מקבץ קודים מסכמים – אנליזה נומרית

תיאור המסמך : מקבץ קודים לשיטות נומריות שנלמדו במהלך קורס אנליזה נומרית.

תנאים מקדימים להרצת הקודים :

- התקנת Python מגרסת 3.6 ומעלה.
- התקנת IDE תומך שפת Python, כדוגמת – PyCharm.
- ייבוא סיפריות Scipy ו- Numpy.
- הבנת השיטות שנלמדו בהרצאות.

מחברים : שלי מירון, אור ממן, איוון רובינסון וסתיו לובל.

חלק א':

שיטות למציאת פתרון של משוואה לא ליניארית

1) שיטת החצייה

```
def findRoots(f, range_start, range_end, acceptable_error = 0):
   """
   Finds the root of a polynomial based on range.
   Input
        f                 : polynomial/ function
        range_start       : start range of interval
        range_end         : end range of interval
        acceptable_error : the acceptable error to stop the loop
   Output
        m : the final root of a polynomial based on bisection algo

   """
   count = 1
   m = (range_start + range_end) / 2.0
   while (range_end - range_start) / 2.0 > acceptable_error:
      print("Iteration num:", count, ", result =", m)
      if f(m) == 0:
         return m
      elif f(range_start) * f(m) < 0:
         range_end = m
      else:
         range_start = m
      m = (range_start + range_end) / 2.0
      count += 1
   return m
```

מקור : http://code.activestate.com/recipes/578417-bisection-method-in-python/

2) שיטת המיתר

```
import math
def findRoots(f, range_start, range_end, iterations=10):
    """
    Finds the root of a polynomial based on range.
    Input
           f                 : polynomial/ function
           range_start       : start range of interval
           range_end         : end range of interval
           iterations        : number of iteration until
    Output
           m : the final root of a polynomial based on secant algo

    """
    for i in range(iterations):
        print("Iteration num:", i, ", result = ", range_end)
        if f(range_end) - f(range_start) == 0:
                return range_end
        x_temp = range_end - (f(range_end) * (range_end -
                 range_start)* 1.0) / (f(range_end) -
                 f(range_start))
        range_start = range_end
        range_end = x_tem
    return range_end
```

מקור: http://code.activestate.com/recipes/578420-secant-method-of-solving-equtions-in-python/

3) שיטת ניוטון-רפסון

```
from math import *

def findRoots(f, derivative, x0=1):

   """
   Finds the root of a polynomial based on range.
   Input
         f                 : polynomial/ function
         derivative        : A derivative of a polynomial
         x0                : a guess of x
   output
         x : the final root of a polynomial based on newton-repson
             algo
   """
   acceptable_error = 1e-3
   x = float(x0)
   while abs(f(x)) > acceptable_error:
      x = x - f(x) / derivative(x)
   return x
```

חלק ב':

פיתרון נומרי של מערכות משוואות ליניאריות

1) שיטת גאוס

```
def gauss(A):
    """
    Solves systems of linear equations using Gauss algo
    Input
         A : the matrix with the solutions of it
    output
         x : vector that contains the solutions of the equations
    """
    n = len(A)

    for i in range(0, n):
        # Search for maximum in this column
        maxEl = abs(A[i][i])
        maxRow = i
        for k in range(i+1, n):
            if abs(A[k][i]) > maxEl:
                maxEl = abs(A[k][i])
                maxRow = k

        # Swap maximum row with current row (column by column)
        for k in range(i, n+1):
            tmp = A[maxRow][k]
            A[maxRow][k] = A[i][k]
            A[i][k] = tmp

        # Make all rows below this one 0 in current column
        for k in range(i+1, n):
            c = -A[k][i]/A[i][i]
            for j in range(i, n+1):
                if i == j:
                    A[k][j] = 0
                else:
                    A[k][j] += c * A[i][j]

    # Solve equation Ax=b for an upper triangular matrix A
    x = [0 for i in range(n)]
    for i in range(n-1, -1, -1):
        # Round - approximation
        x[i] = round(A[i][n]/A[i][i],3)
        for k in range(i-1, -1, -1):
            A[k][n] -= A[k][i] * x[i]
    return x
```

חלק ג':

