**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Компьютерные науки и прикладная математика»  
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа № 3   
по курсу «Численные методы»

Группа: М8О-309Б-22

Студент: Концебалов О. С.

Преподаватель: Шабунина А. А.

Дата: 05.06.2025

Москва, 2025

**ОГЛАВЛЕНИЕ**

[**1**](#_uksw95ug6pl6) **Тема** 3

[**2**](#_whjd5ofr2nre) **Задание** 3

[**3**](#_wi3nzgibx3io) **Теория** 5

[**4**](#_rlqpxczeozgt) **Демонстрация работы программы** 11

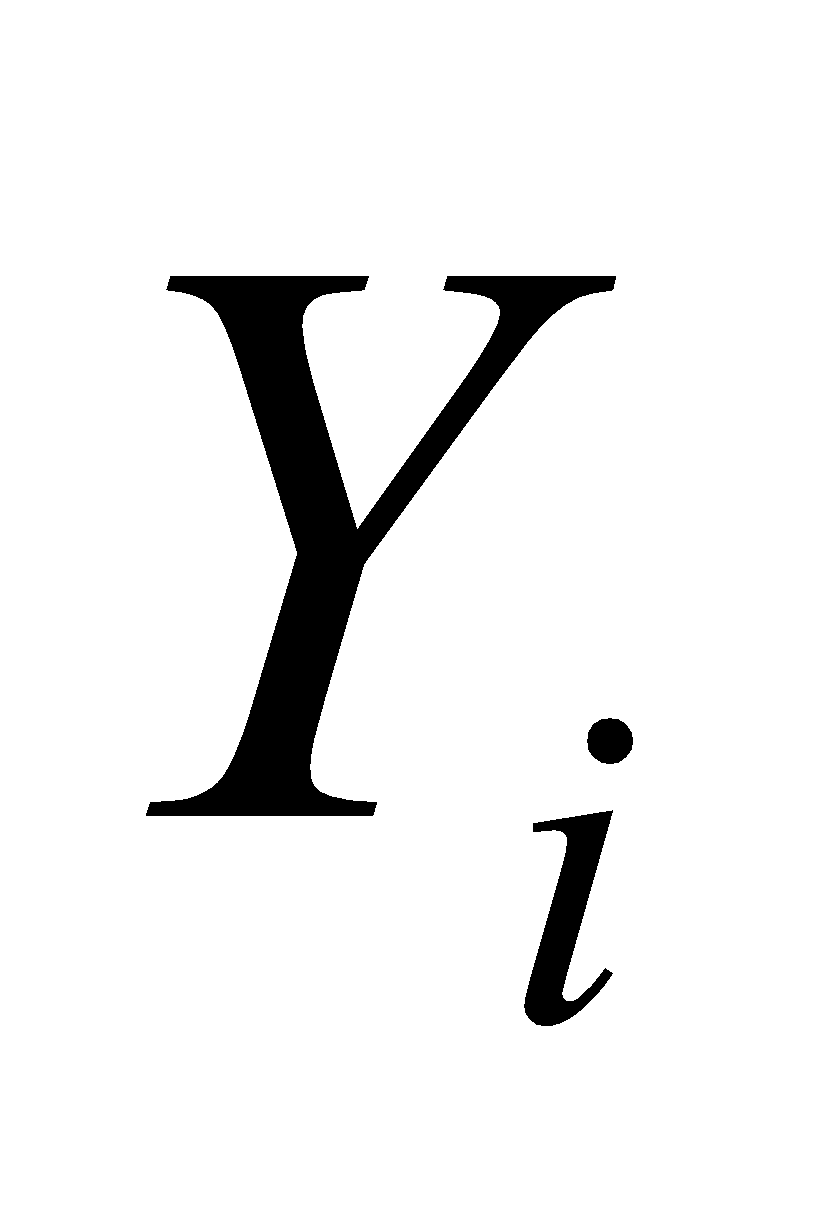
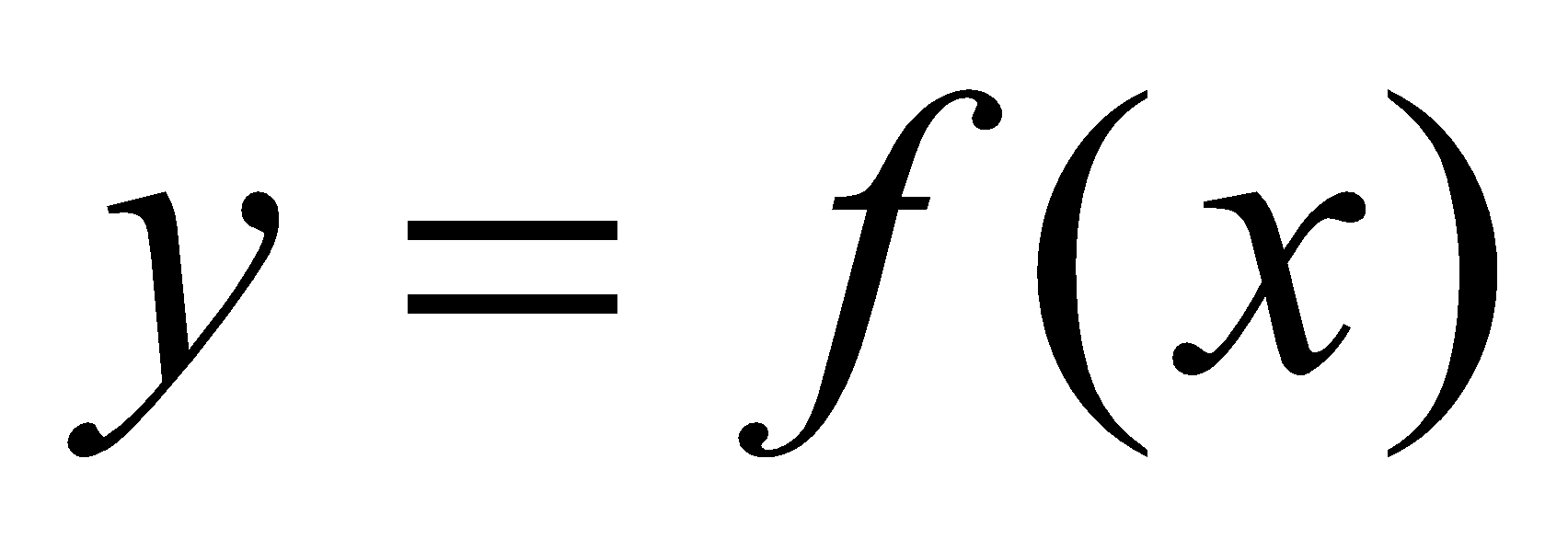
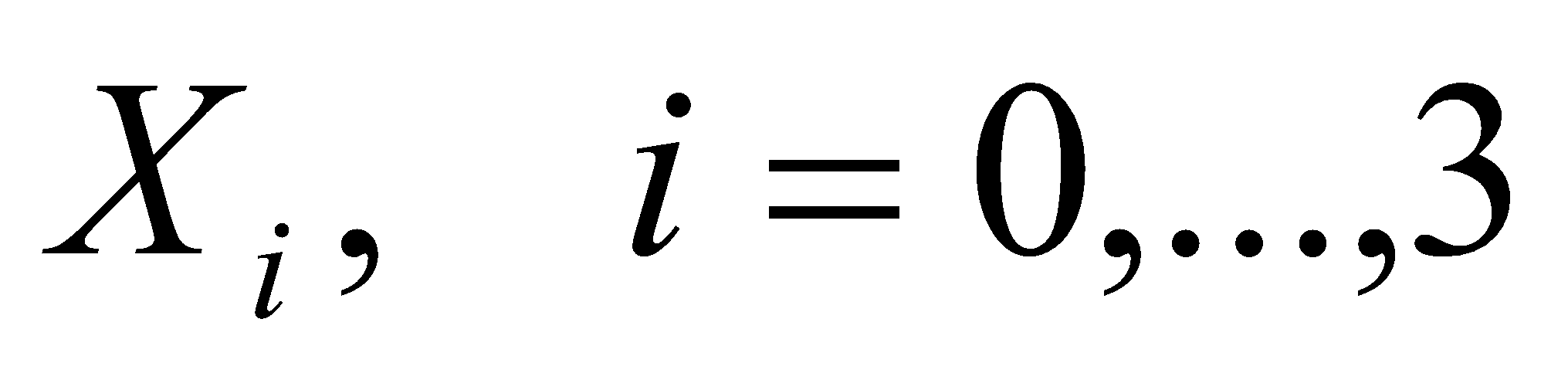
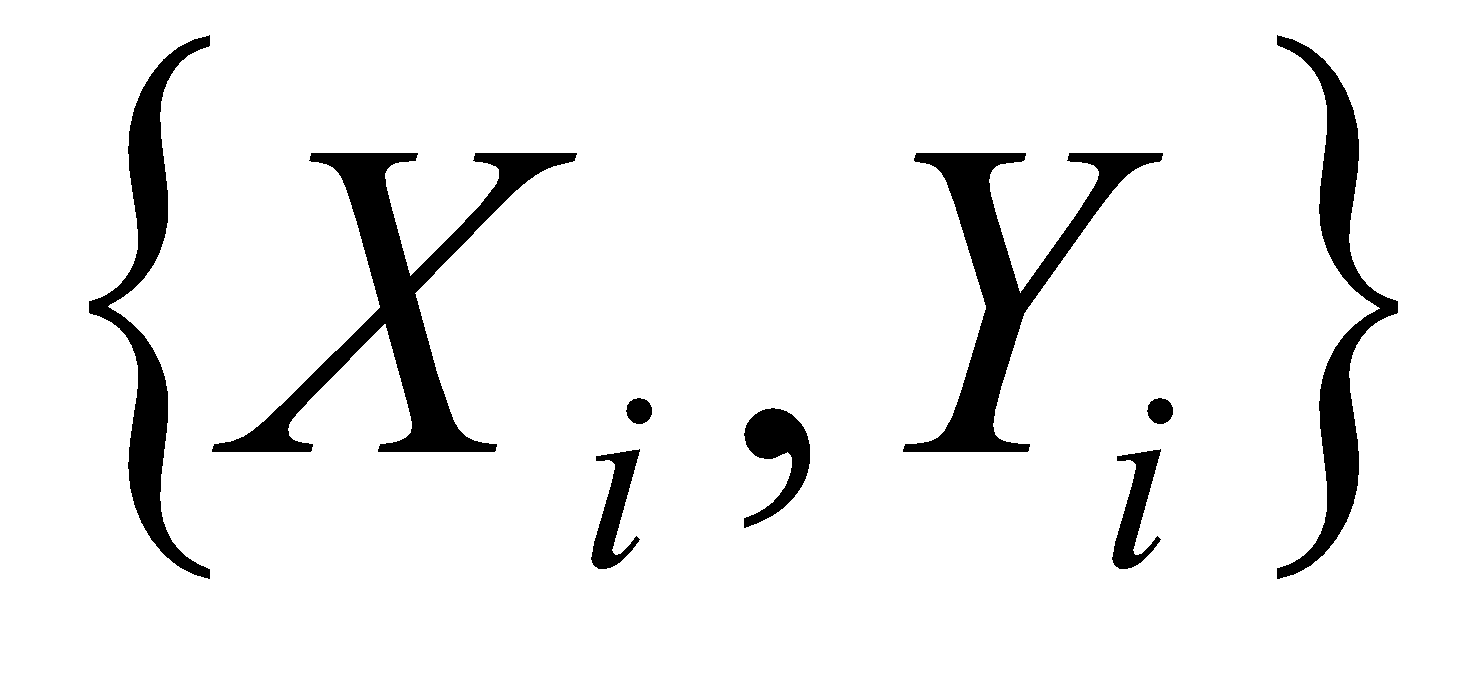
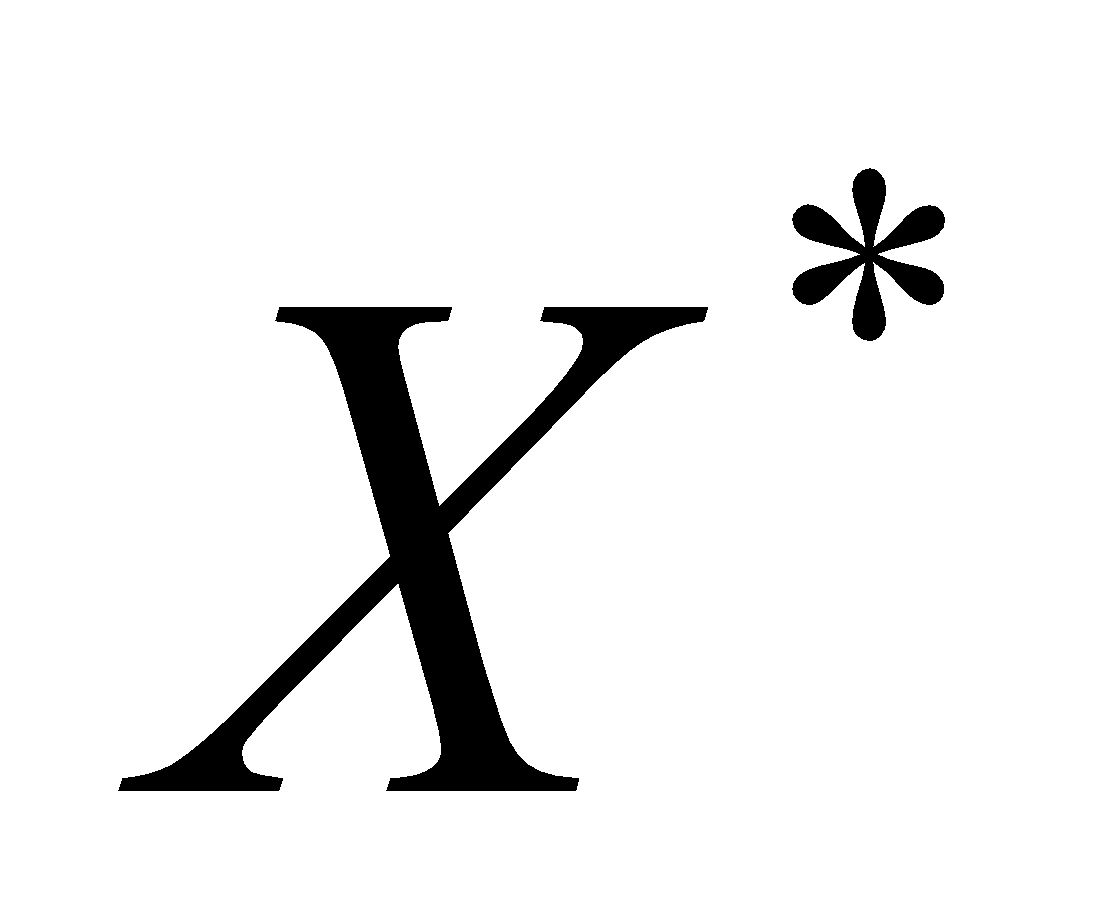
[**5**](#_z6ffj49f98tf) **Исходный код программы** 16

