Лабораторная работа №3

Содержание

Лабораторная работа 3.1

Полиномиальная интерполяция

Задача: Используя таблицу значений *Yi* функции *y = f (x)*, вычисленных в точках *Xi* , *i* = 0,...,3 построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки {*X i* ,*Yi* }. Вычислить значение погрешности интерполяции в точке *X* * .

Условие:

$$y = e^x + x$$
$$а) X_i = -2, -1, 0, 1$$
$$б) X_i = -2, -1, 0, 2, 2$$
$$X^* = -0.5$$

Решение: Интерполяционный многочлен Лагранжа записывается в виде

$$L_n(x) = \sum_{i=0}^{n} f_i \prod_{j=0, j\neq i}^{n} \frac{(x - x_i)}{(x_i - x_j)}.$$

Если ввести функцию $\omega_{n+1}(x) = (x - x_0)(x - x_1)\ldots(x - x_n) = \prod_{i=0}^{n}(x - x_i)$, то выражение для

интерполяционного многочлена Лагранжа примет вид:

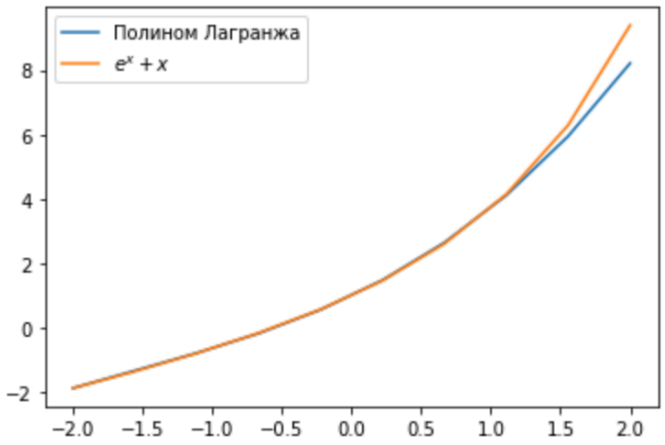
$$L_n(x) = \sum_{i=0}^{n} f_i \frac{\omega_{n+1}(x)}{(x - x_i)\omega'_{n+1}(x_i)}.$$

Недостатки интерполяцинного многочлена Лагранжа является необходимость полного пересчета всех коэффициентов в случае добавления дополнительных интерполяционных узлов. Чтобы избежать указанного недостатка используют интерполяционный многочлен в форме Ньютона:

$$P_n(x) = f(x_0) + (x - x_0)f(x_1, x_0) + (x - x_0)(x - x_1)f(x_0, x_1, x_2) + \ldots + (x - x_0)(x - x_1\ldots(x - x_n))f(x_0, x_1, x_2, \ldots, x_n)$$

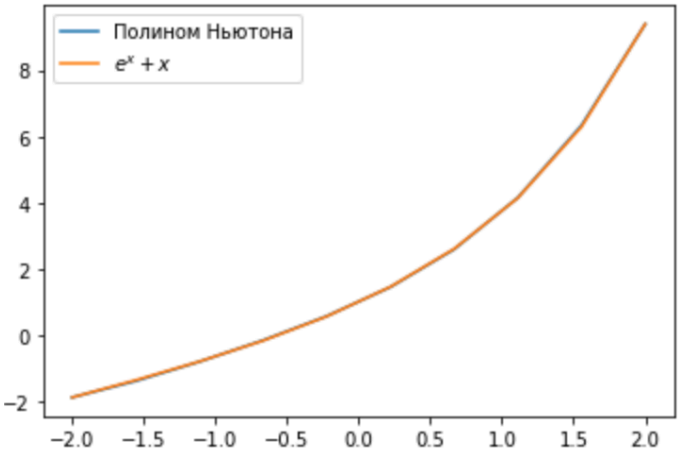
Работа программы:

Получим ошибки в точке X* для метода Лагранжа:
```
L3(-0.5) = 0.09108111617795775
y(-0.5) = 0.10653065971263342
Ошибка = 0.015449543534675675
L4(-0.5) = 0.11873142016964466
y(-0.5) = 0.10653065971263342
Ошибка = 0.01220076045701124
```



Получим ошибки в точке X* для метода Ньютона:

```
P3(-0.5) = 0.0910811161779577
y(-0.5) = 0.10653065971263342
Interpolation error = 0.01544954353467573
P4(-0.5) = 0.11873142016964475
y(-0.5) = 0.10653065971263342
Interpolation error = 0.012200760457011323
```



