משתמשים במתודה **def** all_roots(fx, eps, rangex, itration) אשר תפקידה הוא להחזיר לנו את השורשים הקיימים בפונקציה. מתודה זו עושה שימוש במתודה נוספת והיא **def** find_roots(rangex)

תפקידה של מתודה זו הוא להחזיר רשימת אינטרוולים למציאת השורשים הסופיים אל המתודה שקראה לה.

לאחר חזרת רשימת האינטרוולים המתודה **def** all_roots(fx, eps, rangex, itration) קוראת למתודה

**def** secant(rangex, fx, x0, x1, eps, itration,printList) שמבצעת את האלגוריתם של שיטת המיתר על אינטרוול בודד. אומנם שיטת המיתר בכל אינטרוול עלולה לחרוג ממנו אך בדיקת כל האינטרוולים שואלת האם השורש שנמצא באינטרוול הינו בתחום. המתודה לבסוף תחזיר שורש או None .

במידה והשורש בתחום הרצוי הפונקציה תוסיף אותו לרשימת השורשים root אחרת הפונקציה תחזיר None.

להלן תוצאות היישום:

```
number iteration: 0
Approximation: -0.3717948717948718
number iteration: 1
Approximation: -0.31129861749349813
number iteration: 2
Approximation: -0.2488463893377426
number iteration: 3
Approximation: -0.23022077447195774
number iteration: 4
Approximation: -0.22606946704785266
number iteration: 5
Approximation: -0.2258062148964642
number iteration: 6
Approximation: -0.22580298386721256
number iteration: 0
Approximation: 0.3048780487804878
number iteration: 1
Approximation: 0.2983434947659036
number iteration: 2
Approximation: 0.29848387400312804
number iteration: 3
Approximation: 0.29848414163061
number iteration: 0
Approximation: 0.8771428571428572
number iteration: 1
Approximation: 0.9017544712667296
number iteration: 2
Approximation: 0.9309995545697436
number iteration: 3
Approximation: 0.9270776657687972
number iteration: 4
Approximation: 0.9273166725409996
number iteration: 5
Approximation: 0.9273188411443375
[-0.22580298386721256, 0.29848414163061, 0.9273188411443375]

Process finished with exit code 0
```

**שיטת גאוס זיידל –** היא טכניקה איטרטיבית לפתרון מערכת ריבועית של *n* משוואות ליניאריות עם וקטו
ר נעלמים **X** עבור Ax = b.

למרות שניתן ליישמה לכל מטריצה בעלת איברים שונים מאפס על האלכסון הראשי ,התכנסותה מובטחת
אך ורק אם המטריצה היא אלכסונית דומיננטית, או שהיא סימטרית וחיובית

<u>הדרישות עבור גאוס זיידל הן:</u>

1. אלכסון דומיננטי
2. קיום הפיכות של מטריצה להצבה בנוסחא

<u>תנאי עצירה:</u>

1. אם וקטור הניחוש הבא פחות וקטור הניחוש העכשווי קטן מאפסילון

<u>יתרון השיטה:</u>

1. יתרון שיטה זו הוא בזה שקצב התכנסותה מהיר ובנוסף היא יעילה במקרים של "מטריצות דלילות", כלומר מקצרת זמן ריצה של מציאת הפתרון למטריצות שמרבית איבריה הם קרובים ל0.

<u>חסרון השיטה:</u>

1. השיטה יכולה שלא להתכנס אם אין אלכסון דומיננטי או לא מתקיימת תכונת ההפיכות להצבה בנוסחא.

**שיטת SOR –** היא טכניקה איטרטיבית לפתרון מערכת ריבועית של *n* משוואות ליניאריות עם וקטור נע
למים **X** עבור Ax = b.

למרות שניתן ליישמה לכל מטריצה בעלת איברים שונים מאפס על האלכסון הראשי ,התכנסותה
מובטחת אך ורק אם המטריצה היא אלכסונית דומיננטית, או שהיא סימטרית וחיובית.

בנוסף ניתן לומר כי בהינתן מטריצה ניתן יהיה להפוך אותה לבעלת אלכסון דומיננטי בשונה מגאוס
זיידל.

