

Numeric Analysis - Final Project

Finding Equation Roots + Numerical Integration

Question 4

• שאלה מס' 4

עבור הפונקציה הבאה

$$f(x) = \frac{\sin(2x^3 + 5x^2 - 6)}{2e^{-2x}}$$

1. מצאו באמצעות שתי שיטות את כל השורשים האמיתיים הנמצאים בקטע $[-1, 1.5]$
2. השתמשו בקירוב סימפסון ו בשיטת רומברג למציאת הערך של האינטגרל בקטע $[0, 1]$

GitHub: <https://github.com/Maor-Ar/Numerical-Analysis-Hackathon/blob/main/Q4%20Finding%20Equation%20Roots%20Numerical%20Integration.py>

1. שיטות למציאת שורשי משוואה בקטע הנ"ל שיטת ניוטון רפסון, שיטת המיתר
מינוש שיטת המיתר
פונקציית עזר לחולקה למקטעים ובחירה 2 ניחושים, הפונקציה שולחת 2 ניחושים לפונקציה שמוצאת שורש של משוואה לפי שיטת המיתר, הפונקציהבודקת אם השורש לא נמצא בבראייטרציה קודמת וכן אם השורש הוא בטוחה המבוקש.

```
def SecantMethodInRangeIterations(f, check_range, epsilon=0.0001):
    """
    This function find a root to a function by using the secant method by a given list of values to check between.
    :param f: The function (as a python function).
    :param check_range: List of values to check between ; e.g (1,2,3,4,5) it will check between 1-2,2-3,....
    :param epsilon: The tolerance of the deviation of the solution ;
    How precise you want the solution (the smaller the better).
    :return: Returns a list roots by secant method ,
    if it fails to find a solutions in the given tries limit it will return an empty list .
    """
    roots = []
    iterCounter = 0
    for i in check_range:
        startPoint = round(i, 2)
        endPoint = round(i + 0.1, 2)
        print(bcolors.HEADER, "Checked range:", startPoint, "-", endPoint, bcolors.ENDC)
        # Send to the Secant Method with 2 guesses
        local_root = SecantMethod(f, startPoint, endPoint, epsilon, iterCounter)
        # If the root has been found in previous iterations
        if round(local_root, 6) in roots:
            print(bcolors.FAIL, "Already found that root.", bcolors.ENDC)
        # If the root is out of range tested
        elif not (startPoint <= local_root <= endPoint):
            print(bcolors.FAIL, "root out of range.", bcolors.ENDC)
        elif local_root is not None:
            roots += [round(local_root, 6)]
    return roots
```

פונקציה רקורסיבית למציאת שורש של משווהה לפי שיפור הניחוש בכל קרייה רקורסיבית

תנאי עצירה נבחר לפי אפסילון קבוענו או אם כמות הקריאה הרקורסיבית גדולה מ-100 (השיטה לא מתכנסה) האפסילון הנבחר הוא **0.0000001**

```
def SecantMethod(func, firstGuess, secondGuess, epsilon, iterCounter):
    """
    This function find a root to a function by using the SecantMethod method by a given tow guess.
    :param func: The function on which the method is run
    :param firstGuess: The first guess
    :param secondGuess: The second guess
    :param epsilon: The tolerance of the deviation of the solution
    :param iterCounter: number of tries until the function found the root.
    :return: Returns the local root by Secant method ,
    if it fails to find a solutions in the given tries limit it will return None .
    """
    if iterCounter > 100:
        return

    if abs(secondGuess - firstGuess) < epsilon:#Stop condition
        print("after ", iterCounter, "iterations The root found is: ", bcolors.OKBLUE, round(secondGuess, 6), bcolors.ENDC)
        return round(secondGuess, 6)# Returns the root found

    next_guess = (firstGuess * func(secondGuess) - secondGuess * func(firstGuess)) / (func(secondGuess) - func(firstGuess))
    print(bcolors.OKGREEN, "iteration no.", iterCounter, bcolors.ENDC, "\tXi = ", firstGuess, "\tXi+1 = ", secondGuess,
          "\tf(Xi) = ", func(firstGuess))
    # Recursive call with the following guess
    return SecantMethod(func, secondGuess, next_guess, epsilon, iterCounter + 1)
```

מימוש שיטת ניוטון רפסון

פונקציית עזר לחולקה למקטעים ובחירה ניחוש, הפונקציה שולחת את הניחוש הראשוני לפונקציה שמצוות שורש של משווהה לפי שיטת ניוטון רפסון, הפונקציה בודקת אם השורש לא נמצא כבר בניחוש אחר ואם השורש הוא בטווח המבוקש

```
def NewtonsMethodInRangeIterations(func, check_range, tries=10, epsilon=0.0000001, symbol=sp.symbols('x')):
    """
    This function find a root to a function by using the newton raphson method by a given list of guesses.
    :param func: The function with sympy symbols.
    :param check_range: List of guesses.
    :param tries: Number of tries to find the root.
    :param symbol: The symbol you entered in the function (Default is lower case x)
    :param epsilon: The tolerance of the deviation of the solution ;
    How precise you want the solution (the smaller the better).
    :return: Returns a list roots by raphson method ,
    if it fails to find a solutions in the given tries limit it will return an empty list .
    """
    roots = []
    for i in check_range:
        check_number = round(i, 2)
        print(bcolors.HEADER, "First guess:", check_number, bcolors.ENDC)
        # Send to the Secant Method with one guess
        local_root = NewtonsMethod(func, check_number, tries, epsilon, symbol)
        if round(local_root, 6) in roots:
            print(bcolors.FAIL, "Already found that root.", bcolors.ENDC)
        elif not (check_range[0] <= local_root <= check_range[-1]):
            print(bcolors.FAIL, "root out of range.", bcolors.ENDC)
        elif local_root is not None:
            roots += [round(local_root, 6)]
    return roots
```

פונקציה איטרטיבית למציאת שורש של משואה לפי שיטת ניוטון רפסון ע"פ ניחוש ראשוני הפונקציה מוגבלת בטווich ניסיונות למציאת השורש, האפסילון הנבחר הוא **0.0000001** השיטה משתמשת בנגזרת של הפונקציה ע"מ למצוא את שורש המשיך שבכל איטרציה מתקרב לשורש של הפונקציה אותו אנחנו מחפשים

```
def NewtonsMethod(func, x0, tries=100, epsylon=0.0000001, symbole=sp.symbols('x')):
    """
    This function find a root to a function by using the newton raphson method by a given first guess.
    :param func: The function with sympy symbols.
    :param x0: The first guess.
    :param tries: Number of tries to find the root.
    :param symbole: The symbol you entered in the function (Default is lower case x)
    :param epsylon: The tolerance of the deviation of the solution ;
    How precise you want the solution (the smaller the better).
    :return: Returns the local root by raphson method ,
    if it fails to find a solutions in the given tries limit it will return None .
    """
    if func.subs(symbole, x0) == 0:
        return 0
    for i in range(tries):
        print(bcolors.OKBLUE, "Attempt #", i + 1, ":", bcolors.ENDC)
        print("f({0}) = {1} = {2}".format(x0, func, round(func.subs(symbole, x0), 2)))
        print("f'({0}) = {1} = {2}".format(x0, sp.diff(func, symbole),
                                         round(sp.diff(func, symbole).subs(symbole, x0), 2)))
        if sp.diff(func, symbole).subs(symbole, x0) == 0.0:
            continue
        next_x = (x0 - func.subs(symbole, x0) / sp.diff(func, symbole).subs(symbole, x0))
        print("next_X = ", round(next_x, 2))
        t = abs(next_x - x0)
        if t < epsylon:
            print(bcolors.OKGREEN, "Found a Root Solution ; X =", round(next_x, 8), bcolors.ENDC)
            return next_x
        x0 = next_x
    print(bcolors.FAIL, "Haven't Found a Root Solution ; (returning None)", bcolors.ENDC)
    return None
```

פונקציית עזר לייצרת טווח בין מספרים ממשיים

מקבלת נקודת התחלה ונקודות סוף

וקפיצות רצויות

מחזירה רשימה שמכילה את הטווח הרצוי לפי הקפיצות
ששלחנו

טווח רצוי בשאלה:

1.5-(-1,1)

frange(-1,1.5,0.1)

```
def frange(start, end=None, inc=None):
    "Function for a range with incomplete numbers"
    if end == None:
        end = start + 0.0
        start = 0.0

    if inc == None:
        inc = 1.0

    L = []
    while 1:
        next_ = start + len(L) * inc
        if inc > 0 and next_ >= end:
            break
        elif inc < 0 and next_ <= end:
            break
        L.append(next_)

    return L
```

.2. שיטות לחישוב אינטגרל של הפונקציה בטווח $[0,1]$ **שיטת סימפסון, שיטת רומברג**

IMPLEMENTATION: Simpson's Rule (Simpson's Rule Implementation)

ביצוע קירוב לפונקציה ע"י אינטראפובלציית ריבועית שימוש באינטראפובלציית לגראמ' למציאת פולינום דרך כל 3 נקודות מקרבים את הפונקציה באמצעות פולינום מסדר 2 ומחשבים את האינטגרל, באמצעות שיטת לגראנט'. בשיטה זו החלוקה למקטעים צריכה להיות שווה שכן ח�יב להיות זוגי נחשב את ה Δ לפי הנקודה ההתחלתית והסופית כדי לחלק למקטעים שווים ולאחר מכן בולולה ניצור פולינומיים כך שבמקומות האיזוגים נכפיאם ב-4 ובמקומות האיזוגים נכפיאם ב-2 חיבור של כל הפולינומיים יתן לנו פולינום שמקרב את חישוב האינטגרל

```
def SimpsonRule(func, n, a, b):
    """
    Simpson's Rule is a numerical method that approximates the value of a definite integral by using quadratic functions. Simpson's Rule is based on the fact that given three points, we can find the equation of a quadratic through those points (by Lagrange's interpolation).
    :param func: The desired integral function
    :param n: The division number(must be even)
    :param a: Lower bound
    :param b: Upper bound
    :return: The result of the integral calculation
    """

    if n % 2 != 0:
        return 0, False
    h = (b - a) / n
    print("h = ", h)
    str_even = ""
    str_odd = ""
    k2 = b
    #print(bcolors.FAIL, "Error evaluation En = ", round(SimpsonError(my_f, b, a, h), 6), bcolors.ENDC)
    integral = func(a) + func(b)
    # Calculation of a polynomial lagrangian for the sections
    for i in range(n):
        k = a + i * h # new a
        if i != n-1: # new b
            k2 = a+(i+1)*h
        if i % 2 == 0: #even places
            integral += 2 * func(k)
            str_even = "2 * "+str(func(k))
        else: #odd places
            integral += 4 * func(k)
            str_odd = "4 * "+str(func(k))
        print("h/3 ( ", str(func(k)), " + ", str_odd, " + ", str_even, " + ", str(func(k2)), " )")
    integral *= (h/3)
    return integral, True
```

IMPLEMENTATION: Romberg Integration

שיטת נוספת לחישוב האינטגרל וקביעת מספר המקטעים מבססת על פיתוח טור עבור שגיאות הקיטוע. השיטה מtabases על שיטת הטרפז לחישוב אינטגרל (יוסבר להלן)

השיטה משתמשת בחישובים של שיטת הטרפז ושומרת את הערכים שחושבו במטריצה הפונקציה ללקחת רצף פתרונות מקורבים לחישוב האינטגרל ומשפרת את הקירוב

הקירוב הסופי ישמר באיבר האחרון במטריצה $[n][n]$

קובענו את n לפי הקירוב הרצוי לוצאה המקורית, הגדלה של החלוקה למקטעים משפרת את הדיק בוצאה אף משפיעה מאוד על זמן החישוב.

