**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа № 3**

по курсу «Численные методы»

Тема: Приближение функций, численное интегрирование и дифференцирование.

Студент: Сорокин Н.Э.

Группа: М80-303Б-20

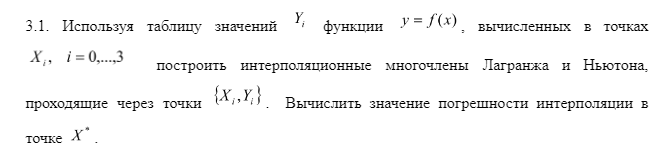
Преподаватель: Иванов И.Э.

Оценка:

Москва, 2023

**№1**

**Постановка задачи:**

****

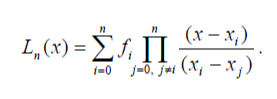
**Вариант 22:**

****

**Теория:**

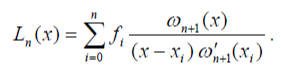
**Многочлен Лагранжа:**

Интерполяционный многочлен Лагранжа имеет вид



Если ввести функцию

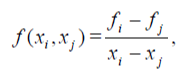
то многочлен примет вид



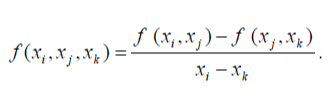
**Многочлен Ньютона:**

Введем понятие разделенной разности. Разделенная разность нулевого порядка совпадает со значением функции в узле.

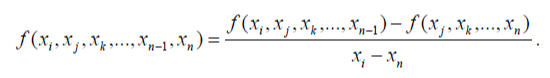
Разделенная разность первого порядка:



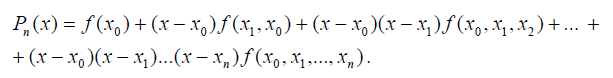
Разделенная разность второго порядка:



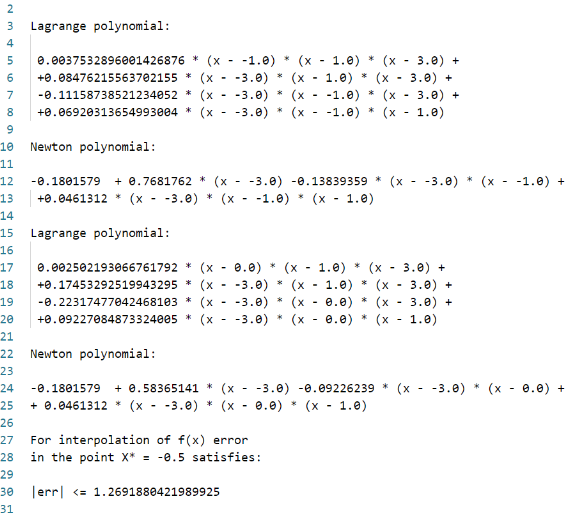
Разделенная разность n – k + 2 порядка:

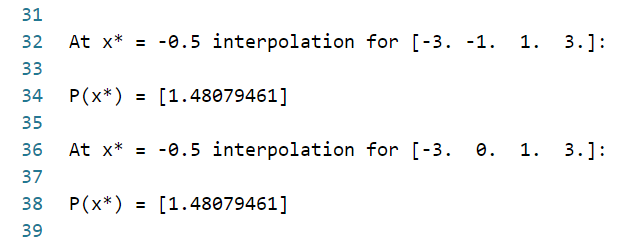


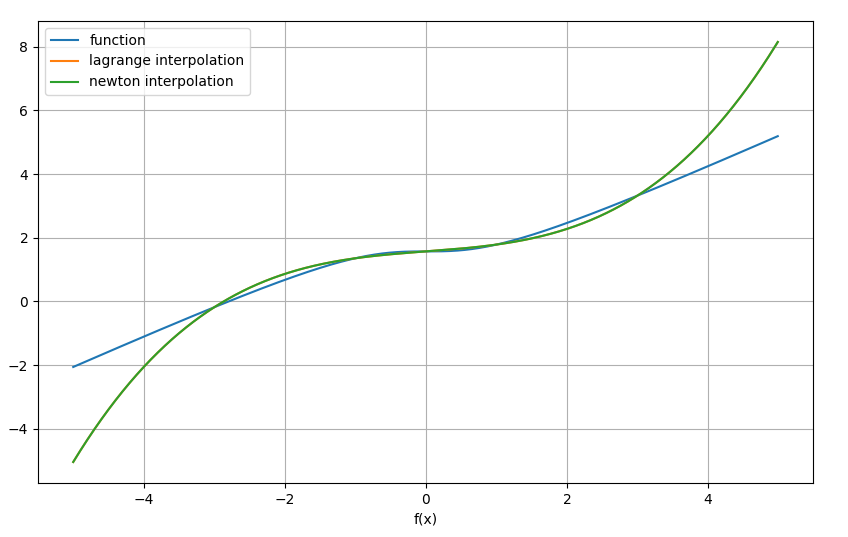
Многочлен Ньютона запишется в виде:

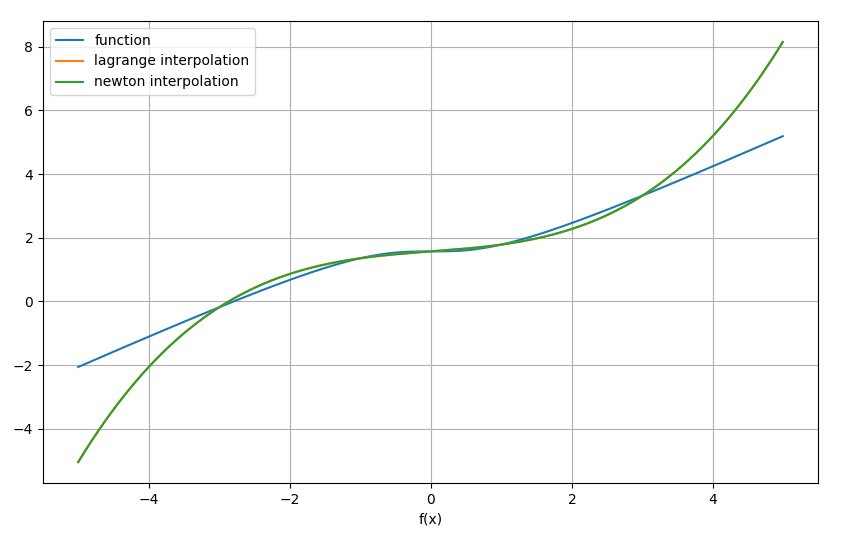


**Полученный ответ:**

****

****

****

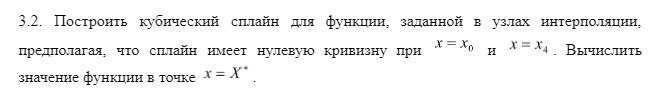
****

**Вывод:** идеи построения интерполяции Лагранжа и Ньютона возникают из самых прямых соображений: воспользоваться многочленами, в узлах интерполяции оставляют одно нужное значение ненулевым. Важно понимать, что многочлен степени n проходящий через n + 1, заданных точек единственен. Этот факт можно рассматривать как следствие некоторых соображений линейной алгебры или китайской теоремы об остатках.

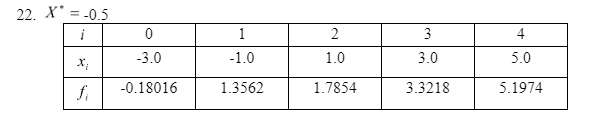
Но несмотря на простоту подхода, такие многочлены лежат в основе численного дифференцирования и интегрирования.

**№2**

**Постановка задачи:**

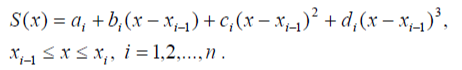
****

**Вариант 22:**

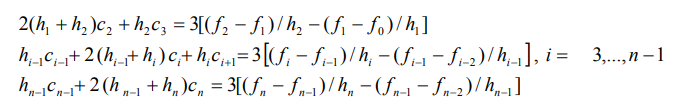
****

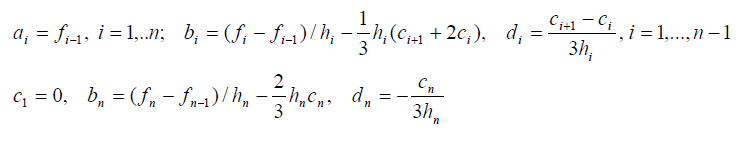
**Теория:**

Для построения кубического сплайна необходимо построить многочлен третьей степени для каждого отрезка разбиения, всего n многочленов. Сам многочлен представляется в виде:

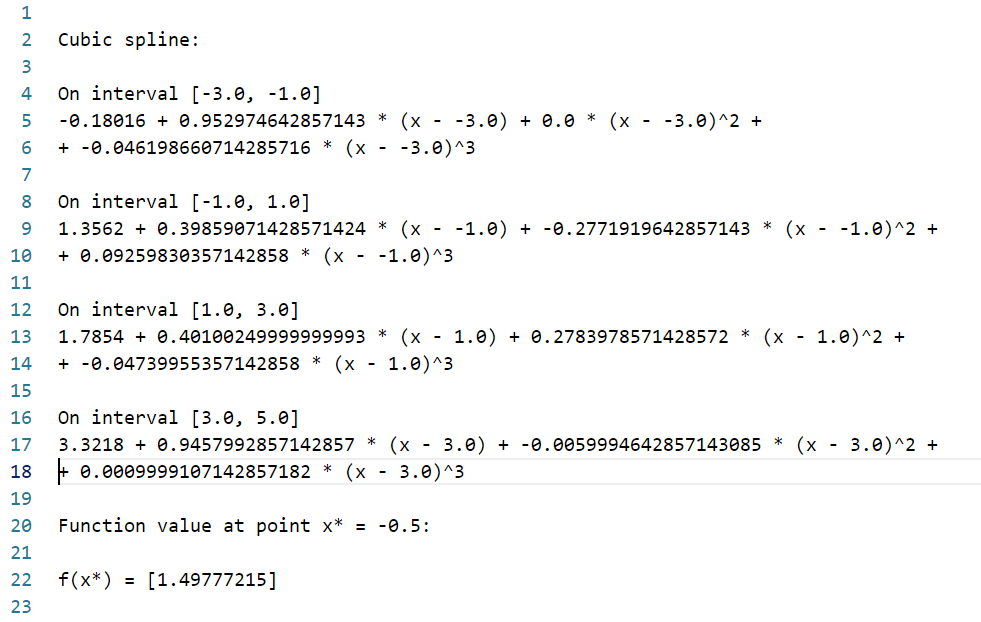


Коэффициенты с\_i находится при помощи решения системы уравнений:

Остальные коэффициенты ищутся по формулам:



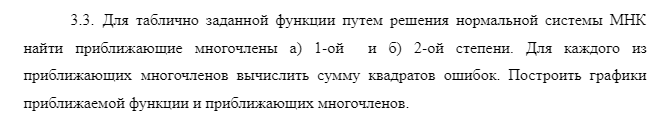
**Полученный ответ:**



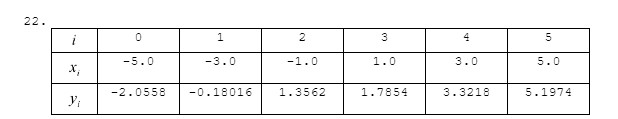
**Вывод:** На большом числе узлов интерполяции многочлены Лагранжа и Ньютона начинают проявлять свои колебательные свойства, в виду того что порядок многочлена будет большим. Поэтому удобно использовать сплайны, например 3 степени. Кубический сплайн представляет из себя непрерывно-дифференцируемую кривую, проходящую через узлы интерполяции, причем на каждом промежутке между узлами сплайн представляет из себя разные многочлены третьей степени, переходящие друг в друга “гладким” образом. (“гладким” в смысле непрерывности первой производной).

Сплайны находят широкое применение, не только в теории интерполяции, но и в компьютерной графике.