[**6**](#_dm4yp7psz4yn) **Выводы** 27

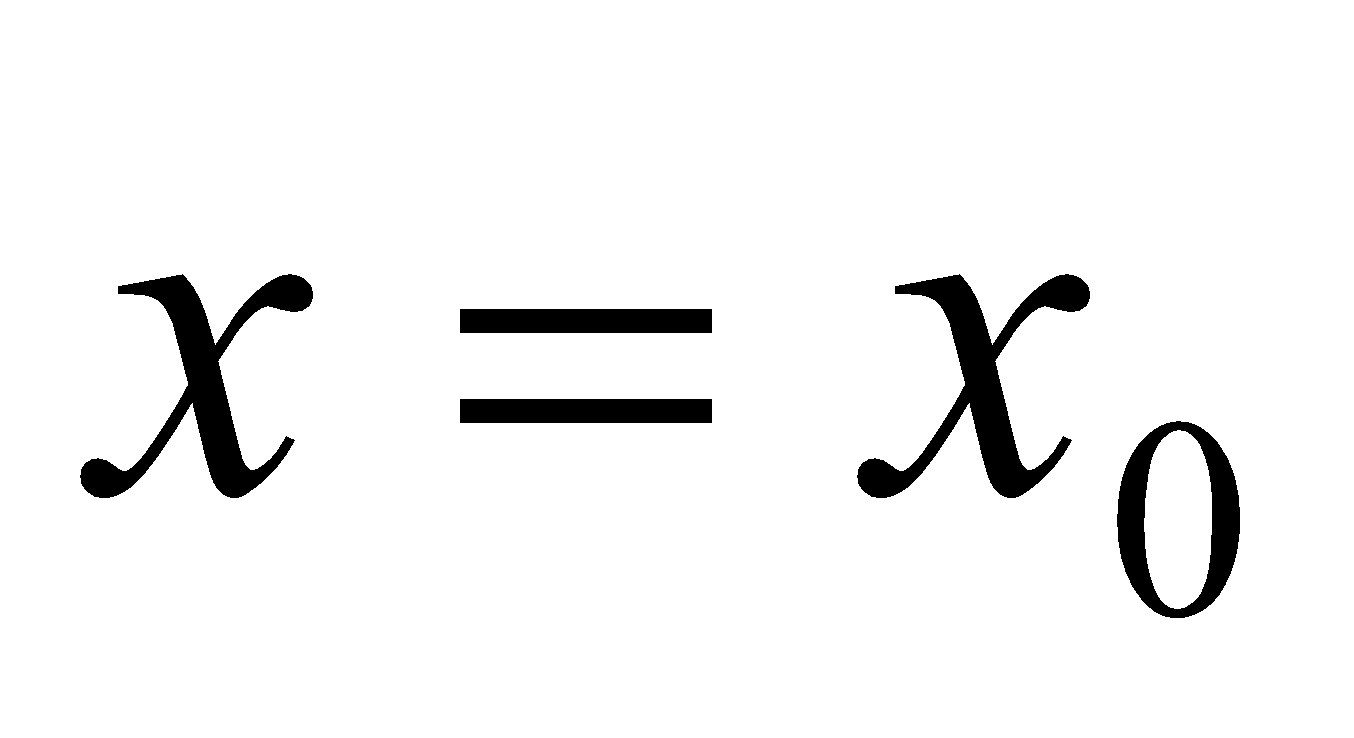
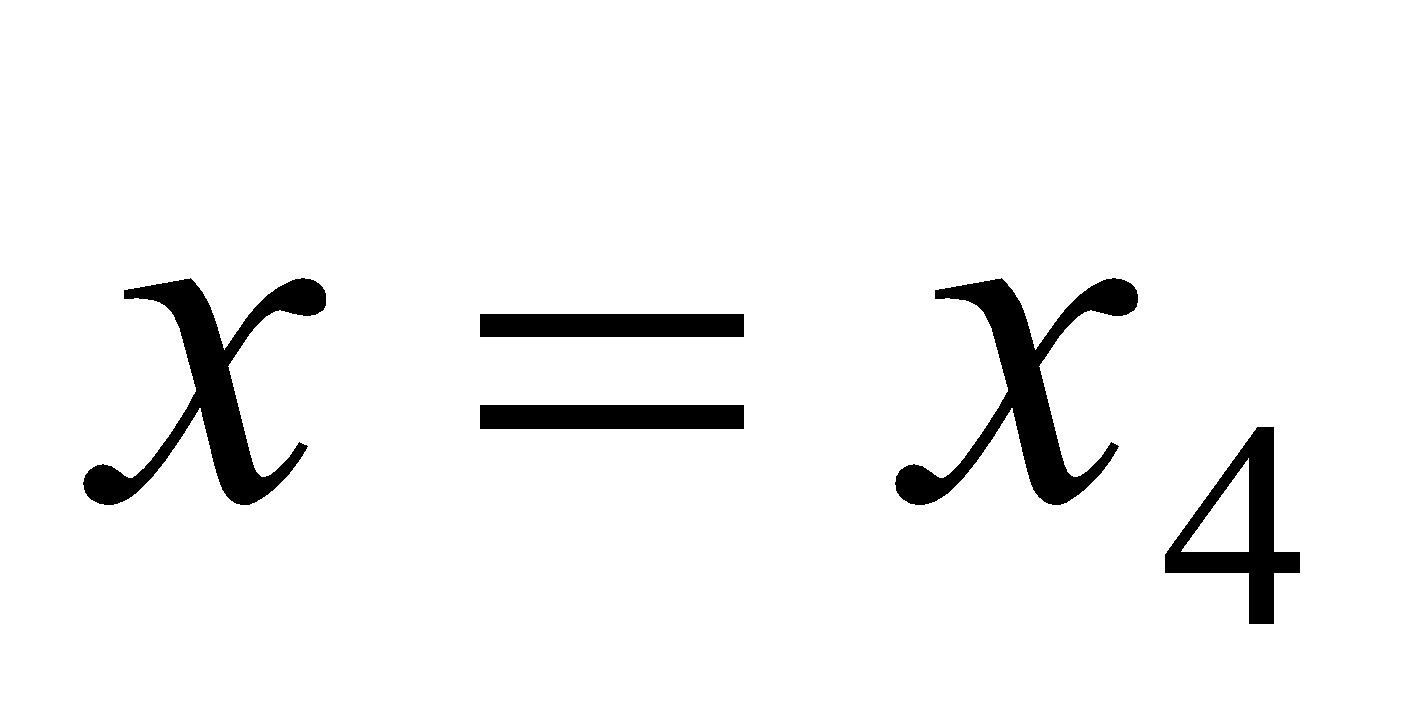
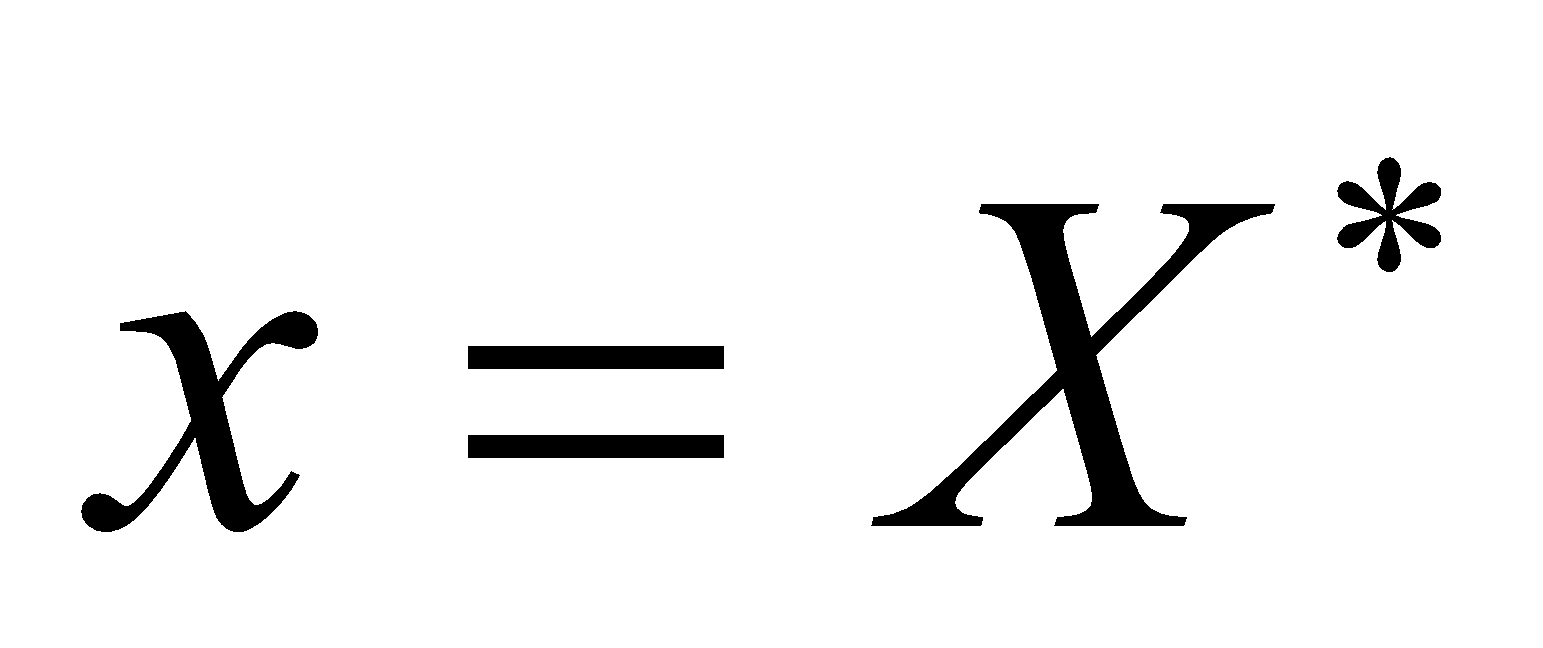
# **Тема**