שיטות איטרטיביות לפתרון של מערכות ליניאריות

1) שיטת יעקובי

```
import scipy
import numpy as np


def Jacobi(A, b, x, n):
    D = np.diag(A)
    R = A - np.diagflat(D)

    for i in range(n):
        x = (b - np.dot(R, x)) / D
        print("Iteration {0}: {1}".format(i, x))
    return x
```

מקור:
https://austingwalters.com/jacobi-method/

2) שיטת גאוס-זיידל

```
import numpy as np
from scipy.linalg import solve

def gaussSeidel(A, b, x, n):
    """
    Solves systems of linear equations using Gauss-zidel algo
    Input
          A                 : matrix of linear equations
          b                 : solutions of linear equations
          x                 : vector that contains the solutions of
                              the equations
          n                 : number of iteration
    Output
         x : vector that contains the solutions of the equations
    """
    L = np.tril(A)
    U = A - L
    for i in range(n):
        x = np.dot(np.linalg.inv(L), b - np.dot(U, x))
        print ('\n','Iter ', i, ':')
        print(x)
    return x
```

מקור: https://austingwalters.com/gauss-seidel-method/

חלק ד':

שיטות אינטרפולציה

1) שיטת האינטרפולציה לפי לאגרנז'

```
import scipy.interpolate as interpol
    """
    We calculate lagrange interpolation by sending 3 points and
    receiving function back
    Input
                xp: input x's in a list of size n
                yp: input y's in a list of size n
    Output
                f : the polynomial of degree n-1
    """

f = interpol.lagrange(xp, yp)
print(f)
print('f({0}) = {1}'.format(x , f(x)))
```

מקור: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.interpolate.lagrange.html

2) שיטת אינטרפולציה לפי נוויל

```
def neville(datax, datay, x):
    """
    Finds an interpolated value using Neville's algorithm.
    Input
      datax: input x's in a list of size n
      datay: input y's in a list of size n
      x: the x value used for interpolation
    Output
      p[0]: the polynomial of degree n
    """
    n = len(datax)
    p = n*[0]
    for k in range(n):
        for i in range(n-k):
            if k == 0:
                p[i] = datay[i]
            else:
                p[i] = ((x-datax[i+k])*p[i]+ \
                        (datax[i]-x)*p[i+1])/ \
                        (datax[i]-datax[i+k])
            print('P{0}{1} = {2}'.format(i, k, p[i]))
    return ('Result => P{0}{1}({3}) = {2}'.format(i, k, p[0],x))
```