Листинг программы:

```python
import numpy as np
import matplotlib.pyplot as plt


def function(x):
    return np.exp(x) + x

# вычисление произведения элементов в методе Лагранжа
def multiplyLagrange(x0, x, n, index):

    multiply = 1
```

```
    for i in range(0, n):
        if index != i:
            multiply = multiply * (x0 - x[i])/
(x[index] - x[i])

    return multiply

# вычисление произведения элементов в методе Ньютона
def multiplyNewton(x0, x, n):
    multiply = 1

    for i in range(0, n):
            multiply = multiply * (x0 - x[i])

    return multiply

def polynomOfLagrange(x0, x, n):
    summary = 0
    for i in range(0, n):
        summary = summary + function(x[i]) *
multiplyLagrange(x0, x, n, i)

    return summary

def divideddifference(x, n):
    if n > 2:
        x1 = np.zeros(n-1)
        x2 = np.zeros(n-1)
        for i in range(0, n):
            if i != n - 1:
                x1[i] = x[i]
            if i != 0:
                x2[i - 1] = x[i]
        return (divideddifference(x1, n - 1) -
divideddifference(x2, n - 1)) / (x[0] - x[n - 1])

    if n == 2:
        return (function(x[0]) - function(x[1])) /
(x[0] - x[1])
    if n == 1:
```

```
        return function(x[0])

def polynomOfNewton(x0, x, n):

    summary = 0
    for i in range(0, n):
        dd = divideddifference(x, i+1)
        summary = summary + dd * multiplyNewton(x0,
x, i)

    return summary

x0 = -0.5
x = [[-2, -1, 0, 1],[-2, -1, 0, 1, 2]]

print("Получим ошибки в точке X* для метода
Лагранжа:")
for x_ in x:
  lagr = polynomOfLagrange(x0, x_, len(x_))

  print("L" + str(len(x_) - 1) + "(" + str(x0) + ") =
" + str(lagr))

  print("y(" + str(x0) + ") = " + str(function(x0)))
  print("Ошибка = " + str(abs(function(x0) - lagr)))

test = np.linspace(-2, 2, 10)
y_lagr = np.zeros(len(test))
for i in range(0, len(test)):
    y_lagr[i] = polynomOfLagrange(test[i], xa, n)

fig, ax = plt.subplots()
ax.plot(test, y_lagr)
ax.plot(test, function(test))
ax.legend(["Полином Лагранжа","$e^x+x$"])
plt.xlim()
plt.show()

print("Получим ошибки в точке X* для метода
Ньютона:")
```

```
for x_ in x:
  newton = polynomOfNewton(x0, x_, len(x_))

  y_newt = np.zeros(len(test))
  for i in range(0, len(test)):
      y_newt[i] = polynomofnewton(test[i], x_,
len(x_))

  print("P" + str(len(x_) - 1) + "(" + str(x0) + ") =
" + str(newton))
  print("y(" + str(x0) + ") = " + str(function(x0)))

  print("Interpolation error = " +
str(abs(function(x0) - newton)))

fig, ax = plt.subplots()
ax.plot(test, y_newt)
ax.plot(test, function(test))
ax.legend(["Полином Ньютона", "$e^x+x$"])
plt.show()
```

Лабораторная работа 3.2

Сплайн-интерполяция

Задача: Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$.

Условие:

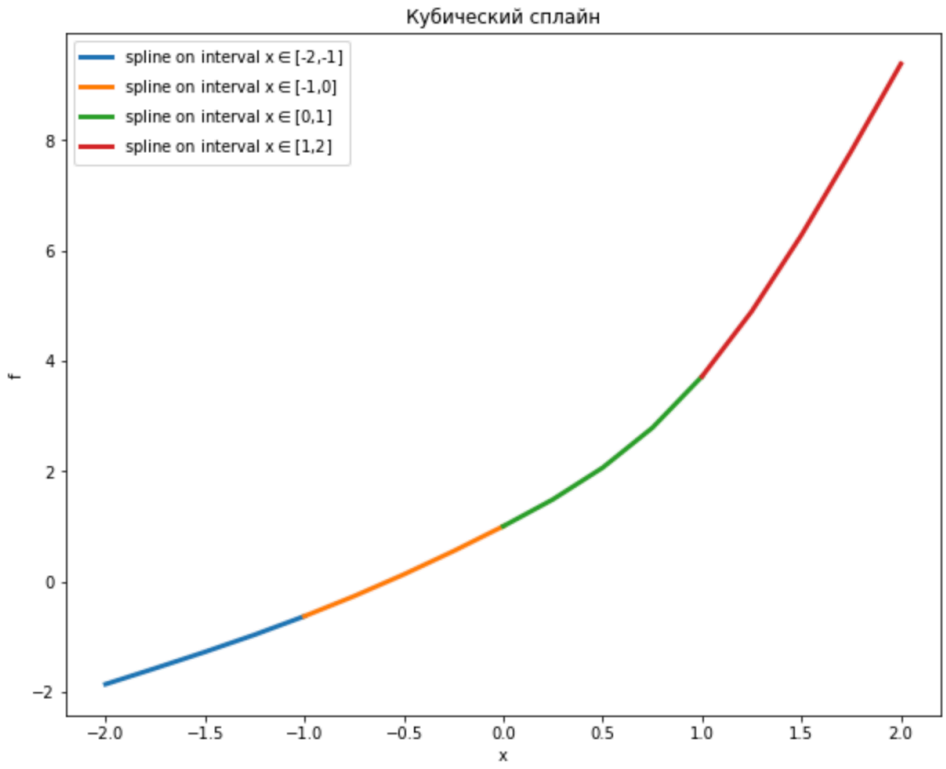
$$x = [-2, -1, 0, 1, 2]$$
$$f = [-1.8647, -0.63212, 1.0, 3.7183, 9.3891]$$

Решение: Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается в применении сплайн-интерполяции. Суть сплайн-интерполяции заключается в определении интерполирующей функции по формулам одного типа для различных непересекающихся промежутков и в стыковке значений функции и её производных на их границах.

Наиболее широко применяемым является случай, когда между любыми двумя точками разбиения исходного отрезка строится многочлен n-й степени, который в узлах интерполяции принимает значения аппрокимируемой функции и непрерывен вместе со своими (n – 1) производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Его коэффициенты находятся из условий равенства в узлах сетки значений сплайна и приближаемой функции, а также равенства n –1 производных соответствующих многочленов.

Работа программы:

Значение функции в x0 = 0.12654089285714284



Листинг программы:

```
import numpy as np
import matplotlib.pyplot as plt

def spline(a, b, c, d, x0, x):
    return a + b * (x0 - x) + c * (x0 - x)**2 + d *
(x0 - x)**3

def runMethod(c, f, size):
    cn = np.zeros(size)
    fn = np.zeros(size)
    yn = np.zeros(size)
    xn = np.zeros(size)

    for i in range(0, size):
        if i == 0:
            yn[i] = c[i][i]
            cn[i] = -c[i][i + 1] / yn[i]
            fn[i] = f[i] / yn[i]
        elif i == size - 1:
            yn[i] = c[i][i] + c[i][i-1] * cn[i-1]
            fn[i] = (f[i] - c[i][i-1] * fn[i-1]) /
yn[i]
        else:
            yn[i] = c[i][i] + c[i][i-1] * cn[i-1]
            cn[i] = -c[i][i+1] / yn[i]
            fn[i] = (f[i] - c[i][i-1] * fn[i-1]) /
yn[i]

    xn[size-1] = fn[size-1]
    i = size-2
    while i >= 0:
        xn[i] = cn[i] * xn[i+1] + fn[i]
        i = i - 1

    return xn

# определение трехдиагональной матрицы для метода
прогонки
```

```
def getCi(x, f, size):

    c = np.zeros((size-2, size-2))
    fpr = np.zeros(size-2)

    for i in range(0, size-2):
        if i == 0:
            c[i][i] = 2*(x[i+2] - x[i])
            c[i][i+1] = x[i+2] - x[i+1]

        elif i == size-3:
            c[i][i - 1] = x[i + 1] - x[i]
            c[i][i] = 2 * (x[i + 2] - x[i])

        else:
            c[i][i - 1] = x[i + 1] - x[i]
            c[i][i] = 2 * (x[i + 2] - x[i])
            c[i][i + 1] = x[i + 2] - x[i + 1]
        fpr[i] = 3 * ((f[i + 2] - f[i + 1]) / (x[i +
1] - x[i])-(f[i + 1] - f[i]) / (x[i + 2] - x[i + 1]))

    return runMethod(c, fpr, size-2)

def getCoeffs(f,x,a,b,c,d):
  tmp = np.zeros(n-2)
  tmp = getCi(x, f, n)
  c[0] = 0
  for i in range(1, n-1):
      c[i] = tmp[i-1]
      for i in range(0, n-1):
        a[i] = f[i]
        hi = x[i+1]-x[i]
        if i != n-2:
            b[i] = (f[i+1]-f[i]) / hi - hi * (c[i+1]
+ 2 * c[i]) / 3
            d[i] = (c[i+1] - c[i]) / (3 * hi)

  hn = x[n - 1] - x[n - 2]
  b[n - 2] = (f[n - 1] - f[n - 2]) / hn - 2 * hn *
c[n - 2] / 3;
```

```
  d[n - 2] = -c[n - 2] / hn / 3

"""Вариант 17:
$$
x = [-2, -1, 0, 1, 2]\\
f = [-1.8647, -0.63212, 1.0, 3.7183, 9.3891]
$$
"""


n = 5
x0 = -0.5
x = [-2, -1, 0, 1, 2]
f = [-1.8647, -0.63212, 1.0, 3.7183, 9.3891]

a = np.zeros(n-1)
b = np.zeros(n-1)
c = np.zeros(n-1)
d = np.zeros(n-1)

getCoeffs(f,x,a,b,c,d)

for i in range(0, n-1):
    # высчитываем значение сплайна в заданной точке
    val = spline(a[i], b[i], c[i], d[i], x0, x[i])
    if (x0 <= x[i + 1]) and (x0 >= x[i]):
        print("Значение функции в x0 = " + str(val))

"""Построим графики сплайнов на интервалах"""

val = np.zeros((n-1, 5))
x_test = np.zeros((n-1, 5))
legends = []
plt.figure(figsize=(10,8))
for i in range(0, n - 1):
    x_test[i] = np.linspace(x[i], x[i + 1], 5)
    legends.append("spline on interval x$\in$[" +
str(x[i]) + "," + str(x[i + 1]) + "]")
    counter = 0
    for x_local in x_test[i]:
```

```
        val[i][counter] = spline(a[i], b[i], c[i],
d[i], x_local, x[i])
        counter = counter + 1
    plt.plot(x_test[i], val[i], linewidth="3")

plt.title("Кубический сплайн");
plt.xlabel('x')
plt.ylabel('f')
plt.legend(legends)
plt.show()
```

Лабораторная работа 3.3

Метод наименьших квадратов

Задача: Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить

значение функции в точке $x = X^*$.

Условия:

i	0	1	2	3	4	5
x_i	0.1	0.5	0.9	1.3	1.7	2.1
y_i	-2.2026	-0.19315	0.79464	1.5624	2.2306	2.8419

Решение: Пусть задана таблично в узлах функция, при этом значения функции определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки: многочлен степени n, неизвестные коэффициенты которого находится из условий минимума квадратичного отклонения многочлена от таблично заданной функции