IMPLEMENTATION OF THE TRAPEZOIDAL METHOD (Method 2)

The trapezoidal method is a way to calculate the definite integral of a function by dividing the area under the curve into several trapezoids and calculating the area of each trapezoid.

The trapezoidal rule approximates the area under a curve by dividing the area into several trapezoids and calculating the area of each trapezoid. This method is similar to the rectangle method, but it uses trapezoids instead of rectangles.

```

def TrapezoidalRule(my_f, n, a, b, tf):
    """
    Trapezoidal Rule is a rule that evaluates the area under the curves by dividing the total area
    into smaller trapezoids rather than using rectangles
    :param my_f: The desired integral function
    :param n: The division number
    :param a: Lower bound
    :param b: Upper bound
    :param tf: Variable to decide whether to perform Error evaluation
    :return: The result of the integral calculation
    """

    h = (b - a) / n
    if tf:
        #print(bcolors.FAIL, "Error evaluation En = ", round(TrapezError(my_f, b, a, h), 6), bcolors.ENDC)
    integral = 0.5 * (my_f(a) + my_f(b))
    for i in range(n):
        integral += my_f(a + h * i)
    integral *= h
    return integral

```

Implementation of the trapezoidal method using Python.

```

def RombergsMethod(f, n, a, b):
    """
    Romberg integration is an extrapolation technique which allows us to take a sequence
    approximate solutions to an integral and calculate a better approximation.
    This technique assumes that the function we are integrating is sufficiently differentiable
    :param f: The desired integral function
    :param n: The division number
    :param a: Lower bound
    :param b: Upper bound
    :return: The result of the integral calculation
    """

    matrix = [[0 for i in range(n)] for j in range(n)]
    for k in range(0, n):
        # Using the trapezoidal method
        matrix[k][0] = TrapezoidalRule(f, 2**k, a, b, False)
        # Romberg recursive formula Using values that have already been calculated
        for j in range(0, k):
            matrix[k][j + 1] = (4 ** (j + 1) * matrix[k][j] - matrix[k - 1][j]) / (4 ** (j + 1) - 1)
            print("R[{0}][{1}] = ".format(k, j+1), round(matrix[k][j+1], 6))
    return matrix

```

Implementation of the Romberg method using Python.

Numeric Analysis - Final Project

Finding Equation Roots + Numerical Integration

Question 12

• שאלה מס' 12

עבור הפונקציה הבאה

$$f(x) = (2xe^{-x} + \ln(2x^2))(2x^4 + 2x^2 - 3x - 5)$$

3. מצאו באמצעות שתי שיטות את כל השורשים האמיטיים הנמצאים בקטע $[0., 1.5]$
4. השתמשו בקירוב סימפסון ובשיטת רומברג למציאת הערך של האינטגרל בקטע $[0.5, 1]$

GitHub: <https://github.com/Maor-Ar/Numerical-Analysis-Hackathon/blob/main/Q12%20Finding%20Equation%20Roots%20Numerical%20Integration.py>

- . שיטות למציאת שורשי משווהה בקטע הנ"ל שיטת החציה, שיטת המיתר
IMPLEMENTATION
פונקציה עוזר להלכה לקטעים ובחירה 2 ביחסים, הפונקציה שולחת 2 ניחושים לפונקציה שמצוות שורש של משווהה לפ'. שיטת המיתר, הפונקציה בודקת אם השורש לא נמצא כבר באיטרציה קודמת וכן אם השורש הוא בטוחה המבוקש.

```
def SecantMethodInRangeIterations(f, check_range, epsilon=0.0001):  
    """  
    This function find a root to a function by using the secant method by a given list of values to check between.  
    :param f: The function (as a python function).  
    :param check_range: List of values to check between ; e.g (1,2,3,4,5) it will check between 1-2,2-3,... .  
    :param epsilon: The tolerance of the deviation of the solution ;  
    How precise you want the solution (the smaller the better).  
    :return: Returns a list roots by secant method ,  
    if it fails to find a solutions in the given tries limit it will return an empty list .  
    """  
    domain = 0  
    roots = []  
    iterCounter = 0  
    for i in check_range:  
        startPoint = round(i, 2)  
        endPoint = round(i + 0.1, 2)  
        if startPoint != domain:  
            print(bcolors.HEADER, "Checked range:", startPoint, "-", endPoint, bcolors.ENDC)  
            local_root = SecantMethod(f, startPoint, endPoint, epsilon, iterCounter)  
            if local_root in roots:  
                print(bcolors.FAIL, "Already found that root.", bcolors.ENDC)  
            elif not (startPoint <= local_root <= endPoint):  
                print(bcolors.FAIL, "root out of range.", bcolors.ENDC)  
            elif local_root is not None:  
                roots += [round(local_root, 6)]  
  
    return roots
```

פונקציה ורקורסיבית למציאת שורש של משווה לפי שיפור הניחוש בכל קרייה ורקורסיבית
 תנאי עצירה נבחר לפי אפסילון שקבענו או אם כמות הקריאה הרקורסיביות גדלה מ-100 (השיטה לא מתכנסת) האפסילון
 הנבחר הוא **0.0000001**

```
def SecantMethod(func, firstGuess, secondGuess, epsilon, iterCounter):
    """
        This function find a root to a function by using the SecantMethod method by a given tow guess.
        :param func: The function on which the method is run
        :param firstGuess: The first guess
        :param secondGuess: The second guess
        :param epsilon: The tolerance of the deviation of the solution
        :param iterCounter: number of tries until the function found the root.
        :return: Returns the local root by Secant method ,
        if it fails to find a solutions in the given tries limit it will return None .
    """

    if iterCounter > 100:
        return

    if abs(secondGuess - firstGuess) < epsilon:#Stop condition
        print("after ", iterCounter, "iterations The root found is: ", bcolors.OKBLUE, round(secondGuess, 6), bcolors.ENDC)
        return round(secondGuess, 6)# Returns the root found

    next_guess = (firstGuess * func(secondGuess) - secondGuess * func(firstGuess)) / (func(secondGuess) - func(firstGuess))
    print(bcolors.OKGREEN, "iteration no.", iterCounter, bcolors.ENDC, "\tXi = ", firstGuess, "\tXi+1 = ", secondGuess,
          "\tf(Xi) = ", func(firstGuess))

    return SecantMethod(func, secondGuess, next_guess, epsilon, iterCounter + 1)
```

מימוש שיטת החציה

בשיטת החציה אנחנו בודקים אם יש שינוי סימן בין מקטעים בקטע, שם יש חישוב השוואת שורש של המשווה
 נשלח את הקטע לשיטת החציה ושם נחלק את הקטע ב-2 ונבדק אם להמשיך לבדוק ימין או משמאל לנק' האמצעית
 פונקציית עזר לחיקות טווח המבוקש למקטעים של 0.1

```
def BisectionMethodSections(func, Cheackrange, epsilon):
    """
        This function dividing the range into small sections and send the to the bisection method
        :param func: Any function
        :param Cheackrange: Range for checking
        :param epsilon: Stop condition

    """
    iterCounter = 0
    result = []
    domainPoints = 0
    for i in Cheackrange:
        seperate = round(i, 2)
        next_seperate = round(i+ 0.1, 2)
        if checkifcontinus(my_f, seperate, sp.symbols('x')) == False :
            print(bcolors.FAIL,"this point ", seperate, " not in domain of this function", bcolors.ENDC)
            continue
        if func(next_seperate) == 0:
            print(bcolors.OKBLUE, "root in ", next_seperate, bcolors.ENDC)
            result.append(next_seperate)
        if (func(seperate) * func(next_seperate)) < 0:
            print("sign changing found between ",separate,'-',next_seperate)
            result += BisectionMethod(func, seperate, next_seperate, epsilon, iterCounter)

    return result
```

השיטה בודקת מקרה מיוחד של פונקציה יש שורש אמיתי בנקודה 0
 בדיקת מקטעים של 0.1, כשהנמצא שינוי סימן נשלח לשיטת החציה שתמצא לנו את השורש בטווח זהה
 השורשים ישמרו בראשימה אחת אותה נחדר

```

def BisectionMethod(polynomial, startPoint, endPoint, epsilon, iterCounter):
    """
    the bisection method is a root-finding method that applies to any
    continuous functions for which one knows two values with opposite signs
    :param polynomial: The function on which the method is run
    :param startPoint: Starting point of the range
    :param endPoint: End point of the range
    :param epsilon: The tolerance of the deviation of the solution
    :param iterCounter: Counter of the number of iterations performed
    :return: Roots of the equation found
    """

    roots = []
    middle = (startPoint + endPoint) / 2

    if iterCounter > EvaluateError(startPoint, endPoint):
        print(bcolors.FAIL, "The Method isn't convergent.", bcolors.ENDC)
        return roots

    if (abs(endPoint - startPoint)) < epsilon:
        print("After ", iterCounter-1, "iterations The root found is: ", bcolors.OKBLUE, round(middle, 6),
              bcolors.ENDC)
        roots.append(round(middle, 6))
        return roots

    if polynomial(startPoint) * polynomial(middle) > 0:
        print(bcolors.OKGREEN, "iteration no.", iterCounter, bcolors.ENDC, "\ta = ", middle, " \tb = ", endPoint, "\tf(a) = ", polynomial(middle),
              "\tf(b) = ", polynomial(endPoint))
        roots += BisectionMethod(polynomial, middle, endPoint, epsilon, iterCounter + 1)
        return roots
    else:
        print(bcolors.OKGREEN, "iteration no.", iterCounter, bcolors.ENDC, "\ta = ", startPoint, " \tb = ", middle, "\tf(a) = ", polynomial(startPoint),
              "\tf(b) = ", polynomial(middle))
        roots += BisectionMethod(polynomial, startPoint, middle, epsilon, iterCounter + 1)
        return roots

```

שיטת החציה פונקציה וקורסיבית שמחשבת את אמצע הקטע וכל פעם מבצעת קריאות רקורסיביות לקטע החצוי משמאלו או לקטע החצוי מימין

התנאי עכירה הוא שהשוני בין 2 הנקודות שאנו חקנו בודקים קטן מפאסילון שבחרנו
בנוסף לכך, נספר את במות האיטרציות עד שהגענו לשורש הרצוי
לשיטת החציה יש חישוב שגיאה שנותן לנו אינדיקציה אם השיטה מתכנסת לשורש או לא

פונקציה להערכת השגיאה של שיטת החציה

חישוב השגיאה עפ"י נוסחה והחזרת הערך

```

def EvaluateError(startPoint, endPoint):
    """
    This function Helps us find out if we can find a root in a limited amount of tries in a specific range.
    :param startPoint: start of range.
    :param endPoint: end of range.
    :return:
    """

    exp = pow(10, -10)
    if endPoint - startPoint == 0:
        return 100
    return ((-1) * math.log(exp / (endPoint - startPoint), e)) / math.log(2, e)

```

פונקציה לבדיקת נקודה בתחום הגדולה

בפונקציה הנתונה בתרגיל יש תחום הגדולה של ערכים שלא מוגדרים בפונקציה, אחד מהמערכים הוא 0 שהוא חלק מהטווח שהتابקשנו לבדוק