Приближение функций. Численные дифференцирование и интегрирование.

# **Задание**

3.1. Используя таблицу значений  функции , вычисленных в точках  построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки . Вычислить значение погрешности интерполяции в точке .

9. , a) ; б) ; .

3.2. Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при  и . Вычислить значение функции в точке .

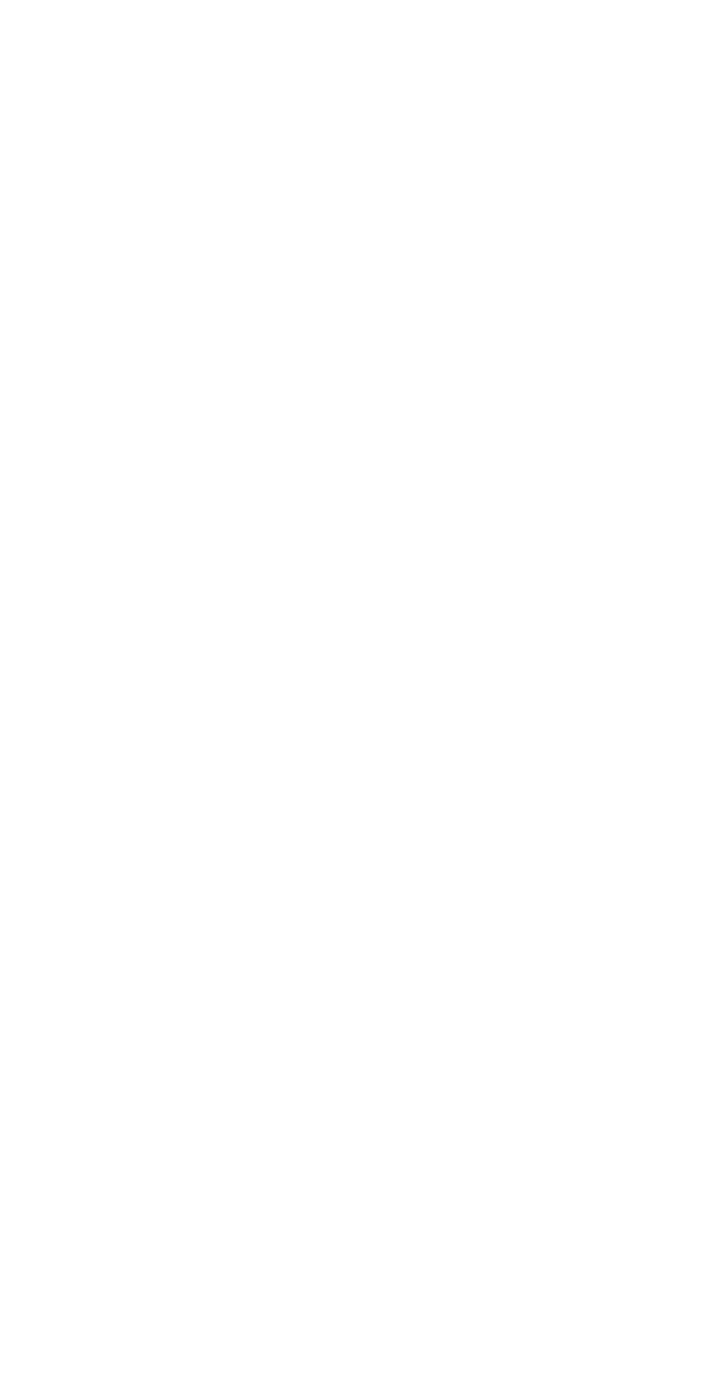
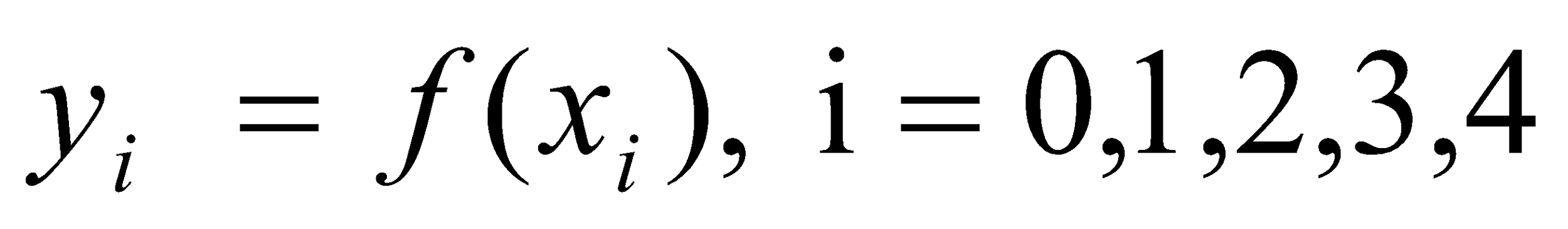
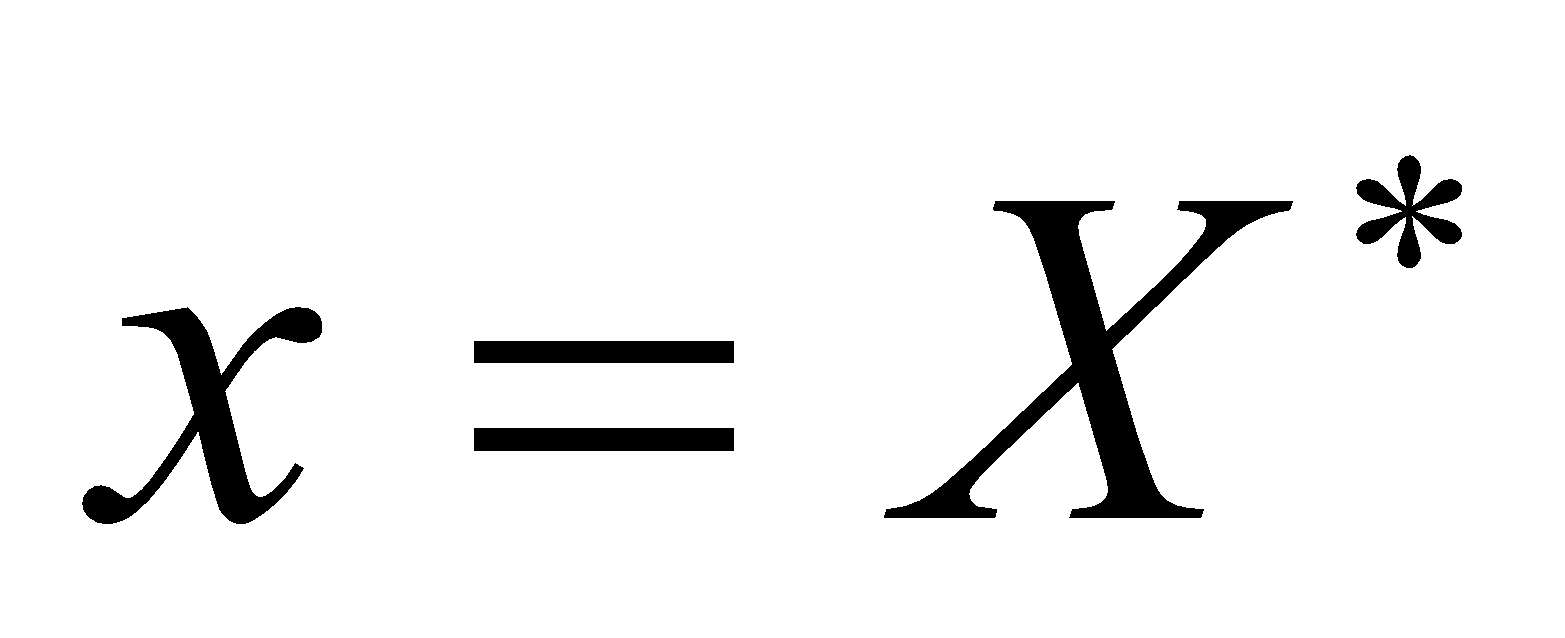
9. 0.1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | -0.4 | -0.1 | 0.2 | 0.5 | 0.8 |
|  | 1.9823 | 1.6710 | 1.3694 | 1.0472 | 0.64350 |

3.3. Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены a) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

9.