Систему

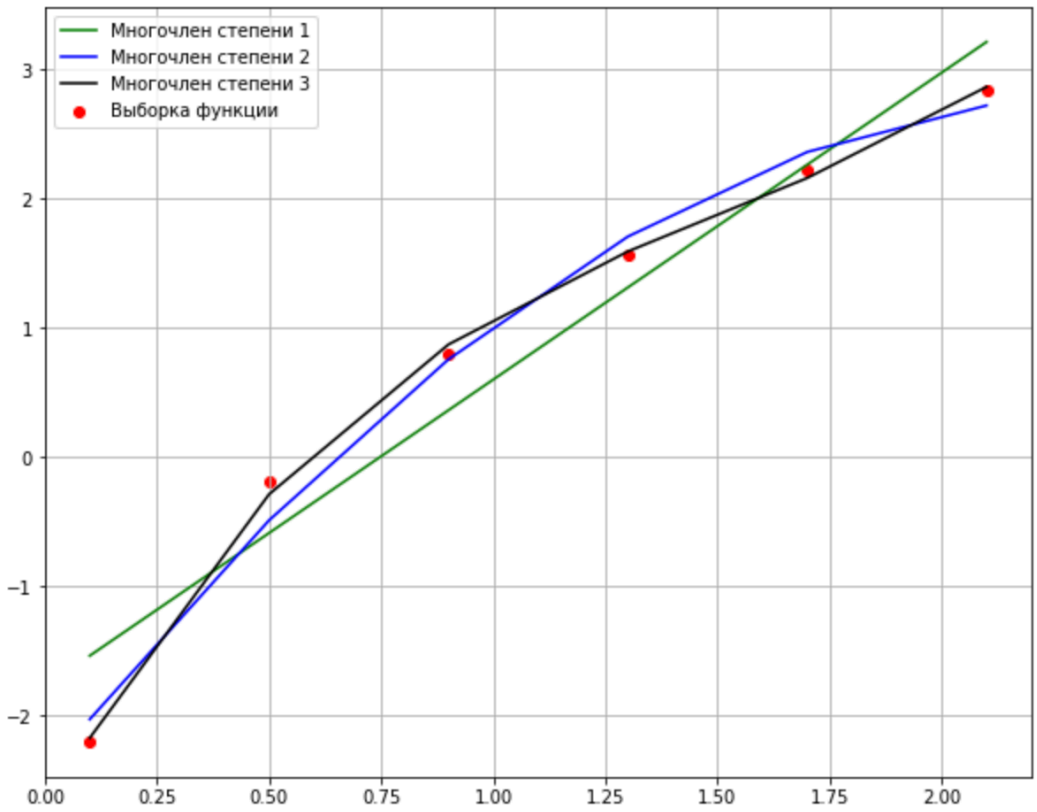
$$\sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{k+i} = \sum_{j=0}^{N} y_j x_j^k,\ k = 0,1,...,n\ .$$

Называют нормальной системой метода наименьших квадратов (МНК). Решив эту систему, можно найти коэффициента многочлена F_n, приближающего значения функции.

$$\Phi = \sum^{N} [F_n(x)j) - y_j]^2 .$$

Работа программы:

```
Degree = 1
F1(x) = -1.77444x^0 + 2.37582x^1
Degree = 2
F2(x) = -2.46046x^0 + 4.40618x^1 + -0.92289x^2
```




```
Degree = 3
F3(x) = -2.79218x^0 + 6.52318x^1 + -3.39847x^2 +
0.75018x^3
```

Error of F1 = 0.98541
Error of F2 = 0.17139
Error of F3 = 0.02202

Листинг программы:

```
import numpy as np
import matplotlib.pyplot as plt

def error(F, y):
    return sum([el**2 for el in F-y])

x = [0, 1.7, 3.4, 5.1, 6.8, 8.5]
y = np.array([-2, 3.0038, 5.2439, 7.3583, 9.4077,
11.415])
#y = np.array([0, 1.3038, 1.8439, 2.2583, 2.6077,
2.9155])
```

```
A = np.array([[len(x), sum(x)],\
              [sum(x), sum([el**2 for el in x])]])
b = np.array([[sum(y)],
              [sum(np.multiply(x, y))]])
a = np.linalg.solve(A, b)
F1 = np.zeros(len(x))
for i in range(len(x)):
    F1[i] = (a[0]+a[1]*x[i])
print('Функция, вычисленная многочленом первой
степени: \n', F1)
print('Сумма квадратов ошибки: \n', error(F1, y))
plt.plot(x, y)
plt.plot(x, F1)

A = np.array([[len(x), sum(x), sum([el**2 for el in
x])],\
              [sum(x), sum([el**2 for el in x]),
sum([el**3 for el in x])],\
              [sum([el**2 for el in x]), sum([el**3
for el in x]), sum([el**4 for el in x])]])
b = np.array([[sum(y)],\
              [sum(np.multiply(x, y))],\
              [sum(np.multiply([el**2 for el in x],
y))]])
a = np.linalg.solve(A, b)
F2 = np.zeros(len(x))
for i in range(len(x)):
    F2[i] = (a[0]+a[1]*x[i]+ a[2]*x[i]**2)
x2 = np.arange(0, 8.6, 0.1)
F22 = np.zeros(len(x2))
for i in range(len(x2)):
    F22[i] = (a[0]+a[1]*x2[i]+ a[2]*x2[i]**2)
print('Функция, вычисленная многочленом второй
степени: \n', F2)
print('Сумма квадратов ошибки: \n', error(F2, y))
plt.plot(x2, F22)
plt.grid()
plt.legend(["Оригинал","Первая степень", "Вторая
степень"])
plt.show()
```

Лабораторная работа 3.4

Численное дифференцирование

Задача: Вычислить первую и вторую производную от таблично заданной функции $yi = f(xi), i$=0,1,2,3,4 вточке x=$X*$.
Условия:

$$X^* = 0.2$$

i	0	1	2	3	4
x_i	-0.2	0.0	0.2	0.4	0.6
y_i	-0.40136	0.0	0.40136	0.81152	1.2435

Решение:

При решении практических задач, как правило, используются аппроксимации первых и вторых производных.