<u>תנאי עצירה:</u>

1. אם וקטור הניחוש הבא פחות וקטור הניחוש העכשווי קטן מאפסילון

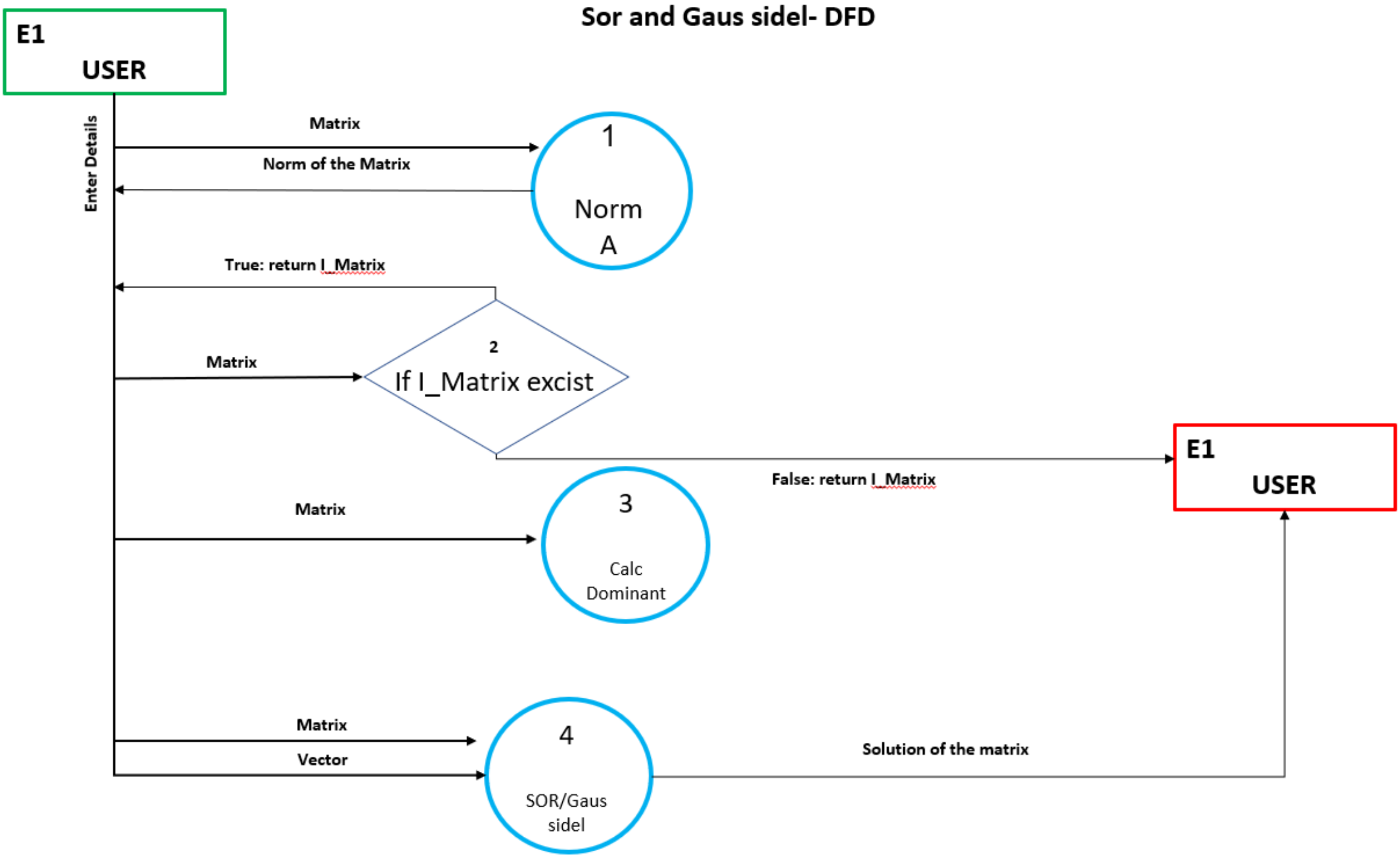
<u>יתרון השיטה:</u>

1. ניתן להפוך כל מטריצה לבעלת אלכסון דומיננטי, כך שכמות האפשרויות למציאת מטריצות שניתן למצוא להם וקטור פתרונות גדלה.