לכן השתמשנו בפונקציית עזר שמקבלת את הפונקציה וערך ומחזירה אמת או שקר אם הוא בתחום הגדולה של הפונקציה

```
def checkifcontinus(func,x,symbol):
    """
    Function for checking if specific x is in domain of specific function
    :param func: Any function
    :param x: specific x
    :param symbol: x as factor
    :return: True / False
    """
    return (sp.limit(func, symbol, x).is_real)
```

בבדיקה הטווח הוספנו תנאי של בדיקה האם הערך הוא בתחום הגדולה של הפונקציה

```
if checkifcontinus(my_f,seperate,sp.symbols('x')) == False:
    print(bccolors.FAIL,"this point ", seperate, " not in domain of this function", bccolors.ENDC)
    continue
```

שיטות אינטגרציה

1. שיטות לחישוב אינטגרל של הפונקציה בטווח $[0,1]$ **שיטת סימפסון, שיטת רומברג**
מימוש שיטת סימפסון (שיטת מסדר 4)
ביצוע קירוב לפונקציה ע"י אינטראפולציה ריבועית שימוש באינטראפולציה לגראנט למציאת פולינום
דרך כל 3 נקודות מקרובים את הפונקציה באמצעות פולינום מסדר 2 ומחשבים את האינטגרל, באמצעות שיטות שיטות לשיטות גראנט.
בשיטת זו החלוקה למקטעים צריכה להיות שווה لكن ח' חייב להיות זוגי
נחשב את ה' לפי הנקודה ההתחלתית והסתopiaת כדי לחלק למקטעים שווים ולאחר מכן בולאה ניצור פולינומים כך שבמקומות
האי זוגיים נכפיל ב4 ובמקומות הזוגיים נכפיל ב2
חיבור של כל הפולינומים יתן לנו פולינום שמקרב את חישוב האינטגרל

```
def SimpsonRule(func, n, a, b):
    """
    Simpson's Rule is a numerical method that approximates the value of a definite integral by using quadratic
    functions. Simpson's Rule is based on the fact that given three points,
    we can find the equation of a quadratic through those points (by Lagrange's interpolation)
    :param func: The desired integral function
    :param n: The division number(must be even)
    :param a: Lower bound
    :param b: Upper bound
    :return: The result of the integral calculation
    """
    if n % 2 != 0:
        return 0, False
    h = (b - a) / n
    print("h = ", h)
    str_even = ""
    str_odd = ""
    k2=b
    #print(bcolors.FAIL, "Error evaluation En = ", round(SimpsonError(my_f, b, a, h), 6), bcolors.ENDC)
    integral = func(a) + func(b)
    for i in range(n):
        k = a + i * h
        if i != n-1:
            k2 = a+(i+1)*h
        if i % 2 == 0:
            integral += 2 * func(k)
            str_even = "2 * "+str(func(k))
        else:
            integral += 4 * func(k)
            str_odd = "4 * "+str(func(k))
        print("h/3 * ( ", str(func(k)), " + ", str_odd, " + ", str_even, " + ", str(func(k2)), " )")
    integral *= (h/3)
    return integral, True
```

מימוש שיטת רומברג

שיטת נוספת לחישוב האינטגרל וקביעת מספר המקטעים מבוססת על פיתוח טור עבור שגיאות הקיטוע. השיטה
מתבססת על שיטת הטרפז לחישוב אינטגרל (יוסבר להלן)

השיטה משתמשת בחישובים של שיטת הטרפז ושומרת את הערכים שחושבו במטריצה הפונקציה לנקודות רצף
פתרונות מקורבים לחישוב האינטגרל ומשפרת את הקירוב

הקירוב הסופי ישמר באיבר האחרון במטריצה $[h][h]$ matrix

קובענו את ה' לפי הקירוב הרצוי לתוכה המקורית, הגדלה של החלוקה למקטעים משפרת את הדיק בתוכה אך
משפיעה מאוד על זמן החישוב.

מימוש שיטת הטרפז (שיטת מסדר 2)

שיטת הטרפז היא שיטת חישוב הנותנת קירוב לערכו של שטח כלוא או לערכו של אינטגרל מסוים על פונקציה בין שני ערכים לא ביצוע פועלם האינטגרל כלל.

השיטה מחלקת את השטח למקטעים לפי ח נתון כך שבכל מקטע נחשב את שטח הטרפז ע"י חישוב של h בשיטה זו ישנה שגיאה והקירוב לחוב יותר רחוק מהתוצאה המקורית.

```
def TrapezoidalRule(f, n, a, b, tf):
    """
    Trapezoidal Rule is a rule that evaluates the area under the curves by dividing the total area
    into smaller trapezoids rather than using rectangles
    :param f: The desired integral function
    :param n: The division number
    :param a: Lower bound
    :param b: Upper bound
    :param tf: Variable to decide whether to perform Error evaluation
    :return: The result of the integral calculation
    """
    h = (b - a) / n
    # if tf:
    #     print(bcolors.FAIL, "Error evaluation En = ", round(TrapezError(func(), b, a, h), 6), bcolors.ENDC)
    integral = 0.5 * (f(a)*f(b))
    for i in range(n):
        integral += f(a+h*i)
    integral *= h
    return integral
```

שיטת רומברג עם שימוש בשיטת הטרפז

```
def RombergsMethod(f, n, a, b):
    """
    Romberg integration is an extrapolation technique which allows us to take a sequence
    approximate solutions to an integral and calculate a better approximation.
    This technique assumes that the function we are integrating is sufficiently differentiable
    :param f: The desired integral function
    :param n: The division number
    :param a: Lower bound
    :param b: Upper bound
    :return: The result of the integral calculation
    """
    matrix = [[0 for i in range(n)] for j in range(n)]
    for k in range(0, n):
        # Using the trapezoidal method
        matrix[k][0] = TrapezoidalRule(f, 2**k, a, b, False)
        # Romberg recursive formula Using values that have already been calculated
        for j in range(0, k):
            matrix[k][j + 1] = (4 ** (j + 1) * matrix[k][j] - matrix[k - 1][j]) / (4 ** (j + 1) - 1)
            print("R[{0}][{1}] = ".format(k, j+1), round(matrix[k][j+1], 6))
    return matrix
```

הערה: מימשו גם פונקציות לחישוב שגיאה (בשיטות) אך לא נתקשנו להתייחס במליה זו.

Numeric Analysis - Final Project

Finding Equation Roots + Numerical Integration

Question 14

• שאלה מס' 14

עבור הפונקציה הבאה

$$f(x) = (xe^{-x^2+5x})(2x^2 - 3x - 5)$$

1. מצאו באמצעות שתי שיטות את כל השורשים האמיטיים הנמצאים בקטע [0,3]
2. השתמשו בקירוב סימפסון ובשיטת רומברג למציאת הערך של האינטגרל בקטע [0.5,1]

GitHub: <https://github.com/Maor-Ar/Numerical-Analysis-Hackathon/blob/main/Q14%20Finding%20Equation%20Roots%20Numerical%20Integration.py>

2. שיטות למציאת שורשי משווהה בקטע הנקרא שיטת החציה, שיטת המיתר
פונקציית עזר להלכה למקטעים ובחירה 2 ניחושים, הפונקציה שולחת 2 ניחושים לפונקציה שמצוות שורש של משווהה לפוי
שיטת המיתר, הפונקציה בודקת אם השורש לא נמצא כבר באיטרציה הקודמת וכן אם השורש הוא בטוחה המבוקש.

```
def SecantMethodInRangeIterations(f, check_range, epsilon=0.0001):
    """
    This function find a root to a function by using the secant method by a given list of values to check between.
    :param f: The function (as a python function).
    :param check_range: List of values to check between ; e.g (1,2,3,4,5) it will check between 1-2,2-3,....
    :param epsilon: The tolerance of the deviation of the solution ;
    How precise you want the solution (the smaller the better).
    :return: Returns a list roots by secant method ,
    if it fails to find a solutions in the given tries limit it will return an empty list .
    """
    roots = []
    iterCounter = 0
    for i in check_range:
        startPoint = round(i, 2)
        endPoint = round(i + 0.1, 2)
        print(bcolors.HEADER, "Checked range:", startPoint, "-", endPoint, bcolors.ENDC)
        # Send to the Secant Method with 2 guesses
        local_root = SecantMethod(f, startPoint, endPoint, epsilon, iterCounter)
        # If the root has been found in previous iterations
        if round(local_root, 6) in roots:
            print(bcolors.FAIL, "Already found that root.", bcolors.ENDC)
        # If the root is out of range tested
        elif not (startPoint <= local_root <= endPoint):
            print(bcolors.FAIL, "root out of range.", bcolors.ENDC)
        elif local_root is not None:
            roots += [round(local_root, 6)]
    return roots
```

פונקציה וקורסיבית למציאת שורש של משווה לפי שיפור הניחוש בכל קריאה וקורסיבית

תנאי עצירה נבחר לפי אפסילון שקבענו או אם כמות הקריאה הקורסיבית גדולה מ-100 (השיטה לא מתכנסת) האפסילון הבוחר הוא **0.0001**

```
def SecantMethod(func, firstGuess, secondGuess, epsilon, iterCounter):
    """
        This function find a root to a function by using the SecantMethod method by a given tow guess.
        :param func: The function on which the method is run
        :param firstGuess: The first guess
        :param secondGuess: The second guess
        :param epsilon: The tolerance of the deviation of the solution
        :param iterCounter: number of tries until the function found the root.
        :return: Returns the local root by Secant method ,
                if it fails to find a solutions in the given tries limit it will return None .
    """
    if iterCounter > 100:
        return

    if abs(secondGuess - firstGuess) < epsilon: #Stop condition
        print("after ", iterCounter, "iterations The root found is: ", bcolors.OKBLUE, round(secondGuess, 6), bcolors.ENDC)
        return round(secondGuess, 6) # Returns the root found

    next_guess = (firstGuess * func(secondGuess) - secondGuess * func(firstGuess)) / (func(secondGuess) - func(firstGuess))
    print(bcolors.OKGREEN, "iteration no.", iterCounter, bcolors.ENDC, "\tXi = ", firstGuess, "\tXi+1 = ", secondGuess,
          "\tf(Xi) = ", func(firstGuess))
    # Recursive call with the following guess
    return SecantMethod(func, secondGuess, next_guess, epsilon, iterCounter + 1)
```

מימוש שיטת החציה

בשיטת בחציה אנחנו בודקים אם יש שינוי סימן בין מקטעים בקטע, שם יש חישוב למצאות שורש של המשווה
נשלח את הקטע לשיטת החציה ושם נחלק את הקטע ב-2 ונבדק אם להמשיך לבדוק ימין או משמאל לנק' האמצעית

```
def BisectionMethodSections(func, Cheackrange, epsilon):
    """
        This function dividing the range into small sections and send the to the bisection method
        :param func: Any function
        :param Cheackrange: Range for checking
        :param epsilon: Stop condition
    """

    iterCounter = 0
    result = []
    for i in Cheackrange:
        seperate = round(i, 2)
        next_seperate = round(i+ 0.1, 2)
        if checkifcontinus(my_f, seperate, sp.symbols('x')) == False:
            print(bcolors.FAIL,"this point ", seperate, " not in domain of this function", bcolors.ENDC)
            continue
        if func(seperate) == 0:
            print(bcolors.OKBLUE, "root in ", seperate, bcolors.ENDC)
            result.append(seperate)
        if (func(seperate) * func(next_seperate)) < 0:
            print("sign changing found between ",seperate,'-',next_seperate)
            result += BisectionMethod(func, seperate, next_seperate, epsilon, iterCounter)
    return result
```