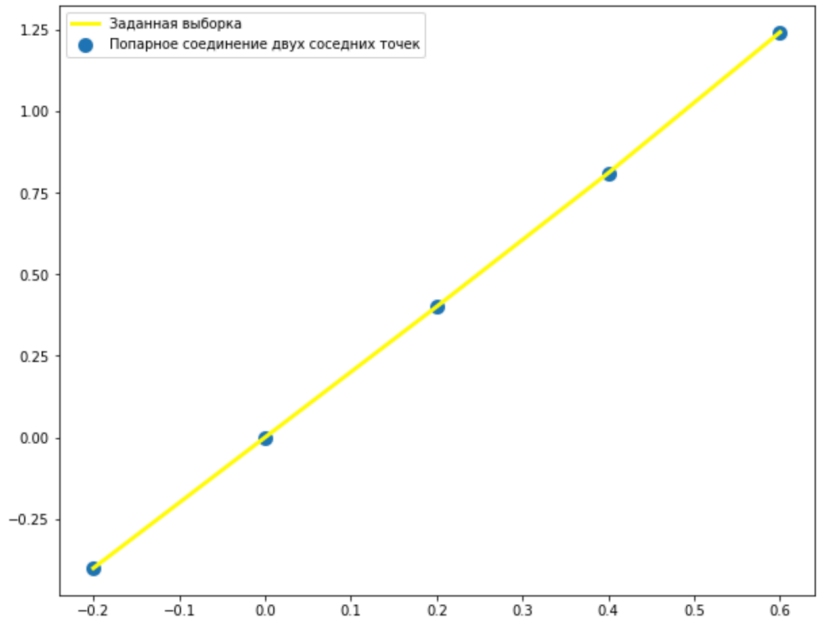
В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой $y(x) \approx \varphi(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i),\ x \in [x_i, x_{i+1}]$. В этом случае:

$$y'(x) \approx \varphi'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = const,\ x \in [x_i, x_{i+1}], \qquad (3.18)$$

производная является кусочно-постоянной функцией и рассчитывается, по формуле (3.18) с первым порядком точности в крайних точках интервала, и со вторым порядком точности в средней точке интервала [1].

Работа программы:

```
Первая производная в точке = 2.0288
Вторая производная в точке = 0.22
```



Зависимость почти линейная, поэтому тангенс угла наклона касательной в 0.2 равен примерно двум, что совпадает с результатом. И так как функция почти линейна, то вторая производная примерно равна нулю, что видно из второго результата работы программы

Листинг программы:

```
def RangeError():
    print('производная в этой точке неопределена')

def a(i, x_v, y_v):
    return (y_v[i + 1] - y_v[i]) / (x_v[i + 1] -
x_v[i])


def b(i, x_v, y_v):
    return (a(i + 1, x_v, y_v) - a(i, x_v, y_v)) /
(x_v[i + 2] - x_v[i])


def derivative1(x, x_v, y_v):
    intervals = [[x_v[i], x_v[i + 1]] for i in
range(0, len(x_v) - 1)]
    index = None
    for i in range(0, len(intervals)):
        if intervals[i][0] <= x <= intervals[i][1]:
            index = i
```

```
            break
    if index is None:
        RangeError()
    if index == 0 or index == len(x_v) - 2:
        return a(index, x_v, y_v)
    else:
        return a(index, x_v, y_v) + b(index, x_v,
y_v) * (2 * x - x_v[index] - x_v[index + 1])


def derivative2(x, x_v, y_v):
    intervals = [[x_v[i], x_v[i + 1]] for i in
range(0, len(x_v) - 1)]
    index = None
    for i in range(0, len(intervals)):
        if intervals[i][0] <= x <= intervals[i][1]:
            index = i
            break
    if index is None or index == len(x_v) - 2:
        RangeError()
    else:
        return b(index, x_v, y_v) * 2.


x = 0.2
x_v = [-0.2, 0.0, 0.2, 0.4, 0.6]
y_v = [1.5722, 1.5708, 1.5694, 1.5593, 1.5273]
print(derivative1(x, x_v, y_v))
print(derivative2(x, x_v, y_v))
```