מקור:
https://github.com/gisalgs/geom/blob/master/neville.py#L23

3) שיטת ספליין-קובי

```
import GaussAlgo
import Functions


def CubicSplineDerivatives(x_values, y_values, first_derivative,
last_derivative):
    """
        Solves for the vector of derivatives of the spline function.
        Parameters:
            x_values - sorted array of floats
            y_values - array of floats
            first_derivative - derivative of spline function at the
1st x_value
            last_derivative - derivative of spline function at the
last x_value
        Returns:
            tuple of derivatives for each range

        Please note that it may be broken for non-natural cubic
splines
    """
    x_values = tuple(x_values)
    y_values = tuple(y_values)
    if len(x_values) != len(y_values):
        raise Exception("x_values and y_values length mismatch")
    if x_values != tuple(sorted(x_values)):
        raise Exception("x_values not sorted in ascending order")

    intervals = []
    for i in range(len(x_values) - 1):
        intervals.append(x_values[i + 1] - x_values[i])

    matrix = ()

    # Presentation slide 7
    a00 = 1  # intervals[0]/3
    a01 = 0  # intervals[0]/6
    ann1 = 0  # intervals[len(intervals)-1]/6
    ann = 1  # intervals[len(intervals)-1]/3
    d0 = 0  # (y_values[1] - y_values[0])/intervals[0] -
first_derivative
    dn = 0  # last_derivative - (y_values[len(y_values)-1] -
y_values[len(y_values)-2])/intervals[len(intervals) - 1]

    # Presentation slide 8
    matrix += ((a00, a01) + tuple(0 for _ in range(len(x_values) -
2)) + (d0,),)
    for i in range(1, len(x_values) - 1):
        matrix += (tuple(0 for _ in range(i - 1)) + (
        intervals[i - 1] / 6, (intervals[i - 1] + intervals[i]) / 3,
intervals[i] / 6) + tuple(
            0 for _ in range(len(x_values) - i - 2)) + (
                   (y_values[i + 1] - y_values[i]) / intervals[i] -
(y_values[i] - y_values[i - 1]) / intervals[
                       i - 1],),)
    matrix += (tuple(0 for _ in range(len(x_values) - 2)) + (ann1,
```

```
ann) + (dn,),)

    return GaussAlgo.gauss(matrix, 7)


def CubicSpline(x_values, y_values, derivative_at_x1,
derivative_at_xn):
    """
        Performs cubic-spline interpolation of unknown function,
described by x_values and y_values.
        Parameters:
            x_values - sorted array of floats
            y_values - array of floats
            derivative_at_x1 - derivative of function at the 1st
x_value
            derivative_at_xn - derivative of function at the last
x_value
        Returns:
            tuple, where each element is a
            tuple of coefficients
            of resulting polynomial
            for x[i] < x <= x[i+1]
            in increasing order.
            coefficients[0] is coefficient of x^0
            coefficients[1] is coefficient of x^1
            etc...
    """
    x_values = tuple(x_values)
    y_values = tuple(y_values)
    if len(x_values) != len(y_values):
        raise Exception("x_values and y_values length mismatch")
    if x_values != tuple(sorted(x_values)):
        raise Exception("x_values not sorted in ascending order")

    derivatives = CubicSplineDerivatives(x_values, y_values,
derivative_at_x1, derivative_at_xn)

    polynomials = ()
    for i in range(len(x_values) - 1):
        interval_size = x_values[i + 1] - x_values[i]
        if interval_size == 0:
            raise Exception("interval size can not be 0")

        coefficients = (
            # Formula for S_i taken from presentation slide 11, and
ran through WolframAlpha
                        (x_values[i] * (x_values[i] ** 2 *
derivatives[i + 1] - 6 * y_values[i + 1] - derivatives[
                i + 1] * interval_size ** 2) + x_values[i + 1] * (
                         derivatives[i] * interval_size ** 2 + 6 *
y_values[i] - x_values[i + 1] ** 2 * derivatives[
                     i])) / (6 * interval_size),
            (derivatives[i] * (3 * x_values[i + 1] ** 2 -
interval_size ** 2) + 6 * (y_values[i + 1] - y_values[i]) +
             derivatives[i + 1] * interval_size ** 2 - 3 *
x_values[i] ** 2 * derivatives[i + 1]) / (6 * interval_size),
            (x_values[i] * derivatives[i + 1] - x_values[i + 1] *
derivatives[i]) / (2 * interval_size),
            (derivatives[i + 1] + derivatives[i]) / (6 *
interval_size)
        )
```

```
        polynomials += (coefficients,)

    return polynomials


def NaturalCubicSpline(x_values, y_values):
    return CubicSpline(x_values, y_values, 0, 0)


def Interpolate(x_values, y_values, derivative_at_x1,
derivative_at_xn, desired_x):
    """
        Performs cubic-spline interpolation,
        and returns the value of the function at the desired_x.
        Does not perform extrapolation - desired_x must be between
the 1st x_values and the last.
        The rest of the parameters are the same as in CubicSpline
    """
    funcs = CubicSpline(x_values, y_values, derivative_at_x1,
derivative_at_xn)
    for i in range(len(x_values) - 1):
        if x_values[i] <= desired_x and desired_x <= x_values[i + 1]:
            return Functions.evaluateFunction(funcs[i], desired_x)
    raise Exception("desired_x out of range")


def InterpolateNatural(x_values, y_values, desired_x):
    return Interpolate(x_values, y_values, 0, 0, desired_x)
```

חלק ה':

שיטות אינטגרציה וגזירה נומריות

1) שיטת הטרפז

```
import numpy as np

def calculate_area(f, a, b, n):
    """
    Calculate the integral of a f(x) based on the trappezodial rule.
    Input
                f    : the polynomial/ function
                a    : the start range of an integral
                b    : the end range of an integral
                n    : number of interval
    Output
                np.trapz(f(x), x): the integral of f(x)
    """
    x = np.linspace(a, b, n + 1)
    print("number of intervals: ", n+1)
    return np.trapz(f(x), x)
```

מקור: https://codereview.stackexchange.com/questions/194184/definite-integral-approximation-using-the-trapezoidal-method