<u>חסרון השיטה:</u>

1. השיטה מתכנסת לאט

**Sor and Gaus sidel- DFD**



הקובץ Gaus sidel and sor מכיל את כל הפונקציות הקשורות לשיטות הנ"ל.

Sor_and_Gaus.py

```python
if __name__ == '__main__':
    a = np.matrix([[2, 7, 4], [10, 2, -5], [3, 4, 9]])
    b = [-7, 3, -3.5]

    norm_A = normMax(a)
    try:
        A_inv = a.I
        norm_A_inv = normMax(A_inv)
        cond = norm_A * norm_A_inv
        print("cond: ", cond)

    except(np.linalg.LinAlgError):
        print("no inverse")

    a = calcDominant(a)
    print(a)

    #print(SOR(a, b))
    print(gaus(a, b))
```

Gaus sidel -הפונקציה Gaus מקבלת מטריצה ווקטור, ומחזירה את ווקטור הפתרונות.

1. תחילה נפרק את המטריצה למטריצות L,D,U . נמצא את המטריצה ההפוכה של L_D

נכפיל את L_D ב-U ונקבל את L_D_U , לאחר מכן נכפיל את D_L_U בווקטור b ורק אז נכפיל את כל ב ב-1-. והכל מתנהל לפי הנוסחה הזו –

$$(D+L)\vec{x}_{r+1} = -U * \vec{x}_r + \vec{b}$$

אנחנו משתמשים בפונקציה iterative שמקבלת

L_D_U_1 – המטריצה שקיבלנו ע"י הנוסחא.

New_b – ווקטור חדש שמוכל מאחדות.

L_D_b – המטריצה L_D שמוכפלת בווקטור b.

PrintList – רשימה ריקה שאמצעותה נדפיס את התוצאות הנכונות של המטריצה.

לאחר כל ביצוע הפונקציה iterative יוחזר לנו כל התוצאות הנכונות של המטריצה שנשלחה בהנחה והמטריצה מתכנסת.

Sor - הפונקציה Sor מקבלת מטריצה ווקטור, ומחזירה את ווקטור הפתרונות.

```python
if __name__ == '__main__':
    a = np.matrix([[2, 7, 4], [10, 2, -5], [3, 4, 9]])
    b = [-7, 3, -3.5]

    norm_A = normMax(a)
    try:
        A_inv = a.I
        norm_A_inv = normMax(A_inv)
        cond = norm_A * norm_A_inv
        print("cond: ", cond)

    except(np.linalg.LinAlgError):
        print("no inverse")

    a = calcDominant(a)
    print(a)

    print(SOR(a, b))
    #print(gaus(a, b))
```

Sor מקבלת מטריצה ווקטור b , תחילת נגדיר w שהוא ייצג לנו את המקדם w בנוסחה.

ניצור D,L,U ושאר המטריצות לחישוב להצבה בנוסחה הבאה –

$$(D + wL)\vec{x}_{r+1} = [(1 - \mathrm{w})\mathrm{D} - \mathrm{wU}]\vec{x}_r + \mathrm{w}\vec{b}$$