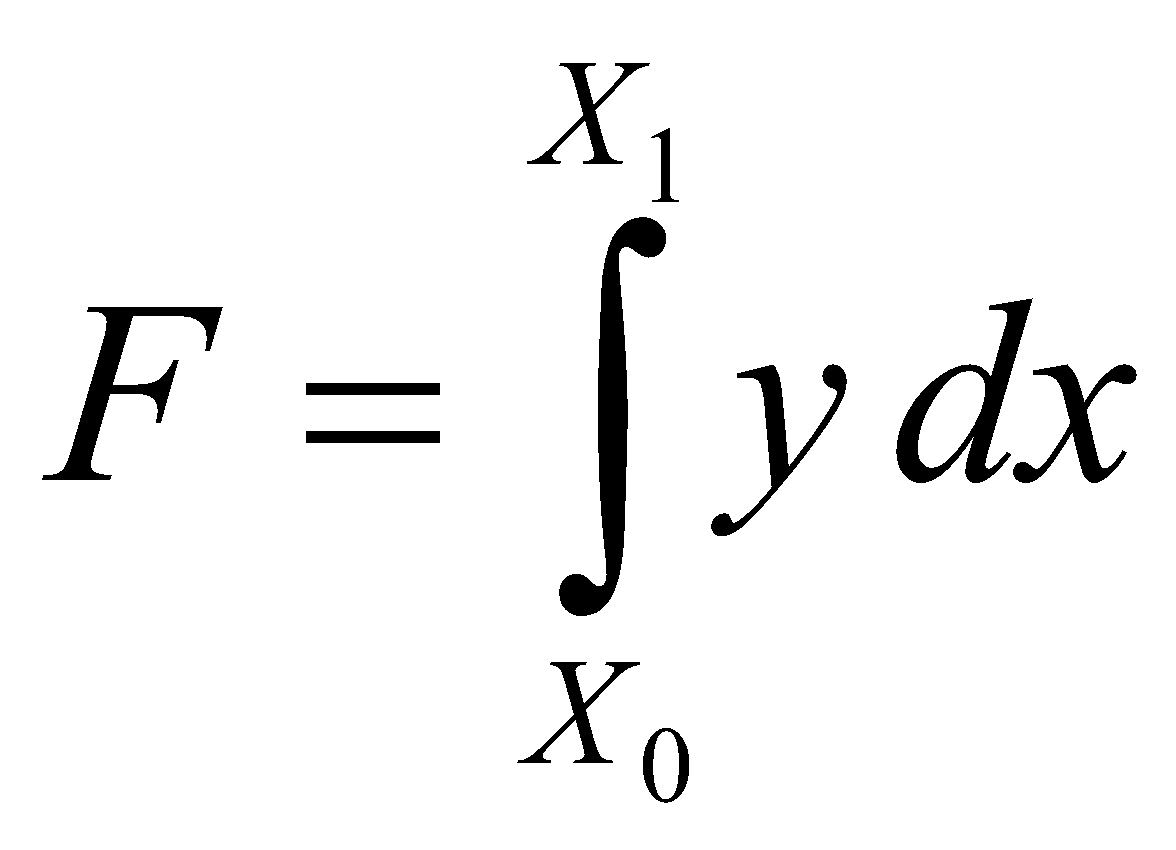
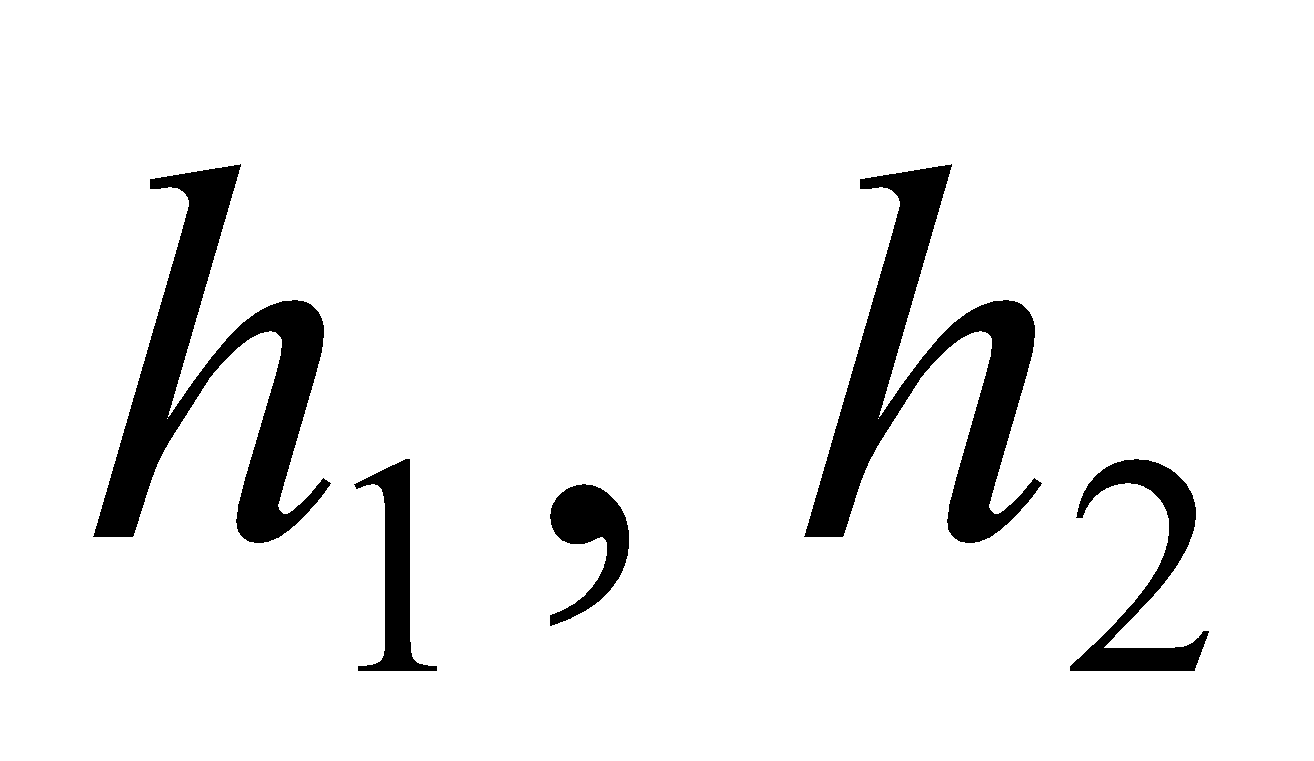
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
|  | -0.7 | -0.4 | -0.1 | 0.2 | 0.5 | 0.8 |
|  | 2.3462 | 1.9823 | 1.671 | 1.3694 | 1.0472 | 0.6435 |

3.4. Вычислить первую и вторую производную от таблично заданной функции в точке .

.

9 1.0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | -1.0 | 0.0 | 1.0 | 2.0 | 3.0 |
|  | 2.3562 | 1.5708 | 0.7854 | 0.46365 | 0.32175 |

3.5. Вычислить определенный интеграл , методами прямоугольников, трапеций, Симпсона с шагами . Оценить погрешность вычислений, используя Метод Рунге-Ромберга:

9. , ;

# **Теория**

**Интерполяция.** Пусть на отрезке задано множество несовпадающих точек (интерполяционных узлов), в которых известны значения функции . Приближающая функция такая, что выполняются равенства

называется интерполяционной.

Интерполяционный многочлен, записанный в форме , называется **интерполяционным многочленом Лагранжа**.

Введем понятие разделенной разности. Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются и определяются через разделенные разности нулевого порядка: *.* Разделенная разность порядка : .

Пусть известны значения аппроксимируемой функции в точках . Интерполяционный многочлен, значения которого в узлах интерполяции совпадают со значениями функции может быть записан в виде: . Запись многочлена в этой формуле есть так называемый **интерполяционный многочлен Ньютона**.

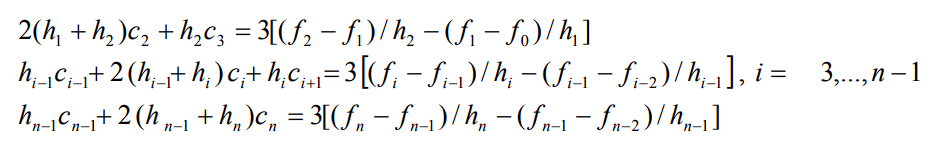
Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается 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:

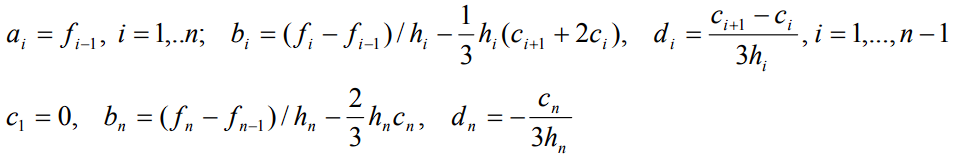
На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т. е. определить 4n неизвестных . Эти коэффициенты ищутся из условий в узлах сетки.

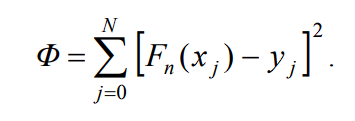
Если ввести обозначение , и исключить из системы , то можно получить систему из линейных алгебраических уравнений относительно с трехдиагональной матрицей:



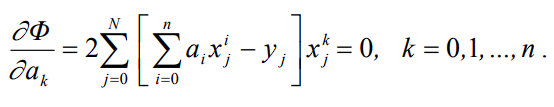
Остальные коэффициенты сплайнов могут быть восстановлены по формулам:



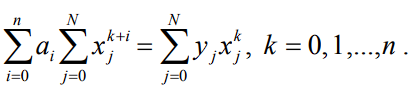
**Метод наименьших квадратов.** Пусть задана таблично в узлах функция . При этом значения функции определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени , у которого неизвестны коэффициенты . Неизвестные коэффициенты будем находить из условия минимума квадратичного отклонения многочлена от таблично заданной функции.



Минимума можно добиться только за счет изменения коэффициентов многочлена . Необходимые условия экстремума имеют вид

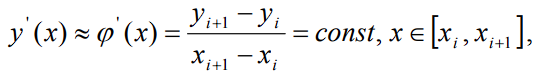


Эту систему для удобства преобразуют к следующему виду:

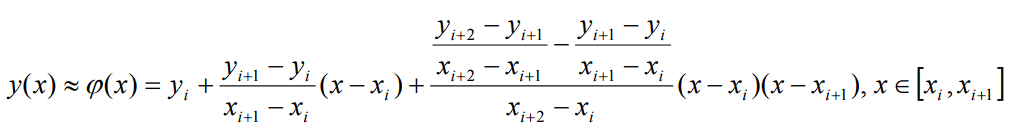


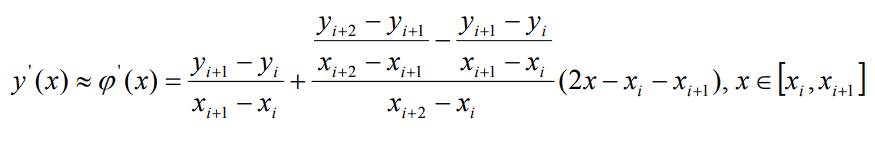
Система называется нормальной системой метода наименьших квадратов (МНК) представляет собой систему линейных алгебраических уравнений относительно коэффициентов . Решив систему, построим многочлен , приближающий функцию и минимизирующий квадратичное отклонение.

**Численное дифференцирование.** При решении практических задач, как правило, используются аппроксимации первых и вторых производных. В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой . В этом случае:



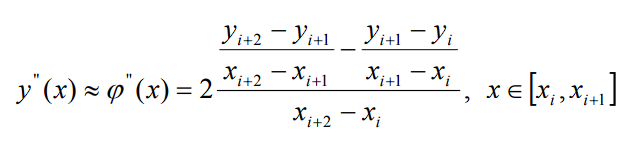
производная является кусочно-постоянной функцией и рассчитывается, по формуле с первым порядком точности в крайних точках интервала, и со вторым порядком точности в средней точке интервала.