2) שיטת סימפסון

```
from scipy import integrate

def simpson(y, x):
    """
    Calculate the integral of a f(x) based on the simpson rule.
    Input
                y    : y's points - y range of a polynomial
                x    : x's points - x range of a polynomial
    Output
                Integral of a polynomial based on particular points
    """

    return integrate.simps(y, x)

print("Integral:", simpson(y, x))
```

מקור:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.simps.html

3) שיטת רומברג

```
from scipy import integrate
import numpy as np

def romberg(f, a, b):
    """
    Calculate the integral of a f(x) based on the romberg rule.
    Input
                f    : polynomial/ function
                a    : x range of integral
                b    : y range of integral
    Output
                Integral of a polynomial based on range
    """

    return integrate.romberg(f, a, b, show=True)
print("Integral: ", romberg(f, a, b))
```

מקור:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.romberg.html

4) תרבועי גאוס

```
from scipy import integrate

"""
Returns:
val : float
Gaussian quadrature approximation (within tolerance) to integral.

err : float
Difference between last two estimates of the integral.

"""
result = integrate.quadrature(f, a, b)
print(result)
```

מקור: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.integrate.quadrature.html

נספחי קוד

1) חישוב מטריצה הופכית

```
def invert_matrix(A):

        return linalg.inv(A)
```

2) חישוב נורמה של מטריצה

```
def Norma(A):

    sum = 0
    temp_sum = 0
    for i in range (len(A)):
        if temp_sum >= sum:
            sum = temp_sum
        temp_sum=0
        for j in range (len(A)):
            temp_sum += abs(A[i][j])
    return sum
```

3) חישוב Cond

```
def cond(A):

    return Norma(A)* Norma(invert_matrix(A))
```

4) חישוב LU

```
import pprint
import scipy


P, L, U = scipy.linalg.lu(A)
```

5) חישוב SOR

```
import numpy as np
# Define function
def solveBySOR(A, b, omegaVal, totlVal):
    #    Actual_1 = [1.0,-1.0,3.0]
    #    Actual_2 = [1.0, 2.0, -1.0, 1.0]
    #    Actual_3 = [[3.0,4.0,-5.0]]

    Asize = np.shape(A)
    rwsize = Asize[0]
    colsize = Asize[1]

    if rwsize != colsize:
        print("A is not a square matrix")
        exit(1)
```

```
if rwsize != b.size:
    print("Dimensions of A and b do not match")
    exit(1)

x = np.zeros((rwsize, 1))
x0 = np.zeros((rwsize, 1))
nk = 0
err = totlVal + 1.0
maxIter = 200.0

while err > totlVal and nk < maxIter:
    nk += 1
    for i in range(0, rwsize):
        x0[i] = x[i]
        mysum = b[i]
        oldX = x[i][0]

        for j in range(0, rwsize):
            if i != j:
                mysum = mysum - A[i][j] * x[j][0]

        x0[i] = x[i]
        mysum = b[i]
        oldX = x[i][0]

        for j in range(0, rwsize):
            if i != j:
                mysum = mysum - A[i][j] * x[j][0]

        mysum = mysum / A[i][i]
        x[i][0] = mysum
        x[i][0] = mysum * omegaVal + (1.0 - omegaVal) * oldX

    diff = np.subtract(x, x0)
    err = np.linalg.norm(diff) / np.linalg.norm(x)
    print(np.linalg.norm(err))

if (nk == maxIter):
    print("Maximum number of Iterations exceeded")
else:
    print("The solution is:")
    print(x)
    print("The number of iterations used: %d" % (nk))
    print("Relative error: %.7f" % (err))
```

מקור: https://github.com/lathestudent/Direct-and-Iterative-Solver-of-Linear-Systems/blob/master/Matrix_Solver_Methods.py

טבלאות נתונים ותרשימים להוכחת נכונות השיטות

שיטות למציאת פתרון של משוואה לא ליניארית

בהינתן הקלט הבא:

```
findRoots(lambda x: x**2 - 2*x, 0.5, 2.1, 0.01)
findRoots(lambda x: x**2-2*x, 1, 10, 100)
findRoots(lambda x: x**2-2*x, lambda x: 2*x-2, 4)
```

התוצאות שהתקבלו בכל אחת מן השיטות:

Bisection algorithm

```
Iteration num: 1 , result = 1.3

Iteration num: 2 , result = 1.7000000000000002

Iteration num: 3 , result = 1.9000000000000001

Iteration num: 4 , result = 2.0
result: 2.0
```