Лабораторная работа 3.5

Численное интегрирование

Задача: Вычислить определенный интеграл $F = \int_{X_0} y dx$, методами прямоугольников, трапеций, Симпсона с шагами h1 , h2 . Оценить погрешность вычислений, используя Метод Рунге-Ромберга:

Условия:

$$y = \frac{1}{254 - x^4}$$

$$X_0 = 0, X_k = 2, h_1 = 1.0, h_2 = 0.5$$

Вычисленное значение определенного интеграла:

$$\frac{\ln(3) + 2\arctan\left(\frac{1}{2}\right)}{256}$$

Решение:

Формула прямоугольников:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} h_i\, f\left(\frac{x_{i-1} + x_i}{2}\right)$$

Формула трапеций:

$$F = \int_a^b f(x)\,dx \approx \frac{1}{2}\sum_{i=1}^{N} (f_i + f_{i-1})\, h_i$$

Формула Симпсона:

$$F = \int_a^b f(x)\,dx \approx \frac{1}{3}\sum_{i=1}^{N} (f_{i-1} + 4f_{i-\frac{1}{2}} + f_i)\, h_i$$

Работа программы:

```
Метод rectangle:
Шаг h =1.0
Значение интеграла = 0.007892259947054468
Шаг h =0.5
Значение интеграла = 0.007908029175480238

Ошибка Рунге-Ромберга 5.256409475256722e-06
Ошибка абсолютная = 5.672022448362582e-06



Метод trapeze:
Шаг h =1.0
Значение интеграла = 0.004036458333333333
Шаг h =0.5
Значение интеграла = 0.00593261543399411

Ошибка Рунге-Ромберга 0.0006320523668869258
Ошибка абсолютная = 0.001981085763934491


Метод simpson:
Шаг h =1.0
Значение интеграла = 0.007919730392156862
Шаг h =0.5
Значение интеграла = 0.007914182284964415

Ошибка Рунге-Ромберга 1.8493690641489813e-06
Ошибка абсолютная = 4.81087035814412e-07
```

Листинг программы:

```python
import numpy as np

def f(x):
    return 1/(16 - x**4)
    #return x/(3*x+4)**2

x0 = -1
xk = 1
h1 = 0.5
```

```
h2 = 0.25

def rect(x, h):
    r = 0
    for i in range(1, len(x)):
        r += f((x[i] + x[i-1])/2)
    return h*r

def trap(x, h):
    r = y[0]/2
    for i in range(1, len(x-1)):
        r += y[i]
    return h*(r+y[-1]/2)

def Simpson(x, h):
    r = y[0]
    for i in range(1, len(x)):
        if i%2!=0:
            r += 4*y[i]
        else:
            r += 2*y[i]
    return (h/3)*(y[len(x)]+r)

def Runge(Fh, Fkh, p, h1, h2):
    return ((Fh - Fkh) / ((h2/h1)**p - 1))

x = np.arange(x0, xk+h1, h1)
y = [f(xi) for xi in x]
for i in range(1, len(x)):
    print('Прямоугольгник', rect(x[0:i+1], h1))
    print('Трапеция', trap(x[0:i+1], h1))
    if i%2 == 0:
        s21 = Simpson(x[0:i], h2)
        print('Симпсон', s21)
    print()
tr2 = trap(x[0:i+1], h1)

x = np.arange(x0, xk+h2, h2)
y = [f(xi) for xi in x]
for i in range(1, len(x)):
```

```
    print('Прямоугольгник', rect(x[0:i+1], h2))
    print('Трапеция', trap(x[0:i+1], h2))
    if i%2 == 0:
        s2 = Simpson(x[0:i], h2)
        print('Симпсон', s2)
    print()
tr = trap(x[0:i+1], h1)

z = 0.0476338
r = Runge(rect(x, h1), rect(x, h2), 2, h1, h2)
print('Рунге прямоугольник', r, abs(z-rect(x, h2)))
r = Runge(trap(x, h1), trap(x, h2), 2, h1, h2)
print('Рунге трапеция', r, abs(z-trap(x, h2)))
r = Runge(s21, s2, 4, h1, h2)
print('Рунге Симпсон', r, abs(z-s2))
```