При использовании для аппроксимации таблично заданной функции интерполяционного многочлена второй степени имеем:  




При равностоящих точках разбиения, данная формула обеспечивает второй порядок точности.

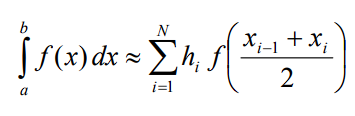
Для вычисления второй производной необходимо использовать интерполяционный многочлен, как минимум второй степени. После дифференцирования многочлена получаем



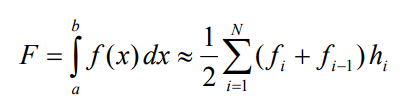
**Численное интегрирование.** Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл не удается. Отрезок разбивают точками с достаточно мелким шагом и на одном или нескольких отрезках подынтегральную функцию заменяют такой приближающей , так что она, во-первых, близка , а, вовторых, интеграл от легко вычисляется. Рассмотрим наиболее простой и часто применяемый способ, когда подынтегральную функцию заменяют на интерполяционный многочлен .

При использовании интерполяционных многочленов различной степени получают формулы численного интегрирования различного порядка точности.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку , получим **формулу прямоугольников**.

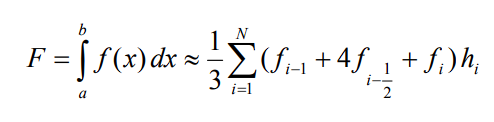


В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т. е. заменить функцию многочленом Лагранжа первой степени.

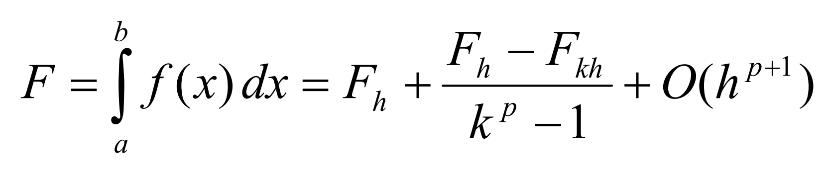


Эта формула носит название **формулы трапеций**.

Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой – интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования:

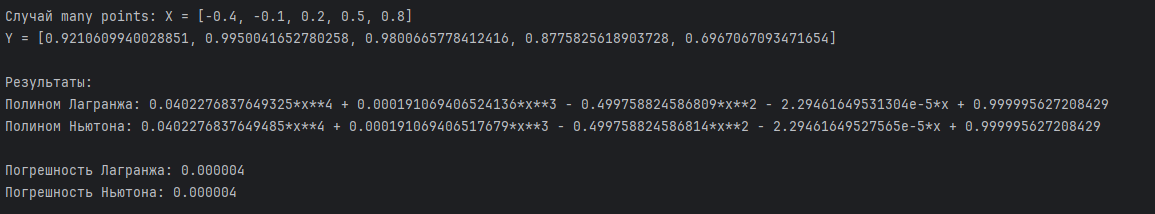
Для случая , получим **формулу Симпсона** (парабол)  


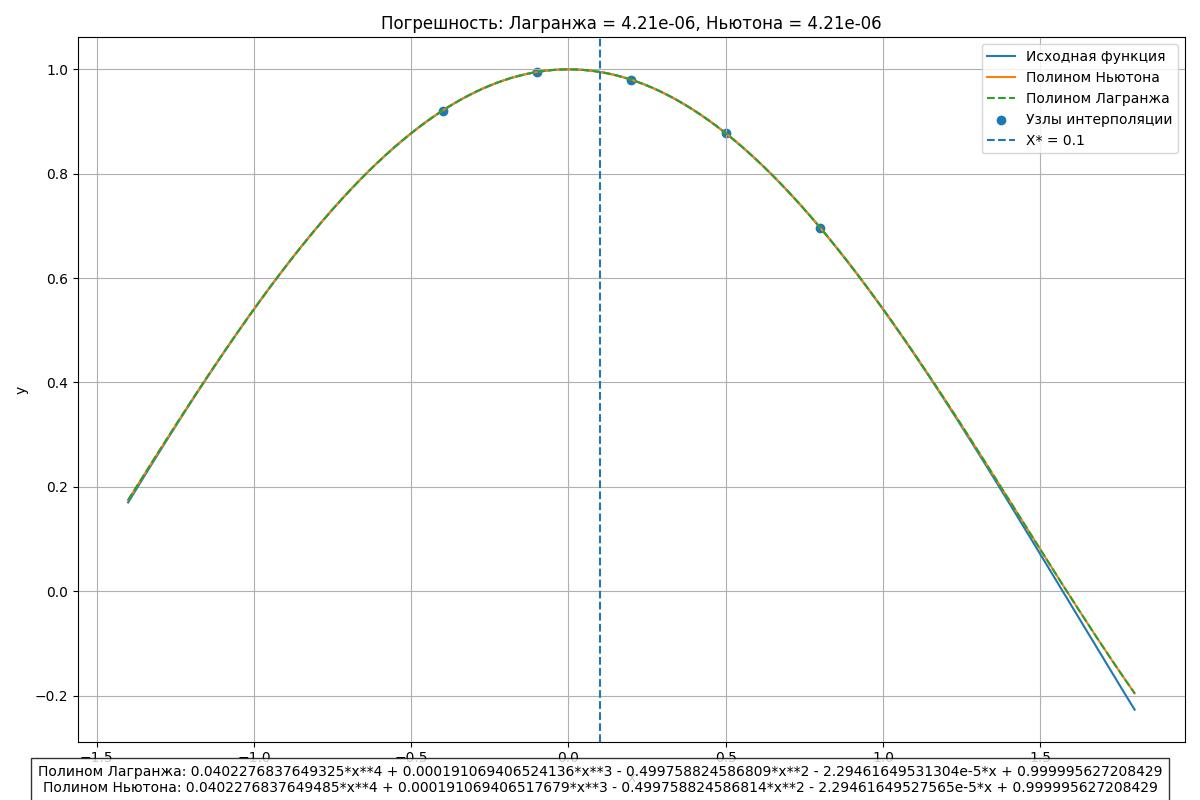
Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла на сетке с шагом и сетке с шагом , то



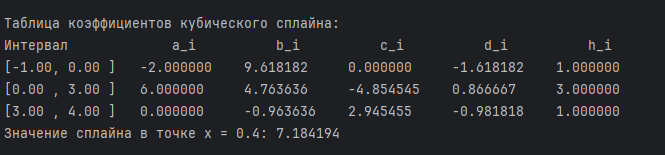
# **Демонстрация работы программы**

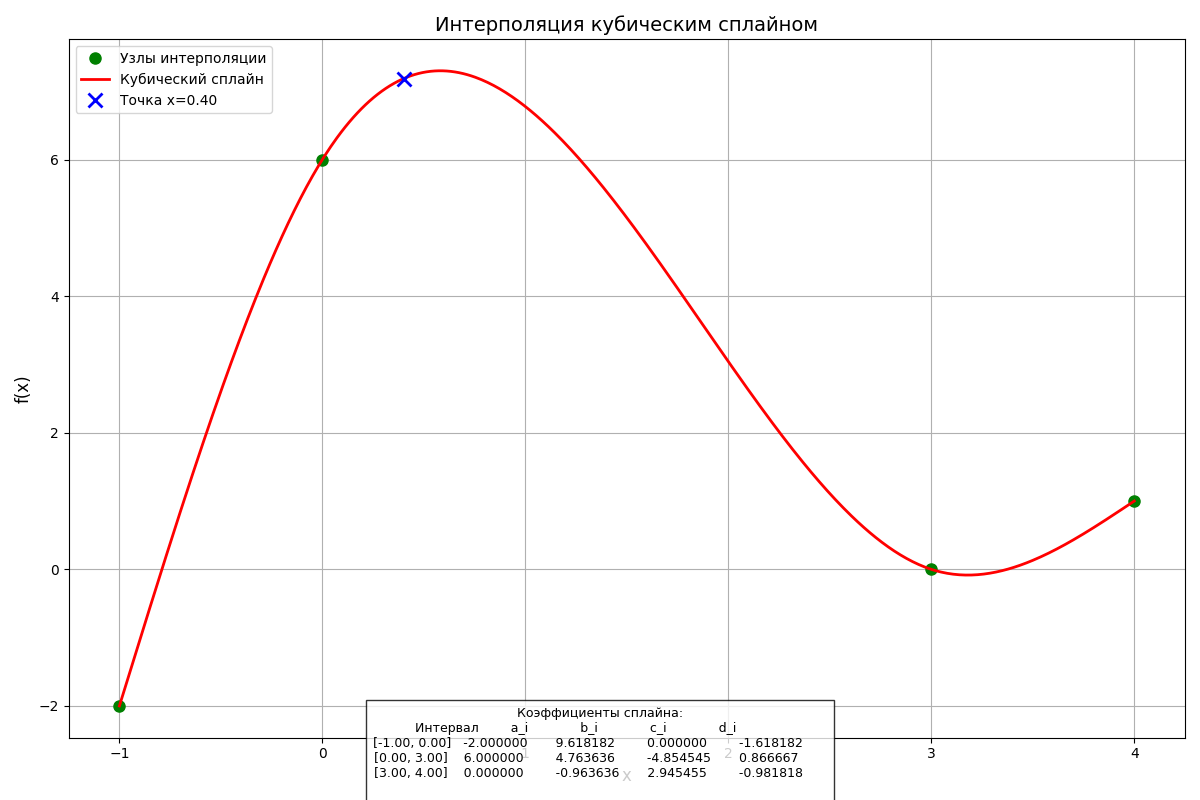
3.1) Интерполяционные многочлены Лагранжа и Ньютона