פונקציית עזר לחילוקת טווח למקטעים של 0.1
בדיקת מקטעים של 0.1 , בשנמצא שני סימן נשלח לשיטת החツיה שתמצא לנו את השורש בטווח זהה

```
def BisectionMethod(polynomial, startPoint, endPoint, epsilon, iterCounter):
    """
    the bisection method is a root-finding method that applies to any
    continuous functions for which one knows two values with opposite signs
    :param polynomial: The function on which the method is run
    :param startPoint: Starting point of the range
    :param endPoint: End point of the range
    :param epsilon: The tolerance of the deviation of the solution
    :param iterCounter: Counter of the number of iterations performed
    :return: Roots of the equation found
    """

    roots = []
    middle = (startPoint + endPoint) / 2

    if iterCounter > EvaluateError(startPoint, endPoint):
        print(bcolors.FAIL, "The Method isn't convergent.", bcolors.ENDC)
        return roots

    if (abs(endPoint - startPoint)) < epsilon:
        print("after ", iterCounter-1, "iterations The root found is: ", bcolors.OKBLUE, round(middle, 6),
              bcolors.ENDC)
        roots.append(round(middle, 6))
        return roots

    if polynomial(startPoint) * polynomial(middle) > 0:
        print(bcolors.OKGREEN, "iteration no.", iterCounter, bcolors.ENDC, "\ta = ", middle, " \tb = ", endPoint, "\tf(a) = ", polynomial(middle),
              "\tf(b) = ", polynomial(endPoint))
        roots += BisectionMethod(polynomial, middle, endPoint, epsilon, iterCounter + 1)
        return roots
    else:
        print(bcolors.OKGREEN, "iteration no.", iterCounter, bcolors.ENDC, "\ta = ", startPoint, " \tb = ", middle, "\tf(a) = ", polynomial(startPoint),
              "\tf(b) = ", polynomial(middle))
        roots += BisectionMethod(polynomial, startPoint, middle, epsilon, iterCounter + 1)
        return roots
```

שיטת החツיה פונקציה וקורסיבית שמחשבת את אמצע הקטע ובכל פעם מבצעת קריאות ורקורסיביות לקטוע החצוי משמאלי או לקטע החצוי מימין

התנאי עזירה הוא שהשימי בין 2 הנקודות שאנו בודקים קטן מפאסילון שבחרנו

בנוסף לכך, נספר את כמות האיטרציות עד שהגענו לשורש הרצוי

לשיטת החツיה יש חישוב שגיאה שנותן לנו אינדיקציה אם השיטה מתכנסת לשורש או לא

פונקציה להערכת השגיאה של שיטת החツיה

חישוב השגיאה עפ"י נוסחה והחזרת הערך

```
def EvaluateError(startPoint, endPoint):
    """
    This function Helps us find out if we can find a root in a limited amount of tries in a specific range.
    :param startPoint: start of range.
    :param endPoint: end of range.
    :return:
    """

    exp = pow(10, -10)
    if endPoint - startPoint == 0:
        return 100
    return ((-1) * math.log(exp / (endPoint - startPoint), e)) / math.log(2, e)
```

פונקציה לבדיקת נקודה בתחום הגדירה

בפונקציה הנתונה יש תחום הגדירה של ערכים שלא מוגדרים בפונקציה, אחד מהערכים הוא 0 שהוא חלק מהטווח שהתבוננו לבדוק
לכן השתמשנו בפונקציית עזר שמקבלת את הפונקציה וערך ומחזירה אמת או שקר אם הוא בתחום הגדירה של הפונקציה

```
def checkifcontinus(func,x,symbol):  
    """  
    Function for checking if specific x is in domain of specific function  
    :param func: Any function  
    :param x: specific x  
    :param symbol: x as factor  
    :return: True / False  
    """  
  
    return (sp.limit(func, symbol, x).is_real)
```

בדיקת הטווח הוספנו תנאי של בדיקה האם הערך הוא בתחום הגדירה של הפונקציה

```
if checkifcontinus(my_f,seperate,sp.symbols('x')) == False :  
    print(bcolors.FAIL,"this point ", seperate, " not in domain of this function", bcolors.ENDC)  
    continue
```

שיטות אינטגרציה

2. שיטות לחישוב אינטגרל של הפונקציה בטווח [0,1] **שיטה סימפסון, שיטת רומברג**
מימוש שיטת סימפסון (שיטת מסדר 4)
ביצוע קירוב לפונקציה ע"י אינטראפולציה ריבועית שימוש באינטראפולציית לגראמ' למציאת פולינום
דרך כל 3 נקודות מקרבים את הפונקציה באמצעות פולינום מסדר 2 ומחשבים את האינטגרל, באמצעות שיטת לגראמ'.
בשיטת זו החלוקה למקטעים צריכה להיות שווה لكن ח' חייב להיות זוגי
נחשב את ה נקודה ההתחלתית והסופית כדי לחלק למקטעים שווים ולאחר מכן בולאה ניצור פולינומים בר שבמקומות
האי זוגיים נכפיל ב 4 ובמקומות הזוגיים נכפיל ב 2
חיבור של כל הפולינומים יתן לנו פולינום שמקרב את חישוב האינטגרל

```
def SimpsonRule(func, n, a, b):  
    """  
    Simpson's Rule is a numerical method that approximates the value of a definite integral by using quadratic  
    functions Simpson's Rule is based on the fact that given three points,  
    we can find the equation of a quadratic through those points (by Lagrange's interpolation)  
    :param func: The desired integral function  
    :param n: The division number(must be even)  
    :param a: Lower bound  
    :param b: Upper bound  
    :return: The result of the integral calculation  
    """  
  
    if n % 2 != 0:  
        return 0, False  
    h = (b - a) / n  
    print("h = ", h)  
    str_even = ""  
    str_odd = ""  
    k2 = b  
    #print(bcolors.FAIL, "Error evaluation En = ", round(SimpsonError(my_f, b, a, h), 6), bcolors.ENDC)  
    integral = func(a) + func(b)  
    # Calculation of a polynomial lagranz for the sections  
    for i in range(n):  
        k = a + i * h # new a  
        if i != n-1: # new b  
            k2 = a+(i+1)*h  
        if i % 2 == 0: #even places  
            integral += 2 * func(k)  
            str_even = "2 * "+str(func(k))  
        else: #odd places  
            integral += 4 * func(k)  
            str_odd = "4 * "+str(func(k))  
        print("h/3 ( ", str(func(k)), " + ", str_odd, " + ", str_even, " + ", str(func(k2)), ")" )"  
    integral *= (h/3)  
    return integral, True
```

מימוש שיטת רומברג

שיטה נוספת לחישוב האינטגרל וקביעת מספר המקטעים מבוססת על פיתוח טור עבור שגיאות הקיטוע. השיטה
מתבססת על שיטת הטרפז לחישוב אינטגרל (יוסבר להלן)

השיטה משתמשת בחישובים של שיטת הטרפז ושומרת את הערכים שחושבו במטריצה הפונקציה לבקשת רצף
פתרונות מקרבים לחישוב האינטגרל ומשפרת את הקירוב

הקירוב הסופי ישמר באיבר האחרון במטריצה $[n][n]$

קובענו את ח' לפי הקירוב הרצוי לתוכה המקורית, הגדלה של החלוקה למקטעים משפרת את הדיק בתוכה אך
משפיעה מאוד על זמן החישוב

מימוש שיטת הטרפז (שיטת מסדר 2)

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.

השיטה מחלקת את השטח למקטעים לפי ח נטען כך שבכל מקטע נחשב את שטח הטרפז ע"י חישוב של h בשיטה זו ישנה שגיאה והקירוב לחוב יוצאות מההתוצאה המקורית.

```
def TrapezoidalRule(my_f, n, a, b, tf):
    """
    trapezoidal Rule is a rule that evaluates the area under the curves by dividing the total area
    into smaller trapezoids rather than using rectangles
    :param my_f: The desired integral function
    :param n: The division number
    :param a: Lower bound
    :param b: Upper bound
    :param tf: Variable to decide whether to perform Error evaluation
    :return: The result of the integral calculation
    """

    h = (b - a) / n
    #if tf:
    #    #print(bcolors.FAIL, "Error evaluation En = ", round(TrapezError(my_f, b, a, h), 6), bcolors.ENDC)
    integral = 0.5 * (my_f(a) * my_f(b))
    for i in range(n):
        integral += my_f(a + h * i)
    integral *= h
    return integral
```

שיטת רומברג עם שימוש בשיטת הטרפז

```
def RombergsMethod(f, n, a, b):
    """
    Romberg integration is an extrapolation technique which allows us to take a sequence
    approximate solutions to an integral and calculate a better approximation.
    This technique assumes that the function we are integrating is sufficiently differentiable
    :param f: The desired integral function
    :param n: The division number
    :param a: Lower bound
    :param b: Upper bound
    :return: The result of the integral calculation
    """

    matrix = [[0 for i in range(n)] for j in range(n)]
    for k in range(0, n):
        # Using the trapezoidal method
        matrix[k][0] = TrapezoidalRule(f, 2**k, a, b, False)
        # Romberg recursive formula Using values that have already been calculated
        for j in range(0, k):
            matrix[k][j + 1] = (4 ** (j + 1) * matrix[k][j] - matrix[k - 1][j]) / (4 ** (j + 1) - 1)
            print("R[{0}][{1}] = ".format(k, j+1), round(matrix[k][j+1], 6))
    return matrix
```

הערה: מימשנו גם פונקציות לחישוב שגיאה (בשיטות) אך לא נתקשנו להתייחס במטריה זו.

Numeric Analysis - Final Project

Systems of Linear Equations

Question 28

* שאלה מס' 28

פתרו את המטריצה הבאה בשתי דרכים והשו בין התוצאות

$$\begin{pmatrix} 1 & 2 & -2 \\ 1 & 1 & 1 \\ 2 & 2 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 7 \\ 2 \\ 5 \end{pmatrix}$$

GitHub: <https://github.com/Maor-Ar/Numerical-Analysis-Hackathon/blob/main/Q28%20Systems%20of%20Linear%20Equations.py>

```
"""
* Authors: Maor Arnon (ID: 205974553) and Neriya Zudi (ID: 207073545) and
* Matan Ohayon (ID: 311435614) and Matan Sofer (ID: 208491811)
* Emails: maor10@ac.sce.ac.il | neriya.zudi@gmail.com
* matan15595m@gmail.com | sofermatan123@gmail.com
* Department of Computer Engineering - Assignment 2 - Numeric Analytics
"""

from datetime import datetime

class bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    # Background colors:
    GREYBG = '\033[100m'
    REDBG = '\033[101m'
    GREENBG = '\033[102m'
    YELLOWBG = '\033[103m'
    BLUEBG = '\033[104m'
    PINKBG = '\033[105m'
    CYANBG = '\033[106m'

local_dt = datetime.now()
d=str(local_dt.day)
h=str(local_dt.hour)
m=str(local_dt.minute)
Five_zeros="00000"
Five_zeros=Five_zeros[:5]

def PrintMatrix(matrix):
    """
    Matrix Printing Function
    :param matrix: Matrix nxn
    """
    for line in matrix:
        line.append(')')
        line.insert(0,'(')
        print(' '.join(map(str, line)))
        line.remove(')')
        line.remove('(')

def PrintVectorFinal(vector):
    """
    Matrix Printing Function
    :param matrix: Matrix nxn
    """
    print('Solution:')
    PrintMatrix(vector)
    print()
    for i in range(len(vector)):
        print("Solution to x"+str(i)+", value, in the required format.:")
        print(str(vector[i])[0]+bcolors.FAIL+''+bcolors.OKGREEN+d+bcolors.OKBLUE+h+m+bcolors.ENDC+'\n')

def Determinant(matrix, mul):
    """
    Recursive function for determinant calculation
    :param matrix: Matrix nxn
    :param mul: The double number
    :return: determinant of matrix
    """
    width = len(matrix)
    # Stop Conditions
    if width == 1:
        return mul * matrix[0][0]
    else:
        sign = -1
        det = 0
        for i in range(width):
            m = []
            for j in range(1, width):
                buff = []
                for k in range(width):
                    if k != i:
                        buff.append(matrix[j][k])
                m.append(buff)
            # Change the sign of the multiply number
            sign *= -1
            # Recursive call for determinant calculation
            det = det + mul * Determinant(m, sign * matrix[0][i])
        return det
```