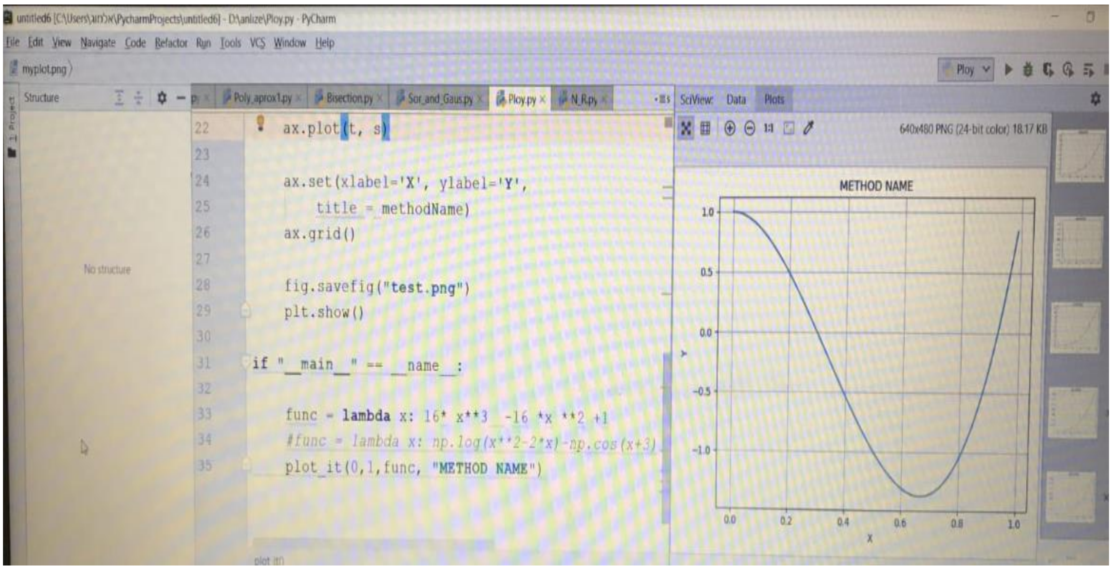
$$0 \leq w \leq 2$$

ונקרא ל iterative שתחזיר לנו את תוצאות המטריצה, לאחר מכן הפונקציה isclose תוודא בכך שהיא תציב את הווקטור שיתקבל מ iterative בכל אחת מן המשוואות שהמטריצה מייצגת.

**<u>שלבי ההאקטון –</u>**

<u>מציאת הקבוע c -</u> תלויה בממוצע השורשים החיוביים של הפונקציה $16x^3 - 16x^2 + 1$
הפונקציה find_c משתמשת בשיטת החציה ובשיטת המייתר בכדי לוודא את נכונות השורשים ומבצעת ממוצע של השורשים החיובים ומחזירה את הערך c .

את הגרף הדפסנו באמצעות המתודה def plot_it(start, end, function, methodName):
המתודה משתמשת בספרייה matplotlib כדי לבצע את השרטוטים.
גרף הפונקציה הינו –



```python
def Find_c(func):
    roots=Find_Roots(func)
    final_roots=[]
    sum = 0
    for n in roots:
        if n>0:
            final_roots.append(n)
            sum+=n
    if(len(final_roots) == 0):
        return "Error"
    return sum / len(final_roots)
```

```
number iteration: 1
Approximation: -0.1875
number iteration: 2
Approximation: -0.21875
number iteration: 3
Approximation: -0.234375
number iteration: 4
Approximation: -0.2265625
number iteration: 5
Approximation: -0.22265625
number iteration: 6
Approximation: -0.224609375
number iteration: 7
Approximation: -0.2255859375
number iteration: 8
Approximation: -0.22607421875
number iteration: 0
Approximation: 0.375
number iteration: 1
Approximation: 0.3125
number iteration: 2
Approximation: 0.28125
number iteration: 3
Approximation: 0.296875
number iteration: 4
Approximation: 0.3046875
number iteration: 5
Approximation: 0.30078125
number iteration: 6
Approximation: 0.298828125
number iteration: 7
Approximation: 0.2978515625
```

```
number iteration: 8
Approximation: 0.29833984375
number iteration: 0
Approximation: 0.875
number iteration: 1
Approximation: 0.9375
number iteration: 2
Approximation: 0.90625
number iteration: 3
Approximation: 0.921875
number iteration: 4
Approximation: 0.9296875
number iteration: 5
Approximation: 0.92578125
number iteration: 6
Approximation: 0.927734375
number iteration: 7
Approximation: 0.9267578125
number iteration: 8
Approximation: 0.92724609375
number iteration: 0
Approximation: 0.625
number iteration: 1
Approximation: 0.6875
number iteration: 2
Approximation: 0.65625
number iteration: 3
Approximation: 0.671875
number iteration: 4
Approximation: 0.6640625
number iteration: 5
Approximation: 0.66796875
```

```
number iteration: 6
Approximation: 0.666015625
number iteration: 7
Approximation: 0.6669921875
number iteration: 8
Approximation: 0.66650390625
number iteration: 0
Approximation: -52.401063022692185
number iteration: 1
Approximation: -44.28866184833888
number iteration: 2
Approximation: -31.820557620370018
number iteration: 3
Approximation: -24.368415874315502
number iteration: 4
Approximation: -18.189429652560026
number iteration: 5
Approximation: -13.688223789175517
number iteration: 6
Approximation: -10.244228427312086
number iteration: 7
Approximation: -7.6592705748300745
number iteration: 8
Approximation: -5.705514481204329
number iteration: 9
Approximation: -4.233851439167912
number iteration: 10
Approximation: -3.1252740170046645
number iteration: 11
Approximation: -2.292067948615462
number iteration: 12
Approximation: -1.6677334808411048
```

```
number iteration: 13
Approximation: -1.2025861925823873
number iteration: 14
Approximation: -0.8595397571982821
number iteration: 15
Approximation: -0.6112176657187176
number iteration: 16
Approximation: -0.437684854306034
number iteration: 17
Approximation: -0.3245274812906652
number iteration: 18
Approximation: -0.26042442124674503
number iteration: 19
Approximation: -0.2331482100300129
number iteration: 20
Approximation: -0.22644933965316227
number iteration: 21
Approximation: -0.22581592854165294
number iteration: 0
Approximation: 0.3048780487804878
number iteration: 1
Approximation: 0.2983434947659036
number iteration: 2
Approximation: 0.29848387400312804
number iteration: 0
Approximation: 0.8771428571428572
number iteration: 1
Approximation: 0.9017544712667296
number iteration: 2
Approximation: 0.9309995545697436
number iteration: 3
Approximation: 0.9270776657687972

number iteration: 4
Approximation: 0.9273166725409996
constant c = 0.61279296875

Process finished with exit code 0
```