Secant algorithm

```
Iteration num: 0 , result =  1.1111111111111107
Iteration num: 1 , result =  1.2195121951219507
Iteration num: 2 , result =  4.0983606557377
Iteration num: 3 , result =  1.5063870812243914
Iteration num: 4 , result =  1.7126628536854314
Iteration num: 5 , result =  2.1163474304073358
Iteration num: 6 , result =  1.9817218421760492
Iteration num: 7 , result =  1.998986393479383
Iteration num: 8 , result =  2.000009353654224
Iteration num: 9 , result =  1.99999995257156
Iteration num: 10 , result =  1.9999999999999778
Iteration num: 11 , result =  2.0
Iteration num: 12 , result =  2.0
2.0
```

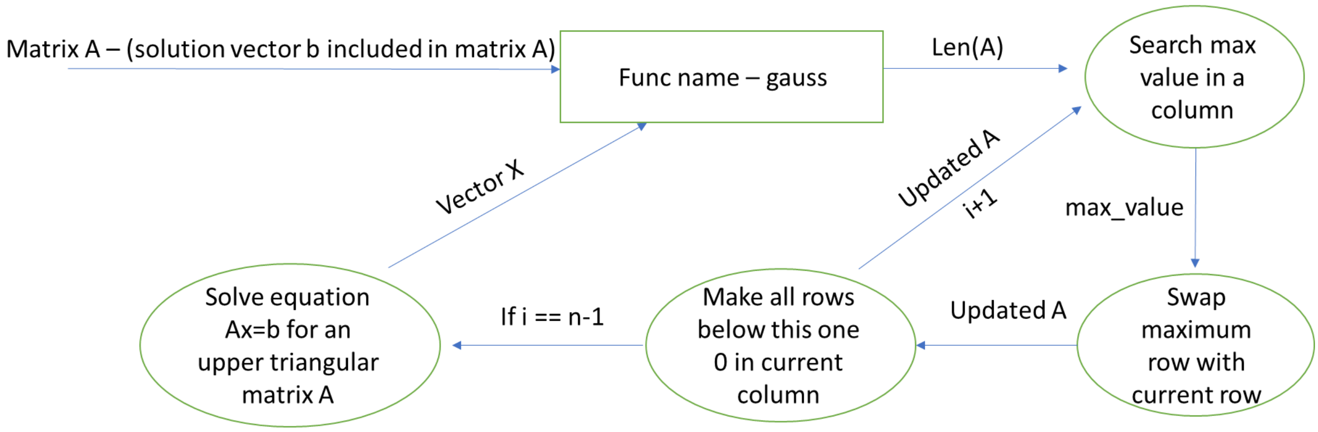
NewtonRepson algorithm

```
Iteration 0 = 2.666666666666667
Iteration 1 = 2.1333333333333337
Iteration 2 = 2.00784313725490 2
Iteration 3 = 2.0000305180437934
result: 2.0000305180437934
```

פתרון נומרי של מערכות משוואות ליניאריות

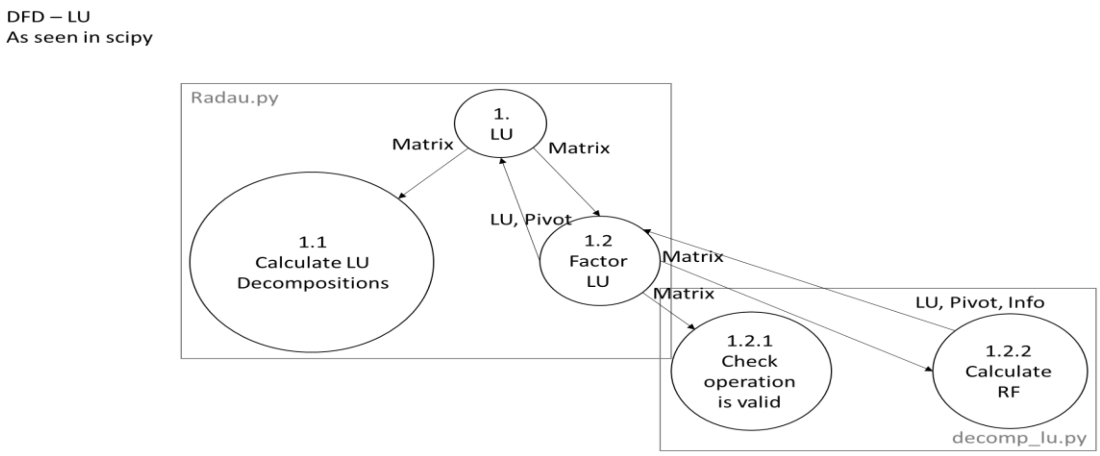
Gauss algorithm

תרשים המציג את פעולת האלגוריתם לפתרון משוואות ליניאריות ע"פ גאוס :