צבעים להדפסות =>

נתוני זמן מהמערכת =>

פונקציה להדפסת מטריצה =>

פונקציה להדפסת פלט לפי הדרישה=>

פונקציה לדטרמיננטה=>

```

def InverseMatrixNoPrints(matrix, vector):
    """
    Function for calculating an inverse matrix, Without printing the steps (for checking cond)
    :param matrix: Matrix nxn
    :return: Inverse matrix
    """
    # Unverti reversible matrix
    if Determinant(matrix, 1) == 0:
        print("Error,Singular Matrix\n")
        return
    # result matrix initialized as singularity matrix
    result = MakeIMatrix(len(matrix), len(matrix))
    # loop for each row
    for i in range(len(matrix[0])):
        # turn the pivot into 1 (make elementary matrix and multiply with the result matrix )
        # pivoting process
        matrix, vector = RowXchange(matrix, vector)
        elementary = MakeIMatrix(len(matrix[0]), len(matrix))
        elementary[i][i] = 1/matrix[i][i]
        result = MultiplyMatrix(elementary, result)
        matrix = MultiplyMatrix(elementary, matrix)
        # make elementary loop to iterate for each row and subtract the number below (specific) pivot to zero (make
        # elementary matrix and multiply with the result matrix )
        for j in range(i+1, len(matrix)):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            elementary[j][i] = -(matrix[j][i])
            matrix = MultiplyMatrix(elementary, matrix)
            result = MultiplyMatrix(elementary, result)
    # after finishing with the lower part of the matrix subtract the numbers above the pivot with elementary for loop
    # (make elementary matrix and multiply with the result matrix )
    for i in range(len(matrix[0])-1, 0, -1):
        for j in range(i-1, -1, -1):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            elementary[j][i] = -(matrix[j][i])
            matrix = MultiplyMatrix(elementary, matrix)
            result = MultiplyMatrix(elementary, result)
    return result

```

פונקציה למציאת הופכי בלי הדרשות =>

```

def RowXchange(matrix, vector):
    """
    Function for replacing rows with both a matrix and a vector
    :param matrix: Matrix nxn
    :param vector: Vector n
    :return: Replace rows after a pivoting process
    """
    for i in range(len(matrix)):
        max = abs(matrix[i][i])
        for j in range(i, len(matrix)):
            # The pivot member is the maximum in each column
            if abs(matrix[j][i]) > max:
                temp = matrix[j]
                temp_b = vector[j]
                matrix[j] = matrix[i]
                vector[j] = vector[i]
                matrix[i] = temp
                vector[i] = temp_b
                max = abs(matrix[i][i])
    return [matrix, vector]

```

פונקציה להחלפת שורות לפי ערך דומיננטי (גם לקטור) =>

```

def GaussJordanElimination(matrix, vector):
    """
    Function for mading a linear equation using gauss's elimination method
    :param matrix: Matrix nxn
    :param vector: Vector n
    :return: Solve Ax=b -> x=A(-1)b
    """
    # Pivoting process
    # matrix, vector = RowXchange(matrix, vector)
    # print("Matrix after pivoting - ")
    # PrintMatrix(matrix)
    # Inverse matrix calculation
    invert = InverseMatrix(matrix, vector)
    print("Matrix after inversion - ")
    PrintMatrix(invert)
    return MulMatrixVector(invert, vector)

```

פונקציה לפתרית מטריצה לפי שיטת גאוס גordan =>

```

def MulMatrixVector(InversedMat, b_vector):
    """
    Function for multiplying a vector matrix
    :param InversedMat: Matrix nxn
    :param b_vector: Vector n
    :return: Result vector
    """
    result = []
    # Initialize the x vector
    for i in range(len(b_vector)):
        result.append([])
        result[i].append(0)
    # Multiplication of inverse matrix in the result vector
    for i in range(len(InversedMat)):
        for k in range(len(b_vector)):
            result[i][0] += InversedMat[i][k] * b_vector[k][0]
    return result

```

פונקציה להכפלת מטריצות =>

פונקציה למציאת מטריצת ה-U =<

```
def UMatrix(matrix, vector):
    """
    :param matrix: Matrix nxn
    :return: Disassembly into a U matrix
    """
    # result matrix initialized as singularity matrix
    U = MakeIMatrix(len(matrix), len(matrix))
    # loop for each row
    for i in range(len(matrix[0])):
        # pivoting process
        matrix, vector = RowXchangeZero(matrix, vector)
        for j in range(i + 1, len(matrix)):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            # Finding the M(ij) to reset the organs under the pivot
            elementary[j][i] = -(matrix[j][i]) / matrix[i][i]
            matrix = MultiplyMatrix(elementary, matrix)
    # U matrix is a doubling of elementary matrices that we used to reset organs under the pivot
    U = MultiplyMatrix(U, matrix)
    return U
```

פונקציה למציאת מטריצת ה-L =<

```
def LMatrix(matrix, vector):
    """
    :param matrix: Matrix nxn
    :return: Disassembly into a L matrix
    """
    # Initialize the result matrix
    L = MakeIMatrix(len(matrix), len(matrix))
    # loop for each row
    for i in range(len(matrix[0])):
        # pivoting process
        matrix, vector = RowXchangeZero(matrix, vector)
        for j in range(i + 1, len(matrix)):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            # Finding the M(ij) to reset the organs under the pivot
            elementary[j][i] = -(matrix[j][i]) / matrix[i][i]
            # L matrix is a doubling of inverse elementary matrices
            L[j][i] = (matrix[j][i]) / matrix[i][i]
            matrix = MultiplyMatrix(elementary, matrix)

    return L
```

```
def RowXchangeZero(matrix, vector):
    """
    Function for replacing rows with both a matrix and a vector
    :param matrix: Matrix nxn
    :param vector: Vector n
    :return: Replace rows after a pivoting process
    """
    for i in range(len(matrix)):
        for j in range(i, len(matrix)):
            # The pivot member is not zero
            if matrix[i][i] == 0:
                temp = matrix[j]
                temp_b = vector[j]
                matrix[j] = matrix[i]
                vector[j] = vector[i]
                matrix[i] = temp
                vector[i] = temp_b

    return [matrix, vector]
```

פונקציית עזר להחלפת שורות למציאת מטריצות "L" ו-"U" =<

```
def SolveLU(matrix, vector):
    """
    Function for deconstructing a linear equation by ungrouping LU
    :param matrix: Matrix nxn
    :param vector: Vector n
    :return: Solve Ax=b -> x=U(-1)L(-1)b
    """
    matrixU = UMatrix(matrix, vector)
    matrixL = LMatrix(matrix, vector)
    return MultiplyMatrix(InverseMatrix(matrixU), MultiplyMatrix(InverseMatrix(matrixL), vector))
```

פונקציה פתרה לפי שיטת LU =<

```
def Cond(matrix, invert):
    """
    :param matrix: Matrix nxn
    :param invert: Inverted matrix
    :return: CondA = ||A|| * ||A(-1)|
    """
    print("\n|| A ||max = ", MaxNorm(matrix))
    print("\n|| A(-1) ||max = ", MaxNorm(invert))
    return MaxNorm(matrix)*MaxNorm(invert)
```

פונקציה לבדיקת תנאי מכירב =<

פונקציה למציאת ה`MaxNorm`:

```
def MaxNorm(matrix):
    """
    Function for calculating the max-norm of a matrix
    :param matrix: Matrix nxn
    :return: max-norm of a matrix
    """
    max_norm = 0
    for i in range(len(matrix)):
        norm = 0
        for j in range(len(matrix)):
            # Sum of organs per line with absolute value
            norm += abs(matrix[i][j])
        # Maximum row amount
        if norm > max_norm:
            max_norm = norm

    return max_norm
```

```
def MultiplyMatrix(matrixA, matrixB):
    """
    Function for multiplying 2 matrices
    :param matrixA: Matrix nxn
    :param matrixB: Matrix nxn
    :return: Multiplication between 2 matrices
    """
    # result matrix initialized as singularity matrix
    result = [[0 for y in range(len(matrixB[0]))] for x in range(len(matrixA))]

    for i in range(len(matrixA)):
        # iterate through columns of Y
        for j in range(len(matrixB[0])):
            # iterate through rows of Y
            for k in range(len(matrixB)):
                result[i][j] += matrixA[i][k] * matrixB[k][j]
    return result
```

```
def MakeIMatrix(cols, rows):
    # Initialize a identity matrix
    return [[1 if x == y else 0 for y in range(cols)] for x in range(rows)]
```

להכפלת מטריצות פונקציה=>

פונקציה לייצור מטריצת היחידה =<

```
def InverseMatrix(matrix, vector):
    """
    Function for calculating an inverse matrix
    :param matrix: Matrix nxn
    :return: Inverse matrix
    """

    # Unversible reversible matrix
    if Determinant(matrix, 1) == 0:
        print("Error, Singular Matrix\n")
        return

    # result matrix initialized as singularity matrix
    result = MakeIMatrix(len(matrix), len(matrix))
    # loop for each row
    for i in range(len(matrix[0])):
        # turn the pivot into 1 (make elementary matrix and multiply with the result matrix )
        # pivoting process
        matrix, vector = RowXchange(matrix, vector)
        print("Matrix after exchanging rows for the", i+1, "time:")
        PrintMatrix(matrix)
        print("Matrix after making row", i+1, "a pivot - (row", i+1, ") /", matrix[i][i], ":")
        elementary = MakeIMatrix(len(matrix[0]), len(matrix))
        elementary[i][i] = 1/matrix[i][i]
        result = MultiplyMatrix(elementary, result)
        matrix = MultiplyMatrix(elementary, matrix)
        PrintMatrix(matrix)

        # make elementary loop to iterate for each row and subtract the number below (specific) pivot to zero (make
        # elementary matrix and multiply with the result matrix )
        for j in range(i+1, len(matrix)):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            elementary[j][i] = -(matrix[j][i])
            matrix = MultiplyMatrix(elementary, matrix)
            result = MultiplyMatrix(elementary, result)
        print("Matrix after subtracting with row", i+1, "making the lower part of the column 0")
        PrintMatrix(matrix)

    # after finishing with the lower part of the matrix subtract the numbers above the pivot with elementary for loop
    # (make elementary matrix and multiply with the result matrix )
    for i in range(len(matrix[0])-1, 0, -1):
        for j in range(i-1, -1, -1):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            elementary[j][i] = -(matrix[j][i])
            matrix = MultiplyMatrix(elementary, matrix)
            result = MultiplyMatrix(elementary, result)
        print("Matrix after subtracting with row", i+1, "making the upper part of the column 0")
        PrintMatrix(matrix)
    return result
```