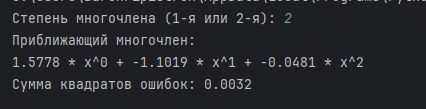


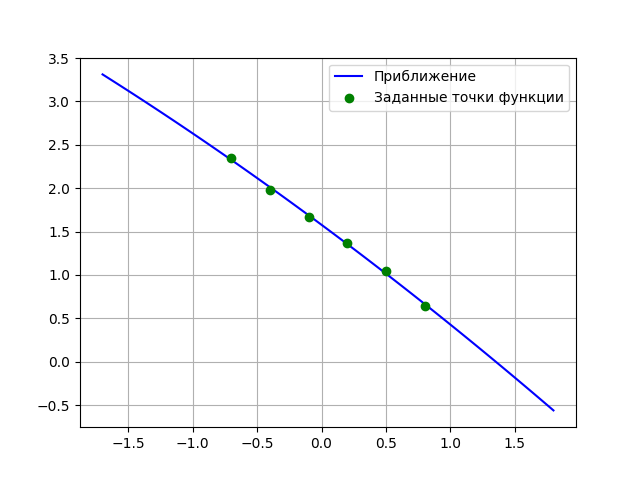
3.2) Кубический сплайн



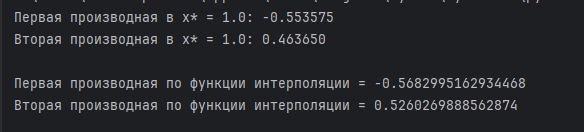


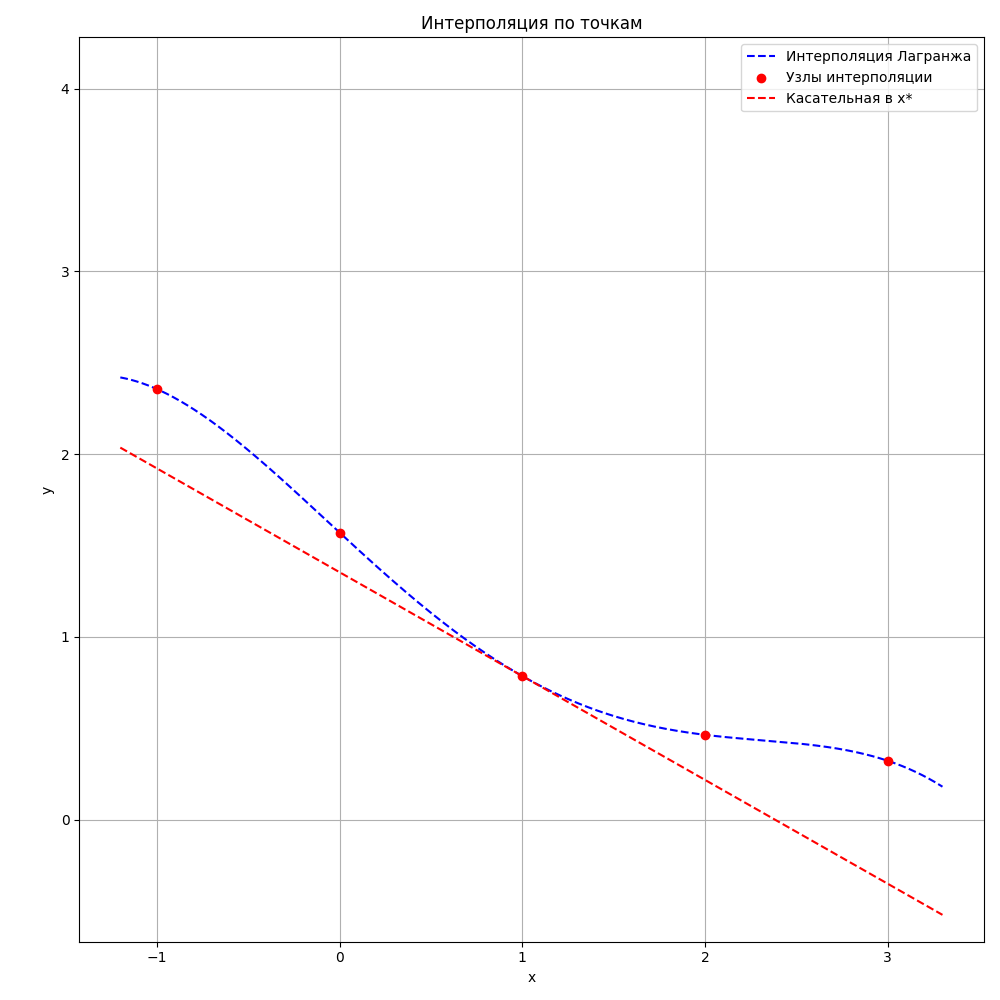
3.3) МНК



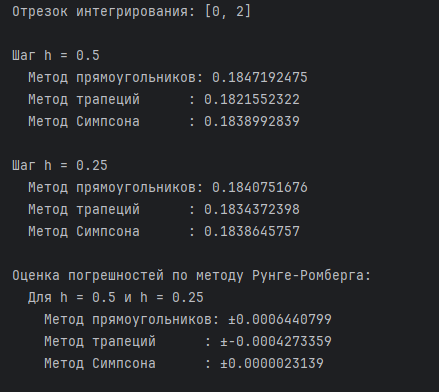


3.4) Численное дифференцирование





3.5) Численное интегрирование



# **Исходный код программы**

Код был реализован на языке Python.

* 1. Интерполяционные многочлены Лагранжа и Ньютона

1. import operator  
   import math  
   from functools import reduce  
   from typing import List, Tuple  
     
   import numpy as np  
   import matplotlib.pyplot as plt  
   import sympy  
   from sympy import simplify, symbols, lambdify  
     
     
   def original\_function(x: float) -> float:  
    *"""Исходная функция для интерполяции"""* return np.cos(x)  
     
     
   def compute\_lagrange\_polynomial(  
    x\_nodes: List[float], y\_nodes: List[float], symbolic\_x: symbols  
   ) -> "sympy.Expr":  
    *"""Строит многочлен Лагранжа по заданным узлам"""* n = len(x\_nodes)  
    if len(y\_nodes) != n:  
    raise ValueError("Количество X и Y должно совпадать")  
     
    lagrange\_terms = []  
    for i in range(n):  
    term = y\_nodes[i]  
    for j in range(n):  
    if j != i:  
    term \*= simplify((symbolic\_x - x\_nodes[j]) / (x\_nodes[i] - x\_nodes[j]))  
    lagrange\_terms.append(term)  
     
    poly = simplify(sum(lagrange\_terms))  
    return poly  
     
     
   def compute\_newton\_polynomial(  
    x\_nodes: List[float], y\_nodes: List[float], symbolic\_x: symbols  
   ) -> "sympy.Expr":  
    *"""Строит многочлен Ньютона с разделенными разностями"""* n = len(x\_nodes)  
    if len(y\_nodes) != n:  
    raise ValueError("Количество X и Y должно совпадать")  
     
    # Таблица разделенных разностей  
    divided\_diffs = y\_nodes.copy()  
    coefficients = [divided\_diffs[0]]  
     
    for order in range(1, n):  
    for j in range(n - order):  
    divided\_diffs[j] = (divided\_diffs[j + 1] - divided\_diffs[j]) / (  
    x\_nodes[j + order] - x\_nodes[j]  
    )  
    coefficients.append(divided\_diffs[0])  
     
    # Построение полинома  
    newton\_terms = []  
    for i, coef in enumerate(coefficients):  
    term = coef  
    for j in range(i):  
    term \*= simplify((symbolic\_x - x\_nodes[j]))  
    newton\_terms.append(term)  
     
    poly = simplify(sum(newton\_terms))  
    return poly  
     
     
   def format\_polynomial(poly: "sympy.Expr") -> str:  
    *"""Форматирует полином в читаемом виде."""* from sympy import expand  
    return str(expand(poly))  
     
     
   def compute\_interpolation\_errors(  
    x\_nodes: List[float], y\_nodes: List[float], test\_point: float  
   ) -> Tuple[float, float, str, str]:  
    *"""Вычисляет погрешности и возвращает полиномы."""* symbolic\_x = symbols("x", real=True)  
    lagrange\_poly = compute\_lagrange\_polynomial(x\_nodes, y\_nodes, symbolic\_x)  
    newton\_poly = compute\_newton\_polynomial(x\_nodes, y\_nodes, symbolic\_x)  
     
    true\_value = original\_function(test\_point)  
    lagrange\_value = float(lagrange\_poly.subs(symbolic\_x, test\_point))  
    newton\_value = float(newton\_poly.subs(symbolic\_x, test\_point))  
     
    return (  
    abs(lagrange\_value - true\_value),  
    abs(newton\_value - true\_value),  
    format\_polynomial(lagrange\_poly),  
    format\_polynomial(newton\_poly)  
    )  
     
     
   def plot\_interpolation\_comparison(  
    x\_nodes: List[float],  
    y\_nodes: List[float],  
    test\_point: float,  
    lagrange\_error: float,  
    newton\_error: float,  
    lagrange\_poly: str,  
    newton\_poly: str  
   ) -> None:  
    *"""Визуализация с отображением полиномов."""* symbolic\_x = symbols("x", real=True)  
    lagrange\_poly\_expr = compute\_lagrange\_polynomial(x\_nodes, y\_nodes, symbolic\_x)  
    newton\_poly\_expr = compute\_newton\_polynomial(x\_nodes, y\_nodes, symbolic\_x)  
     
    lagrange\_func = lambdify(symbolic\_x, lagrange\_poly\_expr, "numpy")  
    newton\_func = lambdify(symbolic\_x, newton\_poly\_expr, "numpy")  
     
    x\_plot = np.linspace(min(x\_nodes) - 1, max(x\_nodes) + 1, 1000)  
    y\_true = original\_function(x\_plot)  
    y\_lagrange = lagrange\_func(x\_plot)  
    y\_newton = newton\_func(x\_plot)  
     
    plt.figure(figsize=(12, 8))  
     
    # Добавляем текст с полиномами  
    plt.figtext(0.5, 0.01,  
    f"Полином Лагранжа: {lagrange\_poly}\nПолином Ньютона: {newton\_poly}",  
    ha="center", fontsize=10, bbox={"facecolor": "white", "alpha": 0.8, "pad": 5})  
     
    plt.plot(x\_plot, y\_true, label="Исходная функция")  
    plt.plot(x\_plot, y\_newton, label="Полином Ньютона")  
    plt.plot(x\_plot, y\_lagrange, "--", label="Полином Лагранжа")  
    plt.scatter(x\_nodes, y\_nodes, marker="o", label="Узлы интерполяции")  
    plt.axvline(test\_point, linestyle="--", label=f"X\* = {test\_point}")  
    plt.title(f"Погрешность: Лагранжа = {lagrange\_error:.2e}, Ньютона = {newton\_error:.2e}")  
    plt.xlabel("x")  
    plt.ylabel("y")  
    plt.legend()  
    plt.grid(True)  
    plt.tight\_layout()  
    plt.show()  
     
     
   def run\_interpolation\_example(  
    x\_nodes: List[float], test\_point: float, case\_name: str  
   ) -> None:  
    *"""Запускает пример интерполяции с выводом полиномов."""* is\_test = True  
     
    if test\_point > max(x\_nodes):  
    print("ERROR: Точка за границами интерполяции")  
    exit(1)  
     
    if test\_point < min(x\_nodes) or test\_point > max(x\_nodes):  
    is\_test = False  
     
    y\_nodes = [original\_function(x) for x in x\_nodes]  
    print(f"\nСлучай {case\_name}: X = {x\_nodes}")  
    print(f"Y = {[float(node) for node in y\_nodes]}")  
     
    error\_lagrange, error\_newton, lagrange\_poly, newton\_poly = compute\_interpolation\_errors(  
    x\_nodes, y\_nodes, test\_point  
    )  
     
    print("\nРезультаты:")  
    print(f"Полином Лагранжа: {lagrange\_poly}")  
    print(f"Полином Ньютона: {newton\_poly}")  
     
    if not is\_test:  
    print(f"Тестовая точка выходит за пределы отрезка, посчитать погрешность невозможно")  
    else:  
    print(f"\nПогрешность Лагранжа: {error\_lagrange:.6f}")  
    print(f"Погрешность Ньютона: {error\_newton:.6f}")  
     
    plot\_interpolation\_comparison(  
    x\_nodes, y\_nodes, test\_point,  
    error\_lagrange, error\_newton,  
    lagrange\_poly, newton\_poly  
    )  
     
     
   # Базовый случай  
   # run\_interpolation\_example(  
   # x\_nodes=[-0.4, -0.1, 0.2, 0.5], test\_point=0.1, case\_name="base"  
   # )  
     
   # Больше точек  
   run\_interpolation\_example(  
    x\_nodes=[-0.4, -0.1, 0.2, 0.5, 0.8], test\_point=0.1, case\_name="many points"  
   )  
     
   # Меньше точек  
   # run\_interpolation\_example(  
   # x\_nodes=[-0.4, -0.1, 0.2], test\_point=0.1, case\_name="little point"  
   # )  
     
    # X\* = 100  
   # run\_interpolation\_example(  
   # x\_nodes=[-0.4, -0.1, 0.2, 0.5], test\_point=100, case\_name="X\* = 100"  
   # )  
     
   # Хитрый тест  
   # run\_interpolation\_example(  
   # x\_nodes=[-1, 0, 3, 4], test\_point=0.1, case\_name="cool test"  
   # )

3.2) Кубический сплайн

from typing import Tuple, Union  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
def solve\_tridiagonal\_system(matrix\_A: np.ndarray, matrix\_B: np.ndarray) -> np.ndarray:  
 *"""Решает трехдиагональную систему уравнений методом прогонки (Томаса).  
  
