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
```

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
```

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 numpy as np

def jacobi(A, b, ITERATION_LIMIT = 1000):
    """
    Solves systems of linear equations using Jacobi algo
    Input
          A                 : matrix of linear equations
          b                 : solutions of linear equations
          ITERATION_LIMIT : max iteration till stop
    output
         x : vector that contains the solutions of the equations
    """
    x = np.zeros_like(b)
    for it_count in range(ITERATION_LIMIT):
        x_new = np.zeros_like(x)

        for i in range(A.shape[0]):
            s1 = np.dot(A[i, :i], x[:i])
            s2 = np.dot(A[i, i + 1:], x[i + 1:])
            x_new[i] = (b[i] - s1 - s2) / A[i, i]

        if np.allclose(x, x_new, atol=1e-10, rtol=0.):
            break

        x = x_new
    # error = np.dot(A, x) - b
    return x
```

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
```

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)))
```

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))
```

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
    # I still don't quite understand where these are taken from, so I
over-fit it for the example (being a natural cubic spline)
    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
            # Atrocious, I'm sorry.
            (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)
```

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))
```

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))
```

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)
```

<u>נספחי קוד</u>

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

```python
def cond(A):

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

4) חישוב LU

```python
import pprint
import scipy


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

5) חישוב SOR

```python
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))
```