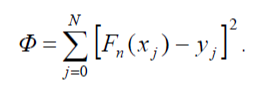
**№3**

**Постановка задачи: **

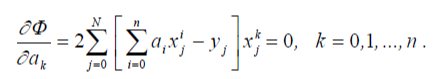
**Вариант 22:**

****

**Теория:** Дана таблично заданная функция, необходимо найти коэффициенты многочлена степени n, который максимально точно аппроксимирует данную функцию. Для этого необходимо добиться минимума функции

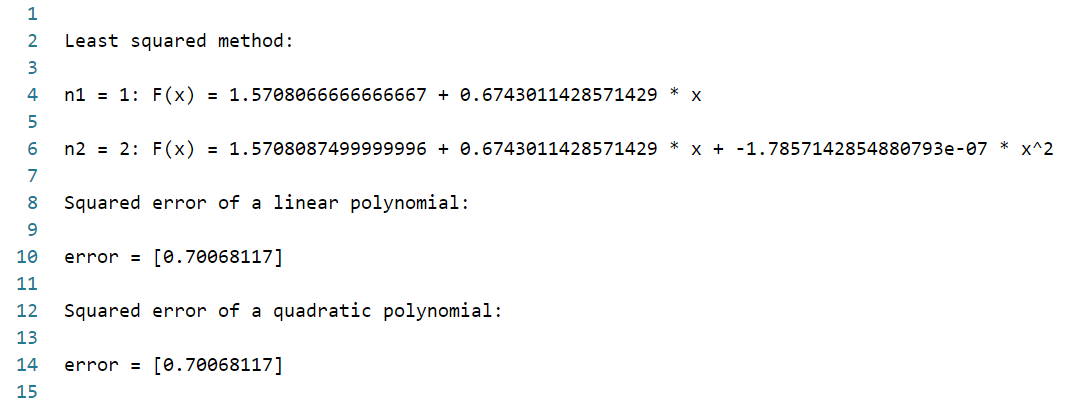


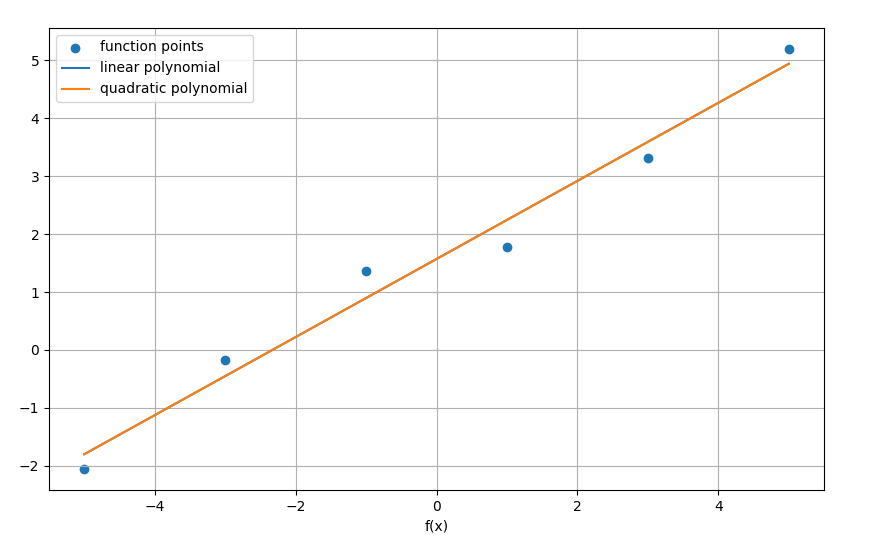
Для этого нужно решить систему из равенств нулю производных:

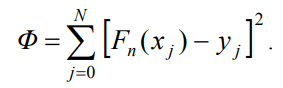
****

Полученное решение будет являться набором коэффициентов для искомого многочлена.

**Полученный ответ:**



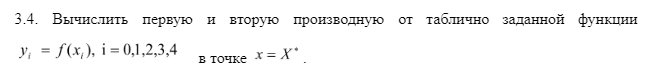


**Вывод:** Метод наименьших квадратов решает задачу поставленную следующим образом: по данному виду функции приближающей исходные данные, например многочлен степени n, найти коэффициенты этого многочлена, такие что минимизируется функция вида: называющаяся квадратичным отклонением.