 Args:  
 matrix\_A: Матрица коэффициентов (n x n).  
 matrix\_B: Вектор правых частей (n x 1).  
  
 Returns:  
 Вектор решений.  
 """* n = len(matrix\_A)  
 augmented\_matrix = np.concatenate((matrix\_A, matrix\_B), axis=1)  
  
 # Инициализация прогоночных коэффициентов  
 a = np.zeros(n + 1) # Нижняя диагональ (a[0] не используется)  
 b = np.zeros(n + 1) # Главная диагональ  
 c = np.zeros(n + 1) # Верхняя диагональ  
 d = np.zeros(n + 1) # Правые части  
  
 # Заполнение коэффициентов  
 for row in range(n):  
 for col in range(len(augmented\_matrix[0])):  
 if row == col:  
 b[row + 1] = augmented\_matrix[row][col]  
 elif row - 1 == col:  
 a[row + 1] = augmented\_matrix[row][col]  
 elif row + 1 == col and row + 1 < n:  
 c[row + 1] = augmented\_matrix[row][col]  
 elif col == len(augmented\_matrix[0]) - 1:  
 d[row + 1] = augmented\_matrix[row][col]  
  
 # Прямой ход прогонки  
 P = np.zeros(n + 1)  
 Q = np.zeros(n + 1)  
 for i in range(1, n + 1):  
 denominator = b[i] + a[i] \* P[i - 1]  
 P[i] = -c[i] / denominator  
 Q[i] = (d[i] - a[i] \* Q[i - 1]) / denominator  
  
 # Обратный ход прогонки  
 x = np.zeros(n + 1)  
 x[n] = Q[n]  
 for i in range(n - 1, 0, -1):  
 x[i] = P[i] \* x[i + 1] + Q[i]  
  
 return x[1:n + 1]  
  
  
def compute\_cubic\_spline\_coefficients(  
 x\_nodes: np.ndarray,  
 y\_values: np.ndarray,  
 verbose: bool = True # Добавлен флаг для вывода таблицы  
) -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:  
 *"""Вычисляет коэффициенты кубического сплайна и выводит таблицу."""* n = len(x\_nodes)  
 if n != len(y\_values):  
 raise ValueError("Количество x и y значений должно совпадать")  
  
 h = np.diff(x\_nodes)  
  
 # Построение системы уравнений  
 A = np.zeros((n - 2, n - 2))  
 B = np.zeros(n - 2)  
  
 for i in range(1, n - 1):  
 B[i - 1] = 3 \* ((y\_values[i + 1] - y\_values[i]) / h[i] -  
 (y\_values[i] - y\_values[i - 1]) / h[i - 1])  
  
 for i in range(n - 2):  
 if i > 0:  
 A[i, i - 1] = h[i]  
 A[i, i] = 2 \* (h[i] + h[i + 1])  
 if i < n - 3:  
 A[i, i + 1] = h[i + 1]  
  
 # Решение системы  
 c\_internal = solve\_tridiagonal\_system(A, B.reshape(-1, 1))  
 c = np.zeros(n)  
 c[1:-1] = c\_internal  
  
 # Вычисление коэффициентов  
 a = y\_values[:-1]  
 b = np.zeros(n - 1)  
 d = np.zeros(n - 1)  
  
 for i in range(n - 1):  
 b[i] = ((y\_values[i + 1] - y\_values[i]) / h[i] -  
 h[i] \* (2 \* c[i] + c[i + 1]) / 3)  
 d[i] = (c[i + 1] - c[i]) / (3 \* h[i])  
  
 # Вывод таблицы коэффициентов  
 if verbose:  
 print("\nТаблица коэффициентов кубического сплайна:")  
 print("{:<20} {:<12} {:<12} {:<12} {:<12} {:<12}".format(  
 "Интервал", "a\_i", "b\_i", "c\_i", "d\_i", "h\_i"))  
 for i in range(n - 1):  
 print("[{:<5.2f}, {:<5.2f}] {:<12.6f} {:<12.6f} {:<12.6f} {:<12.6f} {:<12.6f}".format(  
 x\_nodes[i], x\_nodes[i + 1], a[i], b[i], c[i], d[i], h[i]))  
  
 return a, b, c[:-1], d  
  
  
def evaluate\_cubic\_spline(  
 x\_nodes: np.ndarray,  
 coefficients: Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray],  
 x\_eval: Union[float, np.ndarray]  
) -> Union[float, np.ndarray]:  
 *"""Вычисляет значение кубического сплайна в заданных точках.  
  
 Args:  
 x\_nodes: Узлы интерполяции.  
 coefficients: Коэффициенты сплайна (a, b, c, d).  
 x\_eval: Точка(и) для вычисления.  
  
 Returns:  
 Значение(я) сплайна в точках x\_eval.  
 """* if type(x\_eval) is not np.ndarray:  
 if x\_eval < min(x\_nodes) or x\_eval > max(x\_nodes):  
 print(f"Тестовая точка x = {x\_eval} выходит за пределы отрезка [{min(x\_nodes)}; {max(x\_nodes)}]")  
 return  
 else:  
 pass  
  
 a, b, c, d = coefficients  
  
 if np.isscalar(x\_eval):  
 i = np.clip(np.searchsorted(x\_nodes, x\_eval) - 1, 0, len(a) - 1)  
 dx = x\_eval - x\_nodes[i]  
 return a[i] + b[i] \* dx + c[i] \* dx \*\* 2 + d[i] \* dx \*\* 3  
 else:  
 result = np.zeros\_like(x\_eval)  
 for idx, x\_val in enumerate(x\_eval):  
 i = np.clip(np.searchsorted(x\_nodes, x\_val) - 1, 0, len(a) - 1)  
 dx = x\_val - x\_nodes[i]  
 result[idx] = a[i] + b[i] \* dx + c[i] \* dx \*\* 2 + d[i] \* dx \*\* 3  
 return result  
  
  
def plot\_spline\_results(  
 x\_nodes: np.ndarray,  
 y\_values: np.ndarray,  
 coefficients: Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray],  
 eval\_point: float  
) -> None:  
 *"""Визуализирует результаты с таблицей коэффициентов."""* eval\_result = evaluate\_cubic\_spline(x\_nodes, coefficients, eval\_point)  
  
 x\_plot = np.linspace(x\_nodes[0], x\_nodes[-1], 500)  
 y\_plot = evaluate\_cubic\_spline(x\_nodes, coefficients, x\_plot)  
  
 plt.figure(figsize=(12, 8))  
  
 # Основной график  
 plt.plot(x\_nodes, y\_values, 'go', markersize=8, label='Узлы интерполяции')  
 plt.plot(x\_plot, y\_plot, 'r-', linewidth=2, label='Кубический сплайн')  
 plt.plot(eval\_point, eval\_result, 'bx', markersize=10,  
 markeredgewidth=2, label=f'Точка x={eval\_point:.2f}')  
  
 # Добавляем таблицу коэффициентов на график  
 a, b, c, d = coefficients  
 table\_data = []  
 for i in range(len(a)):  
 table\_data.append([  
 f"[{x\_nodes[i]:.2f}, {x\_nodes[i + 1]:.2f}]",  
 f"{a[i]:.6f}",  
 f"{b[i]:.6f}",  
 f"{c[i]:.6f}",  
 f"{d[i]:.6f}"  
 ])  
  
 # Создаем текст для отображения  
 table\_text = "Коэффициенты сплайна:\n"  
 table\_text += "{:<15} {:<15} {:<15} {:<15} {:<15}\n".format(  
 "Интервал", "a\_i", "b\_i", "c\_i", "d\_i")  
 for row in table\_data:  
 table\_text += "{:<15} {:<15} {:<15} {:<15} {:<15}\n".format(\*row)  
  
 plt.figtext(0.5, 0.01, table\_text, ha="center", fontsize=9,  
 bbox={"facecolor": "white", "alpha": 0.8, "pad": 5})  
  
 plt.xlabel('x', fontsize=12)  
 plt.ylabel('f(x)', fontsize=12)  
 plt.title('Интерполяция кубическим сплайном', fontsize=14)  
 plt.legend(fontsize=10, loc='upper left')  
 plt.grid(True)  
 plt.tight\_layout()  
 plt.show()  
  
x\_nodes = np.array([-1, 0, 3, 4])  
y\_values = np.array([-2, 6, 0, 1])  
eval\_point = 0.4  
  
spline\_coeffs = compute\_cubic\_spline\_coefficients(x\_nodes, y\_values)  
  
result = evaluate\_cubic\_spline(x\_nodes, spline\_coeffs, eval\_point)  
  
if result is not None:  
 print(f"Значение сплайна в точке x = {eval\_point}: {result:.6f}")  
  
 plot\_spline\_results(x\_nodes, y\_values, spline\_coeffs, eval\_point)

* 1. МНК

1. import numpy as np  
   import matplotlib.pyplot as plt  
     
     
   def getValue(as\_: list, x):  
    value = 0  
    for i in range(len(as\_)):  
    value += as\_[i] \* x \*\* i  
    return value  
     
     
   n = int(input("Степень многочлена (1-я или 2-я): ")) + 1  
     
   if n not in [2, 3]:  
    print("Степень многочлена должна быть первой или второй")  
    exit(0)  
     
   # Базовый случай  
   xs = np.array([-0.7, -0.4, -0.1, 0.2, 0.5, 0.8])  
   ys = np.array([2.3462, 1.9823, 1.671, 1.3694, 1.0472, 0.6435])  
     
   # Убрали точки  
   # xs = np.array([-0.7, -0.4, -0.1, 0.2, 0.5])  
   # ys = np.array([2.3462, 1.9823, 1.671, 1.3694, 1.0472])  
     
   # Добавили точки  
   # xs = np.array([-0.7, -0.4, -0.1, 0.2, 0.5, 0.8, 1.1])  
   # ys = np.array([2.3462, 1.9823, 1.671, 1.3694, 1.0472, 0.6435, 0.3593])  
     
   # Ломаная функция  
   # xs = np.array([-1, 0, 3, 4])  
   # ys = np.array([-2, 6, 0, 1])  
     
   N = len(xs)  
     