LU

תרשים המציג את אופן פעולת הפירוק LU :



שיטות איטרטיביות לפתרון מערכות ליניאריות

בהינתן הקלט הבא :

```
Jacobi(np.array([[5.0, -1.0, 2.0], [3.0, 8.0, -2.0], [1.0,
1.0, 4.0]]), [12, -25, 6.0], np.array([0.0, 0.0, 0.0]), 22)

A = np.array([[5.0, -1.0, 2.0], [3.0, 8.0, -2.0], [1.0, 1.0,
4.0]])
b = [12, -25, 6.0]
x = [0, 0, 0]

n = 10

gaussSeidel(A, b, x, n)
```

Jacobi algorithm

```
Iteration 0: [ 2.4    -3.125  1.5   ]
Iteration 1: [ 1.175    -3.65      1.68125]
Iteration 2: [ 0.9975     -3.1453125   2.11875   ]
Iteration 3: [ 0.9234375   -2.969375    2.03695313]
Iteration 4: [ 0.99134375 -2.96205078  2.01148438]
Iteration 5: [ 1.00299609 -2.99388281  1.99267676]
Iteration 6: [ 1.00415273 -3.00295435  1.99772168]
Iteration 7: [ 1.00032046 -3.00212686  1.9997004 ]
Iteration 8: [ 0.99969447 -3.00019507  2.0004516 ]
Iteration 9: [ 0.99978035 -2.99977253  2.00012515]
Iteration 10: [ 0.99999543 -2.99988634  1.99999804]
Iteration 11: [ 1.00002351 -2.99999878  1.99997273]
Iteration 12: [ 1.00001115 -3.00001564  1.99999382]
Iteration 13: [ 0.99999935 -3.00000573  2.00000112]
Iteration 14: [ 0.99999841 -2.99999947  2.0000016 ]
Iteration 15: [ 0.99999947 -2.999999    2.00000027]
Iteration 16: [ 1.00000009 -2.99999973  1.99999988]
Iteration 17: [ 1.0000001  -3.00000006  1.99999991]
Iteration 18: [ 1.00000002 -3.00000006  1.99999999]
Iteration 19: [ 0.99999999 -3.00000001  2.00000001]
Iteration 20: [ 0.99999999 -2.99999999  2.        ]
Iteration 21: [ 1. -3.  2.]
```

Gauss-Seidel algorithm

```
 Iter  0 :
x=> [ 2.4       -4.025     1.90625]

 Iter  1 :
x=> [ 0.8325      -2.960625    2.03203125]

 Iter  2 :
x=> [ 0.9950625  -2.99014062  1.99876953]

 Iter  3 :
x=> [ 1.00246406 -3.00123164  1.99969189]

 Iter  4 :
x=> [ 0.99987691 -3.00003087  2.00003849]

 Iter  5 :
x=> [ 0.99997843 -2.99998229  2.00000096]

 Iter  6 :
x=> [ 1.00000316 -3.00000094  1.99999945]

 Iter  7 :
x=> [ 1.00000003 -3.00000015  2.00000003]

 Iter  8 :
x=> [ 0.99999996 -2.99999998  2.        ]

 Iter  9 :
x=> [ 1. -3.  2.]
```

שיטות אינטרפולציה

בהינתן הקלט הבא :

```
# Example - some points in an array
points_table = [(2, -3.6), (3, 1.25), (6, 4.1)]
#points_table = [(0.2, 0.198669), (0.3, 0.295520), (0.4, 0.389418),
(0.5, 0.479426)]

# We choose 3 points from the table, so that the function f(x) will
be in order 2
xp = [points_table[0][0], points_table[1][0], points_table[2][0]]
yp = [points_table[0][1], points_table[1][1], points_table[2][1]]

# We calculate lagrange interpolation by sending 3 points and
receiving function back
f = interpol.lagrange(xp, yp)
x = 4
print('f({0}) = {1}'.format(x, f(x)))


p = neville(xp, yp, 4)
print(p)
```

lagrange interpolation

```
P0(x) =
       2
-0.9 x + 8.1 x - 16.2

P1(x) =
         2
-1.317 x + 11.43 x - 21.2

P2(x) =
         2
-0.975 x + 9.725 x - 19.15
f(4) = 4.150000000000002
```