מציאת הקבוע k - ערך הפונקציה של הקירוב הפולינומיאלי של הנתונים הבאים –

| X | Y |
|---|---|
| 1 | 10.5 |
| 3 | 6.1 |
| 5 | 3.5 |

(הפולינום שהתקבל – $Fx=0.224x^2 -3.099x+13.374$)

```
polynom: <class 'list'>: [matrix([[0.22498547]]), matrix([[-3.09992736]]), matrix([[13.37493701]])]
```

מן הטבלה x,y למעשה התקבלה מטריצה 3X3 שבה הצבנו את הערכים x ו-y בהתאמה כך שווקטור הנעלמים התקבל לפי שתי שיטות לצורך וידוא נכונות התוצאות, לפי שיטות גאוס זיידל וסור. כך ש - Fx(4.74)=k - ז"א שהוא המקדם k.

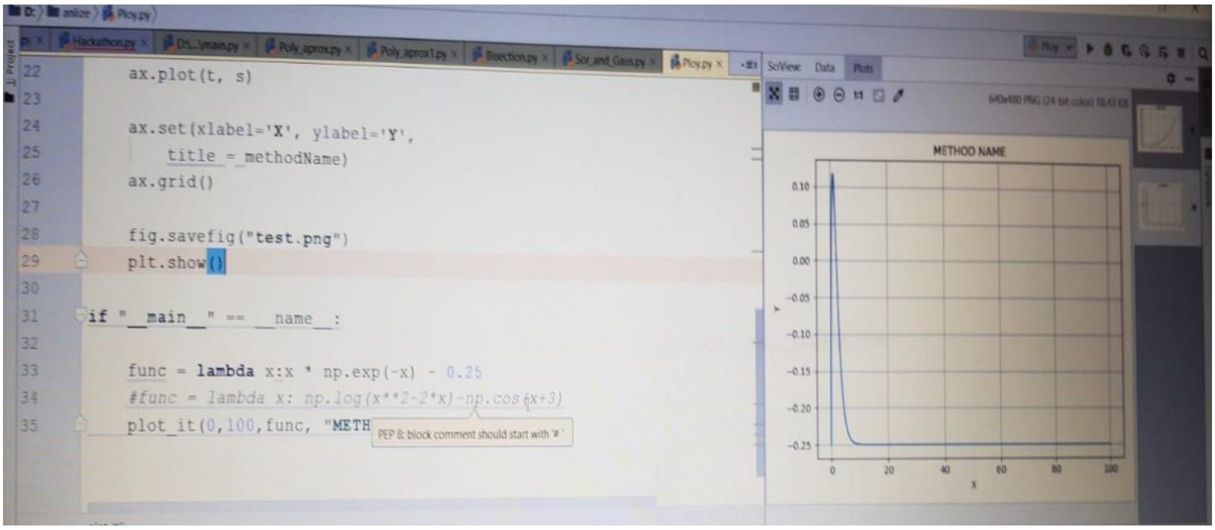
מציאת הקבוע u - הוא מאית השורש הקטן ביותר של הפולינום הבא $Fx=x*e^{-x}-1/4$
לפי שיטות החציה והמיתר קיבלנו שמאית השורש הקטן ביותר של הפולינום הינו :