פונקציה להיפוך מטריצה עם הדפסות =<

```

def GaussSeidelMethod(matrix, vector, epsilon, previous, counter):
    NextGuess = []
    ImprovedGuess = CopyVector(previous)
    for i in range(len(matrix)):
        ins = 0
        for j in range(len(matrix)):
            if i != j:
                ins = ins + matrix[i][j]*ImprovedGuess[j]
        newGuess = 1/matrix[i][i]*(vector[i][0]-ins)
        ImprovedGuess[i] = newGuess
        NextGuess.append(newGuess)

    print("Iteration no. "+str(counter)+" "+str(NextGuess))

    for i in range(len(matrix)):
        if abs(NextGuess[i] - previous[i]) < epsilon:
            return NextGuess

    return GaussSeidelMethod(matrix, vector, epsilon, NextGuess, counter+1)

```

פונקציה לפתרת מטריצה לפי שיטת גאואזידל =>

```

def InitVector(size):
    return [0 for index in range(size)]

def CopyVector(vector):
    copy = []
    for i in range(len(vector)):
        copy.append(vector[i])

    return copy

```

```

def CheckDominantDiagonal(matrix):
    for i in range(len(matrix)):
        sum = 0
        for j in range(len(matrix)):
            if i != j:
                sum += abs(matrix[i][j])
        if abs(matrix[i][i]) < sum:
            return False
    return True

```

```

def DominantDiagonalFix(matrix):
    #Check if we have a dominant for each column
    dom = [0]*len(matrix)
    result = list()
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if (matrix[i][j] > sum(map(abs, map(int, matrix[i])))-matrix[i][j]):
                dom[i]=j
    for i in range(len(matrix)):
        result.append([])
        if i not in dom:
            print("Couldn't find dominant diagonal.")
            return matrix
    for i,j in enumerate(dom):
        result[j]=(matrix[i])
    return result

```

פונקציה לתקן אלכסון דומיננטי =>

```

def CheckGeusZaideLgnorm(matrix):
    return 1 > MaxNorm(GeusZaideLg(matrix))

```

```

def GeusZaideLg(matrix):
    D, L, U = matrixDLUdissassembly(matrix)
    return MultiplyMatrix(minusMatrix(InverseMatrix(matrixAddition(L, D, [[[]]])), U))
    #I want to invert but I dont care about a vector so im reusing the function I made and sending an empty vector

```

```

def matrixDLUdissassembly(matrix):
    D, L, U = list(), list(), list()
    for x, row in enumerate(matrix):
        D.append(list())
        L.append(list())
        U.append(list())
        for y, value in enumerate(row):
            if x == y:
                D[x].append(value)
                L[x].append(0)
                U[x].append(0)
            elif x < y:
                D[x].append(0)
                L[x].append(0)
                U[x].append(value)
            elif x > y:
                D[x].append(0)
                L[x].append(value)
                U[x].append(0)
    return D, L, U

```

פונקציה לפירוק מטריצה למטריצות DLU =>

```

def matrixAddition(matrixA, matrixB):
    return [[a + b for (a, b) in zip(i, j)] for (i, j) in zip(matrixA, matrixB)]

```

פונקציה לחיבור מטריצות =>

```

def minusMatrix(matrix):
    return [[-i for i in j] for j in matrix]

```

פונקציה לחיסור מטריצות =>

בחלק זה השתמשנו בשלושת השיטות על מנת לפתור את המטריצה והדפסנו את השלבים והאטרציות של כל שלב עיקרי , שמראה את הדרך למציאת הפתרון בכל אחת מהשיטות.

בתהליך השתמשנו בכלים השונים כגון: voting, בדיקת התכנסות מוציאות נורמה, בדיקת אלכסון דומיננטי, החלפת שורות במידת הצורך. על מנת להציג לתוצאות מיטביות.

```

matrixA = [[1, 2,-2], [1,1,1], [2,2,1]]
vectorb = [[7], [2], [5]]
detA = Determinant(matrixA, 1)
print("\nMatrix A: \n")
PrintMatrix(matrixA)
print("\nVector b: \n")
PrintMatrix(vectorb)
print("\nDET(A) = ", detA)
print("\n---The First method, according to the Elimination of Gauss Method (of course includes the use of pivoting and the calculation of COND)---")
print("\nCondA = ||A|| * ||A(-1)|| = ", Cond(matrixA, InverseMatrixNoPrints(matrixA, vectorb)))
print("\nGaussJordanElimination\n")
JordanEliminationSol = GaussJordanElimination(matrixA, vectorb)
print("\nfinal result for x=(A^-1) * b \n")
PrintVectorFinal(JordanEliminationSol)
print("\n---The Second method, according to the LU Disassembling Method---\n")
PrintMatrix(matrixA)
print("\n-----Building L and U matrices----\n")
luSolution = SolveLU(matrixA, vectorb)
print("Matrix U: \n")
PrintMatrix(UMatrix(matrixA, vectorb))
print("Matrix L: \n")
PrintMatrix(LMatrix(matrixA, vectorb))
print("\nMatrix A=LU: \n")
PrintMatrix(MultiplyMatrix(LMatrix(matrixA, vectorb), UMatrix(matrixA, vectorb)))
print("\nSolve Ax = b: ")
print('LU vector solution:\n')
PrintVectorFinal(luSolution)
print("\n---The Third method, according to the Gauss Seidel Method (including finding dominant diagonal)---")
epsilon = 0.00001
#geuss

if CheckDominantDiagonal(matrixA):
    print("There is a dominant diagonal.")
    seidelSolution = GaussSeidelMethod(matrixA, vectorb, epsilon, InitVector(len(vectorb)), 1)
    seidelSolution = [[a] for a in seidelSolution]
    PrintVectorFinal(seidelSolution)
else:
    print("There isn't a dominant diagonal.")
    print("We will try to find dominant diagonal.")
    dominantFix=DominantDiagonalFix(matrixA)
    PrintMatrix(dominantFix)
    if dominantFix != matrixA:
        print("Found a dominant diagonal.")
        seidelSolution = GaussSeidelMethod(dominantFix, vectorb, epsilon, InitVector(len(vectorb)), 1)
        PrintVectorFinal(seidelSolution)
    else:
        print("didn't find a dominant diagonal.")
        if CheckGeusZaidelGnorm(matrixA):
            print("The matrix convergent.")
            seidelSolution = GaussSeidelMethod(matrixA, vectorb, epsilon, InitVector(len(vectorb)), 1)
            PrintVectorFinal(seidelSolution)
        else:
            print(bcolors.FAIL+"\nThe matrix isn't convergent.\n"+bcolors.ENDC)
            print("Can't solve this matrix by Gauss Seidel Method \nYou can find other solutions above this one (LU "
                  "disassembling and elimination of Gauss)")

```

Numeric Analysis - Final Project

Systems of Linear Equations

Question 30

• שאלה מס' 30

פתרו את המטריצה הבאה בשתי דרכים והשו בין התוצאות

$$\begin{pmatrix} 0 & 1 & 2 \\ -2 & 1 & 0.5 \\ 1 & -2 & -0.5 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ -4 \end{pmatrix}$$

GitHub: <https://github.com/Maor-Ar/Numerical-Analysis-Hackathon/blob/main/Q30%20Systems%20of%20Linear%20Equations.py>

```
"""
# Authors: Maor Arnon (ID: 205974553) and Neriya Zudi (ID: 207073545) and
# Matan Ohayon (ID: 311435614) and Matan Sofri (ID: 208491811)
# Emails: maorar@ac.sce.ac.il | neriyyazudi@gmail.com
# matan15595@gmail.com | sofermatan123@gmail.com
# Department of Computer Engineering - Assignment 2 - Numeric Analytics
"""

from datetime import datetime

class bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[90m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    HEADER = '\033[96m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    # Background colors:
    GREYBG = '\033[100m'
    REDBG = '\033[101m'
    GREENBG = '\033[102m'
    YELLOWBG = '\033[103m'
    BLUEBG = '\033[104m'
    PINKBG = '\033[105m'
    CYANBG = '\033[106m'

local_dt = datetime.now()
dstr=(local_dt.day)
hstr=(local_dt.hour)
mstr=(local_dt.minute)
Five_zeros="00000"
Five_zeros=Five_zeros[0:5]

def PrintMatrix(matrix):
    """
    Matrix Printing Function
    :param matrix: Matrix nxn
    """
    for line in matrix:
        line.append(')')
        line.insert(0,'(')
        print(''.join(map(str, line)))
        line.remove(')')
        line.remove('(')

def PrintVectorFinal(vector):
    """
    Matrix Printing Function
    :param matrix: Matrix nxn
    """
    print('Solution:')
    PrintMatrix(vector)
    print()
    for i in range(len(vector)):
        print("Solution to x"+str(i)+" value, in the required format.:")
        print(str(vector[i][0])+bcolors.FAIL+'00000'+bcolors.OKGREEN+d+bcolors.OKBLUE+h+m+bcolors.ENDC+'\n')

def Determinant(matrix, mul):
    """
    Recursive function for determinant calculation
    :param matrix: Matrix nxn
    :param mul: The double number
    :return: determinant of matrix
    """
    width = len(matrix)
    # Stop Conditions
    if width == 1:
        return mul * matrix[0][0]
    else:
        sign = -1
        det = 0
        for i in range(width):
            m = []
            for j in range(1, width):
                buff = []
                for k in range(width):
                    if k != i:
                        buff.append(matrix[j][k])
                m.append(buff)
            # Change the sign of the multiply number
            sign *= -1
            det += sign * matrix[i][0] * Determinant(m, mul * matrix[i][0])

local_dt = datetime.now()
dstr=(local_dt.day)
hstr=(local_dt.hour)
mstr=(local_dt.minute)
Five_zeros="00000"
Five_zeros=Five_zeros[0:5]

def PrintMatrix(matrix):
    """
    Matrix Printing Function
    :param matrix: Matrix nxn
    """
    for line in matrix:
        line.append(')')
        line.insert(0,'(')
        print(''.join(map(str, line)))
        line.remove(')')
        line.remove('(')

def PrintVectorFinal(vector):
    """
    Matrix Printing Function
    :param matrix: Matrix nxn
    """
    print('Solution:')
    PrintMatrix(vector)
    print()
    for i in range(len(vector)):
        print("Solution to x"+str(i)+" value, in the required format.:")
        print(str(vector[i][0])+bcolors.FAIL+'00000'+bcolors.OKGREEN+d+bcolors.OKBLUE+h+m+bcolors.ENDC+'\n')

def Determinant(matrix, mul):
    """
    Recursive function for determinant calculation
    :param matrix: Matrix nxn
    :param mul: The double number
    :return: determinant of matrix
    """
    width = len(matrix)
    # Stop Conditions
    if width == 1:
        return mul * matrix[0][0]
    else:
        sign = -1
        det = 0
        for i in range(width):
            m = []
            for j in range(1, width):
                buff = []
                for k in range(width):
                    if k != i:
                        buff.append(matrix[j][k])
                m.append(buff)
            # Change the sign of the multiply number
            sign *= -1
            det += sign * matrix[i][0] * Determinant(m, mul * matrix[i][0])
```