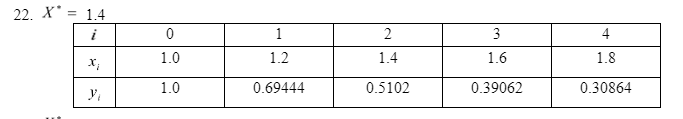
Методом наименьших квадратов решается множество прикладных задач в физике и анализе данных.

**№4**

**Постановка задачи:**

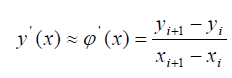
****

**Вариант 22:**

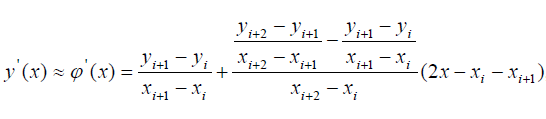
****

**Теория:**

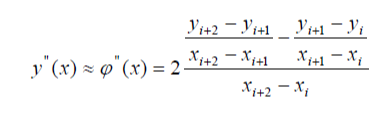
При аппроксимации функции многочленом первой степени первая производная вычисляется по формуле:



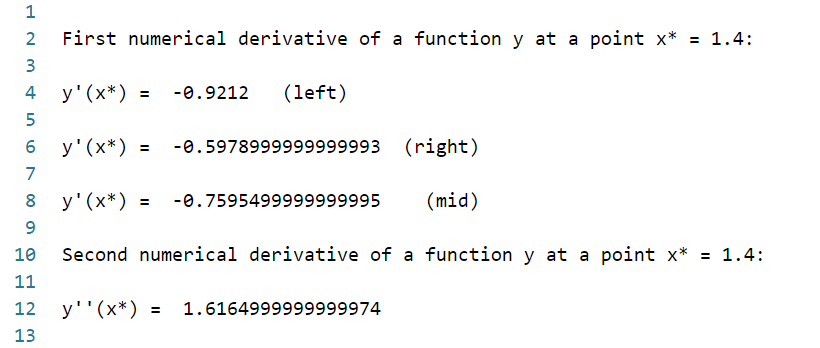
При аппроксимации многочленом второй степени:



Формула для вычисления второй производной:



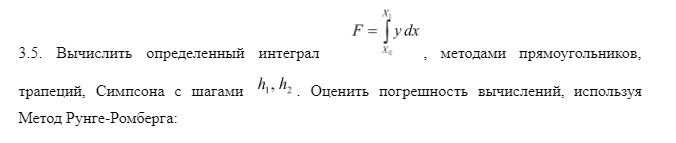
**Полученный ответ:**

****

**Вывод:** Численное дифференцирование полезно когда функция, которую требуется продифференцировать задается таблично. Рассмотренные методы нахождения производных первого и второго порядка применяются в численных методах решения дифференциальных уравнений.

**№5**

**Постановка задачи:**

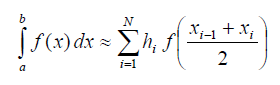
****

**Вариант 22:**

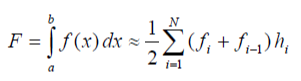
****

**Теория:**

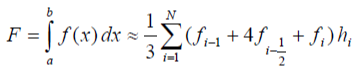
Заменим подынтегральную функцию интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка и получим формулу прямоугольников:



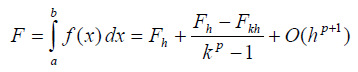
При замене функции многочленом Лагранжа первой степени, проходящим через концы отрезка интегрирования, получим формулу трапеций:



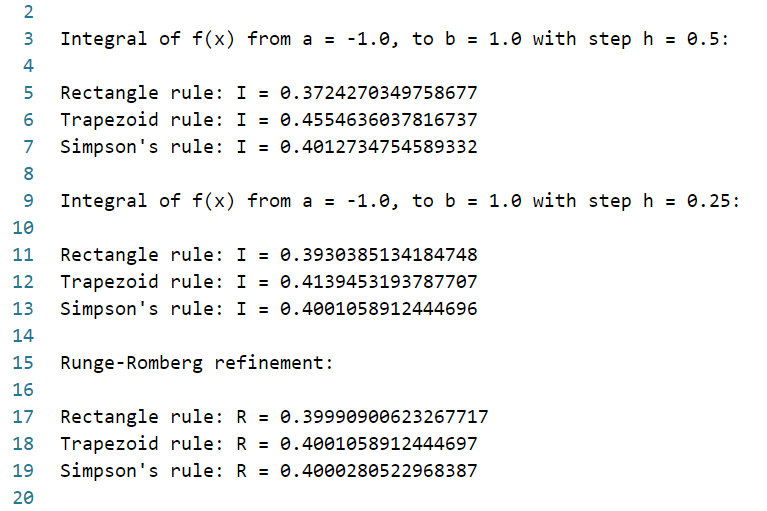
При замене функции многочленом второй степени, проходящим через концы и середину отрезка, получим формулу Симпсона:



Если имеются результаты вычисления определенного интеграла на сетке с шагом h и на сетке с шагом kh, то можно использовать формулу Рунге-Ромберга:



**Полученный ответ:**

****

**Замечание:** Уточненный методом Рунге-Ромберга результат метода трапеций совпадает с неуточненным результатом метода Симпсона с меньшим шагом.

**Вывод:** Численные методы нахождения определенных интегралов являются полезными, когда первообразную найти аналитически не получается. В основе этих методов лежат интерполяционные многочлены.

**Код программы:**

import numpy as np

import math

from LinearAlgebra import \*

#module31

def Lagrange\_interpolation(func, x, x\_nodes):

n = x\_nodes.size

N = x.size

L = np.zeros(N)

L\_str = ""

for i in range(n):

coef\_str = ""

l\_i = 1

denom = 1

for j in range(n):

if (i != j):

l\_i \*= (x - x\_nodes[j]) / (x\_nodes[i] - x\_nodes[j])

denom \*= (x\_nodes[i] - x\_nodes[j])

coef\_str += f" \* (x - {x\_nodes[j]})"

L += func(x\_nodes[i]) \* l\_i

L\_str += f"+ {func(x\_nodes[i]) / denom}" + coef\_str + " "

print(f"Lagrange polynomial: \n\n {L\_str} \n")

return L

def Newton\_interpolation(func, x, x\_nodes):

n = x\_nodes.size

N = x.size

P = np.zeros(N)

P\_str = ""

coef\_str = ""

for i in range(n):

p = 1

for j in range(i):

p \*= (x - x\_nodes[j])

f = divided\_differences(func, x\_nodes[:i + 1])

P += p \* f

P\_str += f" + {f} {coef\_str}"

coef\_str += f" \* (x - {x\_nodes[i]})"

print(f"Newton polynomial: \n\n {P\_str} \n")

return P

def divided\_differences(func, x):

if (x.size == 1):

return func(x)

f1 = divided\_differences(func, x[:-1])

f2 = divided\_differences(func, x[1:])

f = (f2 - f1) / (x[-1] - x[0])

return f

def error\_estimate(M, x\_nodes, x\_0):

n = x\_nodes.size

omega = 1

for i in range(n):

omega \*= abs(x\_0 - x\_nodes[i])

x = np.arange(x\_nodes[0], x\_nodes[-1], 0.1)

M\_max = np.max(abs(M(x)))

n\_fact = math.factorial(n)

eps\_est = M\_max \* omega / n\_fact

return eps\_est

#module32

def cubic\_spline(x, f):

n = x.size - 1

h = np.zeros(n + 1)

for i in range(1, n + 1):

h[i] = x[i] - x[i - 1]

A = np.zeros((n - 1, n - 1))

b = np.zeros(n - 1)

A[0][0] = 2 \* (h[1] + h[2])

A[0][1] = h[2]

b[0] = 3 \* ((f[2] - f[1]) / h[2] - (f[1] - f[0]) / h[1])

A[n - 2][n - 3] = h[n - 1]

A[n - 2][n - 2] = 2 \* (h[n - 1] + h[n])

b[n - 2] = 3 \* ((f[n] - f[n - 1]) / h[n] - (f[n - 1] - f[n - 2]) / h[n - 1])

for i in range(1, n - 2):

A[i][i - 1] = h[i + 1]

A[i][i] = 2 \* (h[i + 2] + h[i + 1])