נתחיל עם שיטת החציה –

```
Bisection.py
if __name__ == '__main__':
    x = Symbol('x')
    epsilon = 0.00000001
    iteration = 100
    Start_Domain = -10
    End_Domain = 10
    interval_jump = 0.5
    # fx=ln(x**2-2*x)+cos(x**3-1)+exp(2*(x**2)-3*x+4) syntx example
    # fx = (sin(x)+(ln(x)*cos(x))) example of syntx writing
    fx = x*exp(-x)-0.25
    tmp1 = f(fx)
    fx_tag = derivative(fx)
    fx = tmp1
```

את הגרף הדפסנו באמצעות המתודה def plot_it(start, end, function, methodName):
המתודה משתמשת בספרייה matplotlib כדי לבצע את השרטוטים.

גרף הפונקציה הינו –




```
number iteration: 0
Approximation: 0.375
number iteration: 1
Approximation: 0.3125
number iteration: 2
Approximation: 0.34375
number iteration: 3
Approximation: 0.359375
number iteration: 4
Approximation: 0.3515625
number iteration: 5
Approximation: 0.35546875
number iteration: 6
Approximation: 0.357421875
number iteration: 7
Approximation: 0.3564453125
number iteration: 8
Approximation: 0.35693359375
number iteration: 9
Approximation: 0.357177734375
number iteration: 10
Approximation: 0.3572998046875
number iteration: 11
Approximation: 0.35736083984375
number iteration: 12
Approximation: 0.357391357421875
```

```
number iteration: 13
Approximation: 2.1533050537109375
number iteration: 14
Approximation: 2.1532974243164062
number iteration: 15
Approximation: 2.1532936096191406
number iteration: 16
Approximation: 2.153291702270508
number iteration: 17
Approximation: 2.153292655944824
number iteration: 18
Approximation: 2.153292179107666
number iteration: 19
Approximation: 2.153292417526245
number iteration: 20
Approximation: 2.1532922983169556
number iteration: 21
Approximation: 2.1532923579216003
number iteration: 22
Approximation: 2.1532923877239227
number iteration: 23
Approximation: 2.1532923728227615
number iteration: 24
Approximation: 2.153292365372181
number iteration: 25
Approximation: 2.1532923616468906
[0.3574029542505741, 2.1532923616468906]
```

וכעת נראה לפי שיטת המיתר –

```python
                roots.append(root)
                Print_roots(printList)
    if len(roots) == 0:
        return None
    return roots

if __name__ == '__main__':
    x = Symbol('x')
    #x*exp(-x)-0.25
    gx = x*exp(-x)-0.25
    fx = func(gx)
    print(all_roots(fx, 0.0001, [-1,3], 100))
```

```
number iteration: 0
Approximation: -0.14816099504223246
number iteration: 1
Approximation: 0.03113422458125717
number iteration: 2
Approximation: 0.22624464505374778
number iteration: 3
Approximation: 0.3165761096115778
number iteration: 4
Approximation: 0.35133540694781723
number iteration: 5
Approximation: 0.3570970223991911
number iteration: 6
Approximation: 0.3574005948706582
number iteration: 7
Approximation: 0.3574029552582999
number iteration: 0
Approximation: 2.2831290828394017
number iteration: 1
Approximation: 2.1545193078735245
number iteration: 2
Approximation: 2.1532792565947223
number iteration: 3
Approximation: 2.1532923651816223
[0.3574029552582999, 2.1532923651816223]
```

התקבלו השורשים הנ"ל – [0.357,2.153]

כך שמאית השורש הקטן מבינהם הוא u = 0.00357.