צבעים להדפסות =>

נתוני זמן מהמערכת =>

פונקציה להדפסת מטריצה =>

פונקציה להדפסת פלט לפי הדרישה=>

פונקציה לדטרמיננטה=>

```

def InverseMatrixNoPrints(matrix, vector):
    """
    Function for calculating an inverse matrix, Without printing the steps (for checking cond)
    :param matrix: Matrix nxn
    :return: Inverse matrix
    """
    # Unveri reversible matrix
    if Determinant(matrix, 1) == 0:
        print("Error,Singular Matrix\n")
        return
    # result matrix initialized as singularity matrix
    result = MakeIMatrix(len(matrix), len(matrix))
    # Loop for each row
    for i in range(len(matrix[0])):
        # turn the pivot into 1 (make elementary matrix and multiply with the result matrix )
        # pivoting process
        matrix, vector = RowXchange(matrix, vector)
        elementary = MakeIMatrix(len(matrix[0]), len(matrix))
        elementary[i][i] = 1/matrix[i][i]
        result = MultiplyMatrix(elementary, result)
        matrix = MultiplyMatrix(elementary, matrix)
        # make elementary loop to iterate for each row and subtract the number below (specific) pivot to zero (make
        # elementary matrix and multiply with the result matrix )
        for j in range(i+1, len(matrix)):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            elementary[j][i] = -(matrix[j][i])
            matrix = MultiplyMatrix(elementary, matrix)
            result = MultiplyMatrix(elementary, result)
    # after finishing with the lower part of the matrix subtract the numbers above the pivot with elementary for loop
    # (make elementary matrix and multiply with the result matrix )
    for i in range(len(matrix[0])-1, 0, -1):
        for j in range(i-1, -1, -1):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            elementary[j][i] = -(matrix[j][i])
            matrix = MultiplyMatrix(elementary, matrix)
            result = MultiplyMatrix(elementary, result)
    return result

def RowXchange(matrix, vector):
    """
    Function for replacing rows with both a matrix and a vector
    :param matrix: Matrix nxn
    :param vector: Vector n
    :return: Replace rows after a pivoting process
    """
    for i in range(len(matrix)):
        max = abs(matrix[i][i])
        for j in range(i, len(matrix)):
            # The pivot member is the maximum in each column
            if abs(matrix[j][i]) > max:
                temp = matrix[j]
                temp_b = vector[j]
                matrix[j] = matrix[i]
                vector[j] = vector[i]
                matrix[i] = temp
                vector[i] = temp_b
                max = abs(matrix[i][i])

    return [matrix, vector]

def GaussJordanElimination(matrix, vector):
    """
    Function for solving a linear equation using gauss's elimination method
    :param matrix: Matrix nxn
    :param vector: Vector n
    :return: Solve Ax=b -> x=A(-1)b
    """
    # Pivoting process
    # matrix, vector = RowXchange(matrix, vector)
    # print("Matrix after pivoting - ")
    # PrintMatrix(matrix)
    # Inverse matrix calculation
    invert = InverseMatrix(matrix, vector)
    print("Matrix after inversion - ")
    PrintMatrix(invert)
    return MulMatrixVector(invert, vector)

def MulMatrixVector(InversedMat, b_vector):
    """
    Function for multiplying a vector matrix
    :param InversedMat: Matrix nxn
    :param b_vector: Vector n
    :return: Result vector
    """
    result = []
    # Initialize the x vector
    for i in range(len(b_vector)):
        result.append([0])
        result[i].append(0)
    # Multiplication of inverse matrix in the result vector
    for i in range(len(InversedMat)):
        for k in range(len(b_vector)):
            result[i][0] += InversedMat[i][k] * b_vector[k][0]
    return result

```

פונקציה למציאת הופci ביל' הדפסות =>

פונקציה להחלפת שורות לפי ערך דומיננטי (גם לוקטור) =>

פונקציה לפתרית מטריצה לפי שיטת גאוס גordan =>

פונקציה להכפלת מטריצות =>

```

def UMatrix(matrix, vector):
    """
    :param matrix: Matrix nxn
    :return: Disassembly into a U matrix
    """
    # result matrix initialized as singularity matrix
    U = MakeIMatrix(len(matrix), len(matrix))
    # loop for each row
    for i in range(len(matrix[0])):
        # pivoting process
        matrix, vector = RowXchangeZero(matrix, vector)
        for j in range(i + 1, len(matrix)):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            # Finding the M(ij) to reset the organs under the pivot
            elementary[j][i] = -(matrix[j][i]) / matrix[i][i]
            matrix = MultiplyMatrix(elementary, matrix)
    # U matrix is a doubling of elementary matrices that we used to reset organs under the pivot
    U = MultiplyMatrix(U, matrix)
    return U

```

פונקציה למציאת מטריצת ה-"U" =<

```

def LMatrix(matrix, vector):
    """
    :param matrix: Matrix nxn
    :return: Disassembly into a L matrix
    """
    # Initialize the result matrix
    L = MakeIMatrix(len(matrix), len(matrix))
    # loop for each row
    for i in range(len(matrix[0])):
        # pivoting process
        matrix, vector = RowXchangeZero(matrix, vector)
        for j in range(i + 1, len(matrix)):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            # Finding the M(ij) to reset the organs under the pivot
            elementary[j][i] = -(matrix[j][i]) / matrix[i][i]
            # L matrix is a doubling of inverse elementary matrices
            L[j][i] = (matrix[j][i]) / matrix[i][i]
            matrix = MultiplyMatrix(elementary, matrix)

    return L

```

פונקציה למציאת מטריצת ה-"L" =<

```

def RowXchangeZero(matrix, vector):
    """
    Function for replacing rows with both a matrix and a vector
    :param matrix: Matrix nxn
    :param vector: Vector n
    :return: Replace rows after a pivoting process
    """
    for i in range(len(matrix)):
        for j in range(i, len(matrix)):
            # The pivot member is not zero
            if matrix[i][i] == 0:
                temp = matrix[j]
                temp_b = vector[j]
                matrix[j] = matrix[i]
                vector[j] = vector[i]
                matrix[i] = temp
                vector[i] = temp_b

    return [matrix, vector]

```

פונקציית עזר להחלפת שורות למציאת מטריצות "L" ו-"U" =<

```

def SolveLU(matrix, vector):
    """
    Function for deconstructing a linear equation by ungrouping LU
    :param matrix: Matrix nxn
    :param vector: Vector n
    :return: Solve Ax=b -> x=U(-1)L(-1)b
    """
    matrixU = UMatrix(matrix, vector)
    matrixL = LMatrix(matrix, vector)
    return MultiplyMatrix(InverseMatrix(matrixU, vector), MultiplyMatrix(InverseMatrix(matrixL, vector), vector))

```

פונקציה פתרה לפי שיטת LU =<

```

def Cond(matrix, invert):
    """
    :param matrix: Matrix nxn
    :param invert: Inverted matrix
    :return: CondA = ||A|| * ||A(-1) ||
    """
    print("\n|| A ||max = ", MaxNorm(matrix))
    print("\n|| A(-1) ||max = ", MaxNorm(invert))
    return MaxNorm(matrix)*MaxNorm(invert)

```

פונקציה לבדיקת תנאי מכירב =<

```

def MaxNorm(matrix):
    """
    Function for calculating the max-norm of a matrix
    :param matrix: Matrix nxn
    :return: max-norm of a matrix
    """
    max_norm = 0
    for i in range(len(matrix)):
        norm = 0
        for j in range(len(matrix)):
            # Sum of organs per line with absolute value
            norm += abs(matrix[i][j])
        # Maximum row amount
        if norm > max_norm:
            max_norm = norm

    return max_norm

def MultiplyMatrix(matrixA, matrixB):
    """
    Function for multiplying 2 matrices
    :param matrixA: Matrix nxn
    :param matrixB: Matrix nxn
    :return: Multiplication between 2 matrices
    """
    # result matrix initialized as singularity matrix
    result = [[0 for y in range(len(matrixB[0]))] for x in range(len(matrixA))]
    for i in range(len(matrixA)):
        # iterate through columns of Y
        for j in range(len(matrixB[0])):
            # iterate through rows of Y
            for k in range(len(matrixB)):
                result[i][j] += matrixA[i][k] * matrixB[k][j]
    return result

def MakeIMatrix(cols, rows):
    # Initialize a identity matrix
    return [[1 if x == y else 0 for y in range(cols)] for x in range(rows)]

def InverseMatrix(matrix, vector):
    """
    Function for calculating an inverse matrix
    :param matrix: Matrix nxn
    :return: Inverse matrix
    """
    # Unveri reversible matrix
    if Determinant(matrix, 1) == 0:
        print("Error,Singular Matrix\n")
        return
    # result matrix initialized as singularity matrix
    result = MakeIMatrix(len(matrix), len(matrix))
    # loop for each row
    for i in range(len(matrix[0])):
        # turn the pivot into 1 (make elementary matrix and multiply with the result matrix )
        # pivoting process
        matrix, vector = RowXchange(matrix, vector)
        print("Matrix after exchanging rows for the", i+1, "time:")
        PrintMatrix(matrix)
        print("Matrix after making row", i+1, "a pivot - (row", i+1, ")/", matrix[i][i], ":")
        elementary = MakeIMatrix(len(matrix[0]), len(matrix))
        elementary[i][i] = 1/matrix[i][i]
        result = MultiplyMatrix(elementary, result)
        matrix = MultiplyMatrix(elementary, matrix)
        PrintMatrix(matrix)
        # make elementary loop to iterate for each row and subtract the number below (specific) pivot to zero (make
        # elementary matrix and multiply with the result matrix )
        for j in range(i+1, len(matrix)):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            elementary[j][i] = -(matrix[j][i])
            matrix = MultiplyMatrix(elementary, matrix)
            result = MultiplyMatrix(elementary, result)
        print("Matrix after subtracting with row", i+1, "making the lower part of the column 0")
        PrintMatrix(matrix)
    # after finishing with the lower part of the matrix subtract the numbers above the pivot with elementary for loop
    # (make elementary matrix and multiply with the result matrix )
    for i in range(len(matrix[0])-1, 0, -1):
        for j in range(i-1, -1, -1):
            elementary = MakeIMatrix(len(matrix[0]), len(matrix))
            elementary[j][i] = -(matrix[j][i])
            matrix = MultiplyMatrix(elementary, matrix)
            result = MultiplyMatrix(elementary, result)
        print("Matrix after subtracting with row", i+1, "making the upper part of the column 0")
        PrintMatrix(matrix)
    return result