   A = np.zeros((n, n))  
   b = np.zeros(n)  
   for i in range(n):  
    for j in range(N):  
    b[i] += ys[j] \* xs[j] \*\* i  
     
    for j in range(n):  
    for k in range(N):  
    A[i][j] += xs[k] \*\* (i + j)  
     
   as\_ = np.linalg.solve(A, b)  
     
   print("Приближающий многочлен:")  
   polynom = []  
   for i in range(len(as\_)):  
    polynom.append(f"{np.round(as\_[i], 4)} \* x^{i}")  
   print(" + ".join(polynom))  
     
   fs = [getValue(as\_, x) for x in xs]  
     
   error = sum([(fs[i] - ys[i]) \*\* 2 for i in range(N)])  
   print(f"Сумма квадратов ошибок: {np.round(error, 4)}")  
     
   x\_plot = np.linspace(min(xs) - 1, max(xs) + 1, 100)  
   f\_plot = [getValue(as\_, x) for x in x\_plot]  
     
   plt.plot(x\_plot, f\_plot, linestyle='-', color=(0, 0, 1), label=f"Приближение")  
   plt.scatter(xs, ys, color='green', label='Заданные точки функции', zorder=5)  
   plt.legend()  
   plt.grid(True)  
   plt.show()

3.4) Численное дифференцирование

import numpy as np  
import matplotlib.pyplot as plt  
  
  
def lagrange\_interpolation(x, xi, yi):  
 n = len(xi)  
 Lx = 0  
 # Для каждого узла интерполяции формируем лагранжев многочлен  
 for i in range(n):  
 li = 1  
 for j in range(n):  
 if i != j:  
 li \*= (x - xi[j]) / (xi[i] - xi[j])  
 Lx += yi[i] \* li  
  
 return Lx  
  
  
# Поиск подходящего интервала [x\_i, x\_{i+1}]  
def get\_index(x\_star, x):  
 for i in range(len(x) - 2): # нужно минимум 3 точки: i, i+1, i+2  
 if x[i] <= x\_star <= x[i + 1]:  
 return i  
  
 print("ERROR: x\* вне допустимого диапазона для формулы (3.20)")  
 exit(0)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Базовый случай  
 x\_star = 1.0  
 x = [-1.0, 0.0, 1.0, 2.0, 3.0]  
 y = [2.3562, 1.5708, 0.7854, 0.46365, 0.32175]  
  
 # Точка за пределами интервала  
 # x\_star = 0.4  
 # x = [0.0, 0.1, 0.2, 0.3, 0.4]  
 # y = [1.0, 1.1052, 1.2214, 1.3499, 1.4918]  
  
 # Неравномерные шаги  
 # x\_star = 0.5  
 # x = [-1.0, 0.3, 1.0, 2.5, 3.7]  
 # y = [2.3562, 1.2, 0.7, 0.4, 0.2]  
  
 if x\_star < min(x) or x\_star > max(x):  
 print("ERROR: X\* не входит в диапазон значений x")  
 exit(0)  
  
 i = get\_index(x\_star, x)  
  
 dy1 = (y[i + 1] - y[i]) / (x[i + 1] - x[i])  
 dy2 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1])  
 correction = (dy2 - dy1) / (x[i + 2] - x[i]) \* (2 \* x\_star - x[i] - x[i + 1])  
 first\_derivative = dy1 + correction  
  
 # Вторая производная по формуле (3.21)  
 second\_derivative = 2 \* (dy2 - dy1) / (x[i + 1] - x[i - 1])  
  
 print(f"Первая производная в x\* = {x\_star}: {first\_derivative:.6f}")  
 print(f"Вторая производная в x\* = {x\_star}: {second\_derivative:.6f}\n")  
  
 tan = lambda x\_star: (lagrange\_interpolation(x\_star+1e-3, x, y) - lagrange\_interpolation(x\_star, x, y))/1e-3  
 print(f"Первая производная по функции интерполяции = {tan(x\_star)}")  
  
  
 x\_vals = np.linspace(-1.2, 3.3, 4000)  
 tangent\_line = tan(x\_star)\*(x\_vals-x\_star) + lagrange\_interpolation(x\_star, x, y)  
  
 lagrange\_vals = [lagrange\_interpolation(point, x, y) for point in x\_vals]  
  
 # График  
 plt.figure(figsize=(10, 10))  
 plt.plot(x\_vals, lagrange\_vals, label='Интерполяция Лагранжа', linestyle='--', color='blue')  
  
 plt.scatter(x, y, color='red', label='Узлы интерполяции', zorder=5)  
  
 plt.plot(x\_vals, tangent\_line, '--', label='Касательная в x\*', color='red')  
  
 # Подписи и легенда  
 plt.title('Интерполяция по точкам')  
 plt.xlabel('x')  
 plt.ylabel('y')  
 plt.legend()  
 plt.grid(True)  
 plt.tight\_layout()  
 plt.axis('square')  
 plt.show()  
 plt.savefig("3\_4\_interpolation")  
  
  
 x\_vals2 = np.linspace(-1.2, 3.3, 4000)  
 derivative\_line = [tan(elem) for elem in x\_vals2]  
  
  
 plt.figure(figsize=(10, 10))  
 plt.title('Производная')  
 plt.xlabel('x')  
 plt.ylabel('y')  
 plt.grid(True)  
 plt.tight\_layout()  
 plt.axvline(x\_star, color='gray', linestyle='--', label=f"x\* = {x\_star}")  
 plt.plot(x\_vals2, derivative\_line, label='Вторая производная', linestyle='-', color='purple')  
 plt.legend()  
 plt.axis('square')  
 plt.savefig("derivative")  
  
  
 sec\_deriv = (tan(x\_star+1e-3) - tan(x\_star))/+1e-3  
 print(f"Вторая производная по функции интерполяции = {sec\_deriv}")

3.5) Численное интегрирование

import sys  
  
def f(x):  
 return x / (x\*\*2 + 9)  
  
def splitting(x0, xk, h):  
 if h <= 0:  
 raise ValueError("Шаг h должен быть положительным числом.")  
 if x0 >= xk:  
 raise ValueError("Начальная точка x0 должна быть меньше конечной точки xk.")  
 xs = []  
 x = x0  
 while x < xk + sys.float\_info.epsilon \* 10:  
 if x > xk:  
 x = xk  
 xs.append(x)  
 x += h  
 if abs(xs[-1] - xk) > sys.float\_info.epsilon \* 10:  
 xs.append(xk)  
 if len(xs) > 1 and abs(xs[-1] - xs[-2]) < sys.float\_info.epsilon \* 10 and abs(xs[-1] - xk) < sys.float\_info.epsilon \* 10:  
 xs.pop(-2)  
 return xs  
  
def rectangles(x0, xk, h):  
 xs = splitting(x0, xk, h)  
 if len(xs) < 2:  
 #интеграл на единственном отрезке оценивается по значению функции в его середине, если x0!=k и точек меньше 2  
 return 0.0 if x0 == xk else f((x0 + xk) / 2) \* (xk - x0)  
 integral = 0  
 for i in range(len(xs) - 1):  
 midpoint = (xs[i] + xs[i+1]) / 2  
 integral += h \* f(midpoint)  
 return integral  
  
def trapezoids(x0, xk, h):  
 xs = splitting(x0, xk, h)  
 if len(xs) < 2:  
 return 0.0 if x0 == xk else 0.5 \* (f(x0) + f(xk)) \* (xk - x0)  
 integral = 0  
 for i in range(len(xs) - 1):  
 integral += 0.5 \* h \* (f(xs[i]) + f(xs[i+1]))  
 return integral  
  
def simpson(x0, xk, h):  
 xs = splitting(x0, xk, h)  
 n = len(xs) - 1  
 if n <= 0:  
 return 0.0  
 if n % 2 != 0:  
 raise ValueError(f"Для метода Симпсона требуется чётное число интервалов. Получено {n}.")  
 integral = 0  
 for i in range(0, n, 2):  
 integral += h / 3 \* (f(xs[i]) + 4 \* f(xs[i+1]) + f(xs[i+2]))  
 return integral  
  
def runge\_romberg(F\_h, F\_kh, k, p):  
 return (F\_h - F\_kh) / (k\*\*p - 1)  
  
# Базовый случай  
input\_data = {  
 "x0": 0,  
 "x\_end": 2,  
 "h": [0.5, 0.25]  
}  
  
# Какой-то хитрый тест  
# input\_data = {  
# "x0": -2,  
# "x\_end": 2.5,  
# "h": [1, 0.25]  
# }  
  
print(f"Отрезок интегрирования: [{input\_data['x0']}, {input\_data['x\_end']}]\n")  
  
length = abs(input\_data['x0'] - input\_data['x\_end'])  
if length % input\_data['h'][0] != 0 or length % input\_data['h'][1] != 0:  
 print("Отрезок не делится на шаги равномерно")  
 exit(1)  
  
results = {}  
for step in input\_data["h"]:  
 print(f"Шаг h = {step}")  
 try:  
 rect = rectangles(input\_data["x0"], input\_data["x\_end"], step)  
 trap = trapezoids(input\_data["x0"], input\_data["x\_end"], step)  
 simp = None  
 try:  
 simp = simpson(input\_data["x0"], input\_data["x\_end"], step)  
 except ValueError as ve:  
 simp = None  
  
 results[step] = {  
 "rect": rect,  
 "trap": trap,  
 "simp": simp  
 }  
  
 print(f" Метод прямоугольников: {rect:.10f}")  
 print(f" Метод трапеций : {trap:.10f}")  
 if simp is not None:  
 print(f" Метод Симпсона : {simp:.10f}")  
 else:  
 print(f" Метод Симпсона : Невозможно применить (нечётное число интервалов)")  
 except Exception as e:  
 print(f"Ошибка при шаге h = {step}: {e}")  
 print()  
  
print("Оценка погрешностей по методу Рунге-Ромберга:")  
h\_big, h\_small = input\_data["h"]  
k = h\_big / h\_small  
print(f" Для h = {h\_big} и h = {h\_small}")  
  
try:  
 eps\_rect = runge\_romberg(results[h\_big]["rect"], results[h\_small]["rect"], k, p=1)  
 print(f" Метод прямоугольников: ±{eps\_rect:.10f}")  
except:  
 print(" Метод прямоугольников: ошибка оценки")  
  
try:  
 eps\_trap = runge\_romberg(results[h\_big]["trap"], results[h\_small]["trap"], k, p=2)  
 print(f" Метод трапеций : ±{eps\_trap:.10f}")  
except:  
 print(" Метод трапеций : ошибка оценки")  
  
try:  
 if results[h\_big]["simp"] is not None and results[h\_small]["simp"] is not None:  
 eps\_simp = runge\_romberg(results[h\_big]["simp"], results[h\_small]["simp"], k, p=4)  
 print(f" Метод Симпсона : ±{eps\_simp:.10f}")  
 else:  
 print(" Метод Симпсона : нельзя применить (одно из значений отсутствует)")  
except:  
 print(" Метод Симпсона : ошибка оценки")

# **Выводы**

В ходе данной лабораторной работы были освоены и программно реализованы фундаментальные численные методы. Изучены методы интерполяции функций с помощью многочленов Лагранжа и Ньютона, которые показали идентичные результаты для одних и тех же узлов, при этом точность аппроксимации зависела от их расположения и количества. Кубическая сплайн-интерполяция продемонстрировала построение гладкой кривой, точно проходящей через заданные точки. Метод наименьших квадратов позволил аппроксимировать данные полиномами различной степени, минимизируя сумму квадратов ошибок, что полезно для анализа экспериментальных данных. Были реализованы алгоритмы численного дифференцирования для нахождения первой и второй производных на основе таблично заданной функции. Также были применены методы численного интегрирования: прямоугольников, трапеций и Симпсона. Для оценки погрешности и уточнения результатов интегрирования успешно использовался метод Рунге-Ромберга. Работа позволила на практике оценить применимость и эффективность различных численных алгоритмов.