לייצוג המרחקים בין הגלאי לכל מרכזי התאים ברשת אנו נשתמש בנוסחא:

$$R^2_{I,J} = (X_i - X_j)^2 + (Y_i - Y_j)^2 + Z^2$$

חיבור שתי הנוסחאות הנ"ל יוצר לנו נוסחה חדשה המאפשרת קריאה של הגלאי בתא.

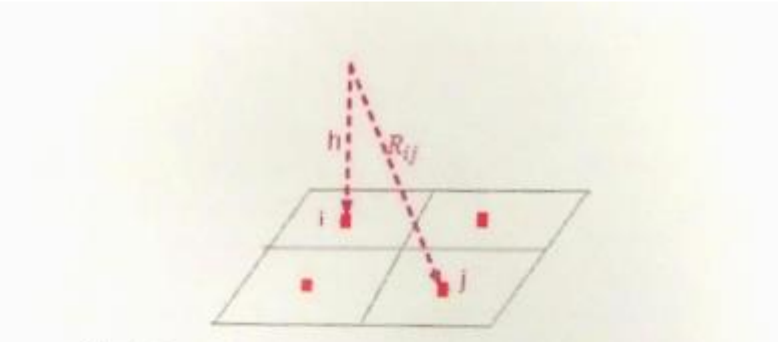


Fig. 1 Sensor location and distances with respect to cells i and j.

כדי למדוד את עוצמת הקרינה הנמצאת במרחק R ממרכז התא אנו נעזר בנוסחה ובקבועים שמצאנו בסעיפים הקודמים c,u,k וכך נחשב את המטריצה D:

$$D_{i,j} = C(1 + k*h)e^{-uR} / R^2$$

<u>המטריצה שהתקבלה היא –</u>

| | | | |
|---|---|---|---|
| 0.00561386612663356 | 0.00412817527959669 | 0.00412817527959669 | 0.00318781756564433 |
| 0.00412817527959669 | 0.00561386612663356 | 0.00318781756564433 | 0.00412817527959669 |
| 0.00412817527959669 | 0.00318781756564433 | 0.00561386612663356 | 0.00412817527959669 |
| 0.00318781756564433 | 0.00412817527959669 | 0.00412817527959669 | 0.00561386612663356 |

כעת נוכל להציב בנוסחה הבאה – DC=M , כאשר M הוא ווקטור הנתון לפי סעיף 6.

על מנת לחשב את ווקטור C שהוא ווקטור הקרינה בכל תא נשתמש בשיטות הנומריות גאוס זיידל וסור.

וקטור C המתקבל הינו –

Main.py | Poly_aprox.py

```python
    step_size=100
    matrix_D = Calculate_D(step_size,c,k,u,h)

    vector_M = [900,950,1000,1100]
    vector_C = Calculate_Vector_C(matrix_D,vector_M)
    print(vector_C)
```

Run: Main

```
x 1 :  122031.62055226578
x 2 :  99
x 3 :  39593.026442826405
number iteration: 24664.062956074937
x 1 :  45273.720648353206
x 2 :  122031.62688606705
x 3 :  100
number iteration: 39593.03454627122
x 1 :  24664.05705392857
x 2 :  45273.71309716623
x 3 :  122031.63238597625
number iteration: 101
x 1 :  39593.04158283725
x 2 :  24664.051928843684
x 3 :  45273.70654014924
[39593.04158283725, 24664.051928843684, 45273.70654014924, 122031.63716178144]
```

<u>וקטור C –</u>

| |
|---|
| 39593.04158283725 |
| 24664.051928843684 |
| 45273.70654014924 |
| 122031.63716178144 |