```

פונקציה למציאת ה \max נורם \leq MaxNorm

להכפלת מטריצות פונקציה \leq MultiplyMatrix

פונקציה ליצירת מטריצת היחידה \leq MakeIMatrix

פונקציה להיפוך מטריצה עם הדפסות \leq InverseMatrix

```

def GaussSeidelMethod(matrix, vector, epsilon, previous, counter):
    NextGuess = []
    ImprovedGuess = CopyVector(previous)
    for i in range(len(matrix)):
        ins = 0
        for j in range(len(matrix)):
            if i != j:
                ins = ins + matrix[i][j]*ImprovedGuess[j]
        newGuess = 1/matrix[i][i]*(vector[i][0]-ins)
        ImprovedGuess[i] = newGuess
        NextGuess.append(newGuess)

    print("Iteration no. " +str(counter)+ " " +str(NextGuess))

    for i in range(len(matrix)):
        if abs(NextGuess[i] - previous[i]) < epsilon:
            return NextGuess

    return GaussSeidelMethod(matrix, vector, epsilon, NextGuess, counter+1)

def InitVector(size):
    return [0 for index in range(size)]

def CopyVector(vector):
    copy = []
    for i in range(len(vector)):
        copy.append(vector[i])

    return copy

def CheckDominantDiagonal(matrix):
    for i in range(len(matrix)):
        sum = 0
        for j in range(len(matrix)):
            if i != j:
                sum += abs(matrix[i][j])
        if abs(matrix[i][i]) < sum:
            return False
    return True

def DominantDiagonalFix(matrix):
    #Check if we have a dominant for each column
    dom = [0]*len(matrix)
    result = list()
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if (matrix[i][j] > sum(map(abs, map(int, matrix[i])))-matrix[i][j]):
                dom[i]=j
    for i in range(len(matrix)):
        result.append([])
        if i not in dom:
            print("Couldn't find dominant diagonal.")
            return matrix
    for i,j in enumerate(dom):
        result[j]=(matrix[i])
    return result

def CheckGeusZaideL1norm(matrix):
    return 1 > MaxNorm(GeusZaideL1(matrix))

def GeusZaideL1(matrix):
    D, L, U = matrixDLUDissassembly(matrix)
    return MultiplyMatrix(minusMatrix(InverseMatrix(MatrixAddition(L, D, [[0]]))), U)
    #I want to invert but I dont care about a vector so im reusing the function I made and sending an empty vector

def matrixDLUDissassembly(matrix):
    D, L, U = list(), list(), list()
    for x, row in enumerate(matrix):
        D.append(list()), L.append(list()), U.append(list())
        for y, value in enumerate(row):
            if x == y:
                D[x].append(value), L[x].append(0), U[x].append(0)
            elif x < y:
                D[x].append(0), L[x].append(0), U[x].append(value)
            elif x > y:
                D[x].append(0), L[x].append(value), U[x].append(0)
    return D, L, U

def matrixAddition(matrixA, matrixB):
    return [[a + b for (a, b) in zip(i, j)] for (i, j) in zip(matrixA, matrixB)]

def minusMatrix(matrix):
    return [[-i for i in j] for j in matrix]

```

פונקציה לפתרת מטריצה לפי שיטת גאואזידל =>

פונקציה ליצירת מטריצת אפסים =>

פונקציה להעתיקת וקטור =>

פונקציה לבדיקת אלכסון דומיננטי =>

פונקציה לתיקון אלכסון דומיננטי =>

פונקציית עזר לגאואזידל לבדיקת Gnorm =>

פונקציה לפתרת מטריצה לשיטת גאואזידל =>

פונקציה לפירוק מטריצה למטריצות DLU =>

פונקציה לחיבור מטריצות =>

פונקציה לחיסור מטריצות =>

בחלק זה השתמשנו בשלושת השיטות על מנת לפתור את המטריצה והדפסנו את השלבים והאטרציות של כל שלב עיקרי , שמראה את הדרך למציאת הפתרון בכל אחת מהשיטות.

בתהליך השתמשנו בכלים השונים כגון: voting, בדיקת התכנסות מוציאות נורמה, בדיקת אלכסון דומיננטי, החלפת שורות במידת הצורך. על מנת להציג לתוצאות מיטביות.

```

matrixA = [[0, 1, 2], [-2, 1, 0.5], [1, -2, -0.5]]
vectorb = [[0], [4], [-4]]
detA = Determinant(matrixA, 1)
print("\nMatrix A: \n")
PrintMatrix(matrixA)
print("\nVector b: \n")
PrintMatrix(vectorb)
print("\nDET(A) = ", detA)
print("\n---The First method according to the elimination of Gauss (of course includes the use of pivoting and the calculation of COND)---")
print("\nCondA = ||A|| * ||A(-1)|| = ", Cond(matrixA, InverseMatrixNoPrints(matrixA, vectorb)))
print("\nGaussJordanElimination\n")
JordanEliminationSol = GaussJordanElimination(matrixA, vectorb)
print("\nfinal result for x=(A^-1) * b \n")
PrintVectorFinal(JordanEliminationSol)
print("\n---The Second method according to the LU dismantling---\n")
PrintMatrix(matrixA)
print("\n----Building L and U matrices----\n")
luSolution = SolveLU(matrixA, vectorb)
print("Matrix U: \n")
PrintMatrix(UMatrix(matrixA, vectorb))
print("\nMatrix L: \n")
PrintMatrix(LMatrix(matrixA, vectorb))
print("\nMatrix A=LU: \n")
PrintMatrix(MultiplyMatrix(LMatrix(matrixA, vectorb), UMatrix(matrixA, vectorb)))
print("\nSolve Ax = b: ")
print('LU vector solution:\n')
PrintVectorFinal(luSolution)
print("\n---The Third method according to the Gauss Seidel Method (including finding dominant diagonal)----")
epsilon = 0.00001
#geuss
if CheckDominantDiagonal(matrixA):
    print("There is a dominant diagonal.")
    seidelSolution = GaussSeidelMethod(matrixA, vectorb, epsilon, InitVector(len(vectorb)), 1)
    seidelSolution = [[a] for a in seidelSolution]
    PrintVectorFinal(seidelSolution)
else:
    print("There isn't a dominant diagonal.")
    print("We will try to find dominant diagonal.")
    dominantFix=DominantDiagonalFix(matrixA)
    PrintMatrix(dominantFix)
    if dominantFix != matrixA:
        print("Found a dominant diagonal.")
        seidelSolution = GaussSeidelMethod(dominantFix, vectorb, epsilon, InitVector(len(vectorb)), 1)
        PrintVectorFinal(seidelSolution)
    else:
        print("didn't find a dominant diagonal.")
        if CheckGeusZaielGnorm(matrixA):
            print("The matrix convergent.")
            seidelSolution = GaussSeidelMethod(matrixA, vectorb, epsilon, InitVector(len(vectorb)), 1)
            PrintVectorFinal(seidelSolution)
        else:
            print(bcolors.FAIL+"\nThe matrix isn't convergent.\n"+bcolors.ENDC)
            print("Can't solve this matrix by Gauss Seidel Method \nYou can find other solutions above this one (LU "
                  "disassembling and elimination of Gauss)")

```

Numeric Analysis - Final Project

Interpolation and Extrapolation

Question 32

שאלה מס' 32

נתונה הטבלה הבאה :

x	F(x)
0.2	13.7241
0.35	13.9776
0.45	14.0625
0.6	13.9776
0.75	13.7241
0.85	13.3056
0.9	12.7281

העריכו את $F(0.65)$ באמצעות שתי שיטות אינטראפולציה שונות

GitHub: <https://github.com/Maor-Ar/Numerical-Analysis-Hackathon/blob/main/Q32%20Interpolation%20and%20Extrapolation.py>

שיטות למציאת קירוב לנקודה לפי ערכי טבלה ע"י שימוש ב **שיטת אינטראפולציה לגראנט'**, אלגוריתם נoil מימוש אינטראפולציה לגראנט

בכלט לפונקציה המממשת נשלח את הנקודה שבעורה מחפשים את הערך ובנוסף מערך זו מימדי שמכיל בכל אינדקס את ערך x וערך y נחשב פולינום מדרגה 1-7 כאשר זה מספר ערכי הטבלה.

אצלנו בשאלת יש 7 נקודות אז יהיה פולינום מסדר 6, עבור כל נקודה ניצור פולינום בהתאם לנוסחה המוגדרת ליצירת

$$L_1(x) = \prod_{\substack{j=0 \\ i \neq j}}^n \frac{x - x_j}{x_i - x_j}$$

הפולינום-----<
 $p_n(x) = \sum_{i=0}^n L_i(x)y_i$ -----> ולאחר מכן ידי חיבור של כל הפולינומיים והצבה של ערך y על פי הנוסחה --->

נקבל את הפתרון (במובן שם גם נציב את ערך x שבעורו מחפשים)

הfonקציה שמחשבת :
 העורות על דרך החישוב בfonקציה עצמה)

```
def Lagrange_interpolation(Table,Point_To_Find): ##calculate by lagrange
    result=0 ##final result
    str1=" ##each iteration the string will include"
    finalresult="" ##include all polinoms
    mul=[] ##each iteration includes all xi-xj
    print("----Lagrange_interpolation---")
    print("Table values are : ")
    for i in range(len(Table)):##print the table
        print("x"+str(i)+" is "+str(Table[i][0])+" F(X) is --> "+str(Table[i][1]))
    for i in range(len(Table)):##running for each table index
        val=1
        for j in range(len(Table)):
            if i!=j: ##using the formula according to i!=j
                str1+=(x-*str(Table[j][0]))
                mul.append((Table[i][0]-Table[j][0]))
                if(j==len(Table)-1 or i==len(Table)-1):
                    str1+="/"
                else:
                    str1+=","
            val = val * (Point_To_Find - Table[j][0]) / (Table[i][0] - Table[j][0]) ##calculate by formula x-xj/xi-xj
        print(bcolors.OKBLUE+"L"+str(i)+" ="+bcolors.OKGREEN+str1+str(multiplyList(mul))+bcolors.ENDC)
        finalresult+= str1+str(multiplyList(mul))+"*"+str(Table[i][1])
        mul.clear()
        str1=""
        result+= val * Table[i][1] ##calculate after inner loop the final result
    print("Final polinom is " + finalresult)
    print("Final solution by Lagrange interpolation " + str(round(result,6))+ bcolors.FAIL + '00000' + bcolors.OKGREEN + d + bcolors.OKBLUE + h + m + bcolors.ENDC + '\n')
```

fonקציה עוזר שמחשבת באיטרציות הפנימיות מכפלה של כל $x_j - x_i$ בכל איטרציה

```
def multiplyList(myList): ##function to calculate multiply of all elements
    result = 1
    for x in myList:
        result = result * x
    return result
```

ימוש אלגוריתם נויל:

בקלט לfonקציה המממשת נשלח את הנקודה שעבורה מחפשים את הערך ובנוסף מערך דו מימדי שמכיל בכל אינדקס את ערכי x ו y נחשב בfonקציה את כל פעם את הפולינום בשלבים וככל פעם נעשה חישובים עבור כל ה"תתי קומבינציות" בטבלה לפי הנוסחה עד שנגיע ל06P. (6 מכיוון שננתנו לנו 7 נקודות בטבלה)
 הרעיון באלגוריתם זה, שיש את חישובי היסוד וכל חישוב משתמש בפתרונות חישובים קודמים.

הfonקציה המחשבת (העורות על דרך חישוב בfonקציה עצמה)

```
def Neville_interpolation(Table,pointToFindVal): ##calcuante Nevil algorithm by the nested formula
    print("----Neville's_interpolation----")
    length = len(Table)
    result = 0 ##final result
    for j in range(1,length):
        for i in range(length-1,j-1,-1): ##running for each pair combination in the table and calculate new value at table
            nevPrevSol = Table[i][1]
            EQ1=((pointToFindVal-Table[i-j][0])*Table[i][1] - (pointToFindVal-Table[i][0])*Table[i-1][1])
            EQ2=(Table[i][0]-Table[i-j][0])
            Table[i][1] =EQ1/EQ2
            print(bcolors.OKBLUE+"P"+str(i)+str(j)+" ="+bcolors.ENDC+"(x-"+str(Table[i-j][0])+")*"+str(nevPrevSol)+"- (x-"+str(Table[i][0])+")*"+str(Table[i-1][1])+")/(" +str(Table[i][0])+ "-"+str(Table[i-j][0])+") ="+bcolors.OKGREEN +str(Table[i][1])+bcolors.ENDC)
    result = Table[length-1][1]
    print("Final solution by Neville's interpolation " + str(round(result,6)) + bcolors.FAIL + '00000' + bcolors.OKGREEN + d + bcolors.OKBLUE + h + m + bcolors.ENDC + '\n')
```