Neville interpolation

```
P00 = -3.6
P10 = 1.25
P20 = 4.1
P01 = 6.1
P11 = 2.1999999999999997
P02 = 4.1499999999999995
Result => P02(4) = 4.1499999999999995
```

בהינתן הקלט:

```
# Example - some points in an array
points_table = [(2, -3.6), (3, 1.25), (6, 4.1)]


def InterpolateNatural(x_values, y_values, desired_x):
    return Interpolate(x_values, y_values, 0, 0, desired_x)


# x values, y values, the x we want to calculate its y
print(InterpolateNatural([1, 2, 3, 4, 5], [1, 2, 1, 1.5, 1], 5))

# the polynomial of each 2 dots
index = 0
for i in tuple(CubicSpline([1, 2, 3, 4, 5], [1, 2, 1, 1.5, 1], 0,0)):
    print("s{0} = {1}".format(index, i))
    index += 1
```

Cubi-spline interpolation

```
1.0000000000000213
s0 = (0.0, 2.321428566666667, -1.98214285, -0.6607142833333334)
s1 = (23.7142857, -25.250000016666664, 9.80357145, -0.017857133333333348)
s2 = (-48.92857180000001, 41.392857416666665, -11.41071435, 0.23214286666666664)
s3 = (52.785714000000006, -30.892856966666667, 6.16071425, -0.4107142833333333)
```

שיטות גזירה ואינטגרציה נומרית

בהינתן הקלט :

```
calculate_area(lambda x: 1/(1+x**5), 0, 3, 5)
```

```
xp = [0, 0.5, 1, 3/2, 2, 5/2, 3]
yp = [1, 32/33, 1/2, 32/275, 1/33, 32/3157, 1/244]
print("Integral:", simpson(yp, xp))
```

```
romberg(lambda x: 1/(1+ x**5), 0, 3)
```

```
quadrature(lambda x: 1/(1 + x**5), 0, 3)
```

Trapezoidal integration

```
number of intervals:  6
S =  1.0675413370366504
```

Simpson integration

```
Integral: 1.0749152777561413
```

Romberg integration

```
 Steps  StepSize   Results
     1  3.000000  1.506148
     2  1.500000  0.927619  0.734776
     4  0.750000  1.082750  1.134461  1.161106
     8  0.375000  1.065943  1.060341  1.055400  1.053722
    16  0.187500  1.065859  1.065831  1.066197  1.066368  1.066418
    32  0.093750  1.065874  1.065878  1.065882  1.065877  1.065875  1.065874
    64  0.046875  1.065877  1.065879  1.065879  1.065878  1.065879  1.065879  1.065879
   128  0.023438  1.065878  1.065879  1.065879  1.065879  1.065879  1.065879  1.065879  1.065879
   256  0.011719  1.065878  1.065879  1.065879  1.065879  1.065879  1.065879  1.065879  1.065879  1.065879

The final result is 1.065878542502731 after 257 function evaluations.
Integral:  1.065878542502731
```

[Gaussian-quadrature integration]

```
iter_1: 0.3490909090909091
iter_2: 1.3806110901546584
iter_3: 0.9910081928866128
iter_4: 1.0549337294487455
iter_5: 1.0867603706343547
iter_6: 1.0560354325749726
iter_7: 1.067441218069673
iter_8: 1.0668563385800636
iter_9: 1.0650416501027626
iter_10: 1.066161057852442
iter_11: 1.065877882342355
iter_12: 1.0658261522377184
iter_13: 1.0659077697949935
iter_14: 1.0658720873453191
iter_15: 1.0658766729878693
iter_16: 1.0658807850937375
iter_17: 1.0658776461812698
iter_18: 1.0658786217456224
iter_19: 1.0658786657113715
iter_20: 1.0658784593313908
iter_21: 1.06587856568758
iter_22: 1.0658785451466877
iter_23: 1.0658785367119168
('result: 1.0658785367119168', ', estimate error:8.434770881748932e-09')
```