A[i][i + 1] = h[i + 2]

b[i] = 3 \* ((f[i + 2] - f[i + 1]) / h[i + 2] - (f[i + 1] - f[i]) / h[i + 1])

c = np.zeros(n + 1)

c[2:] = sweep\_method(A, b)

a = np.zeros(n + 1)

b = np.zeros(n + 1)

d = np.zeros(n + 1)

a[n] = f[n - 1]

b[n] = (f[n] - f[n - 1]) / h[n] - 2 / 3 \* h[n] \* c[n]

d[n] = -c[n] / (3 \* h[n])

for i in range(1, n):

a[i] = f[i - 1]

b[i] = (f[i] - f[i - 1]) / h[i] - 1 / 3 \* h[i] \* (c[i + 1] + 2 \* c[i])

d[i] = (c[i + 1] - c[i]) / (3 \* h[i])

k = 100

f\_spline = np.zeros(n \* k)

x\_spline = np.zeros(n \* k)

for i in range(1, n + 1):

x\_spline[(i - 1) \* k: i \* k] = np.linspace(x[i - 1], x[i], k, endpoint=False)

x\_del = x\_spline[(i - 1) \* k: i \* k] - x[i - 1]

f\_spline[(i - 1) \* k: i \* k] = a[i] + b[i] \* x\_del + c[i] \* x\_del\*\*2 + d[i] \* x\_del\*\*3

return x\_spline, f\_spline

#module33

def least\_squares(x, y, n):

N = x.size - 1

A = np.zeros((n + 1, n + 1))

b = np.zeros((n + 1))

for i in range(n + 1):

for k in range(N + 1):

b[i] += y[k] \* x[k]\*\*i

for j in range(n + 1):

for k in range(N + 1):

A[i][j] += x[k]\*\*(i + j)

L, U, b = get\_LU(A, b)

a = solve\_LU(L, U, b)

step = 0.01

x\_ls = np.arange(x[0], x[-1] + step, step)

y\_ls = np.zeros(x\_ls.size)

for i in range(n + 1):

y\_ls += a[i] \* x\_ls\*\*i

return x\_ls, y\_ls

def squared\_error(x\_ls, y\_ls, x, y):

for i in range(x\_ls.size):

x\_ls[i] = round(x\_ls[i], 3)

error = 0

for i in range(x.size):

error += (y[i] - y\_ls[np.where(x\_ls == x[i])])\*\*2

return error

#module34

def numerical\_first\_derivative(x, y, x\_star):

i = 0

for j in range(x.size):

if(x[j] > x\_star):

i = j - 1

break

y\_left = (y[i] - y[i - 1]) / (x[i] - x[i - 1])

y\_der = (y[i + 1] - y[i - 1]) / (x[i + 1] - x[i - 1])

y\_right = (y[i + 1] - y[i]) / (x[i + 1] - x[i])

return y\_left, y\_der, y\_right

def numerical\_second\_derivative(x, y, x\_star):

i = 0

for j in range(x.size):

if(x[j] > x\_star):

i = j - 1

break

y\_der = (y[i + 1] - 2 \* y[i] + y[i - 1]) / ((x[i + 1] - x[i - 1]) / 2)\*\*2

return y\_der

#module35

def rectangle\_rule(func, a, b, h):

x\_nodes = np.arange(a, b + h, h)

I = 0

for i in range(x\_nodes.size - 1):

I += h \* func((x\_nodes[i] + x\_nodes[i + 1]) / 2)

return I

def trapezoid\_rule(func, a, b, h):

x\_nodes = np.arange(a, b + h, h)

I = 0

for i in range(x\_nodes.size - 1):

I += h \* (func(x\_nodes[i]) + func(x\_nodes[i + 1])) / 2

return I

def Simpsons\_rule(func, a, b, h):

x\_nodes = np.arange(a, b + h, h)

I = 0

for i in range(x\_nodes.size - 1):

x\_mean = (x\_nodes[i] + x\_nodes[i + 1]) / 2

I += h \* (func(x\_nodes[i]) + 4 \* func(x\_mean) + func(x\_nodes[i + 1])) / 6

return I

def Runge\_Rombegr\_method(I\_h, I\_kh, k, p):

R = - (I\_h - I\_kh) / ((1 / k)\*\*p - 1)

return R