МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информатики и прикладной математики

Кафедра прикладной математики и экономико-математических методов

КУРСОВАЯ РАБОТА

по дисциплине:

«Численные методы»

Тема: «QR-алгоритм со сдвигом»

Направление: 01.03.02 Прикладная математика и информатика

Студент: Бронников Егор Игоревич

Группа: ПМ1901 Подпись:_____

Проверил: Хазанов Владимир Борисович

Должность: д. ф-м. н., профессор

Оценка:_____ Дата:_____

Подпись:_____

Санкт-Петербург

2021

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Данная курсовая работа содержит описание методов решения некоторых задач численного анализа, их программную реализацию, решение тестовых задач и анализ полученных результатов.

Цель работы:

- Изучить и продемонстрировать принцип работы определённых численных методов алгебры и анализа на языках программирования Wolfram Mathematica и Python.
- Изучить и сделать выводы о целесообразности использования QR-алгоритма со сдвигом для решения проблемы собственных значений для матриц.

Задачи:

- Определить необходимую теоретическую основу QR-алогритма со сдвигом и других методов.
- Реализовать QR-алгоритм и прочие методы на языках программирования Wolfram Mathematica и Python.
- Проверить корректность работы алгоритмов.

1. МЕТОД ГАУССА ДЛЯ РЕШЕНИЯ МАТРИЧНОГО УРАВНЕНИЯ С ЧАСТИЧНЫМ ВЫБОРОМ ВЕДУЩЕГО ЭЛЕМЕНТА

1.1 Описание метода

На первом шаге прямого хода метода Гаусса выбирается максимальный по модулю элемент в первом столбце. Этот элемент является ведущим. Если он равен нулю, то $det(A)=0$. Если ведущий элемент не является элементов a_{11}, то перестановкой строк его следует поместить на место a_{11}. При этом соответственно переставляются строки матрицы F. Затем применяются формулы метода Гаусса.

На $i-ом$ шаге прямого хода метода Гаусса непреобразованный столбец является частью столбца i, начиная с элемента a_{ii}, то $a_{ii}, a_{i+1,i}, \ldots, a_{ni}$. Необходимо найти максимальный по модулю элемент в непреобразованном столбце. Этот элемент является ведущим. Если он равен нулю, то $det(A)=0$.
Если ведущий элемент не является элементов a_{ii}, то перестановкой строк его следует поместить на место a_{ii}. При этом соответственно производится перестановка строк матрицы F. Затем следует применить формулы метода Гаусса.

После $(n-1)-го$ шага получаем верхнюю треугольную матрицуRи преобразованную матрицу G в правой части. Выполняем обратный ход.

Метод Гаусса с частичным выбором ведущего элемента позволяет получить точное решение, а для вырожденных матриц — сообщение о том, что матрица является вырожденной.

1.2 Необходимые формулы

Этапы метода Гаусса для решения матричного уравнения. (см. формулу 1)

$$(A|F)\Rightarrow(R|G)\Rightarrow(I|X), где |A|\neq 0 \qquad (1)$$

$$A-n\times n,\ F-n\times l,\ X-n\times l$$

Прямой ход метода Гаусса. (см. формулу 2)

$$R_{k^*}=a_{kk}^{(k-1)^{-1}}A_{k^*}^{(k-1)};\ G_{k^*}=a_{kk}^{(k-1)^{-1}}F_{k^*}^{(k-1)},\ k=1,2,\dots,n \tag{2}$$

$$A_{i^*}^{(k)}=A_{i^*}^{(k-1)}-a_{ik}^{(k-1)}R_{k^*};\ F_{i^*}^{(k)}=F_{i^*}^{(k-1)}-a_{ik}^{(k-1)}G_{k^*},\ i=k+1,\dots,n$$

Обратный ход метода Гаусса. (см. формулу 3)

$$X_{n^*}=G_{n^*} \tag{3}$$

$$X_{k^*}=G_{k^*}-\sum_{j=k+1}^{n}r_{kj}X_{j^*},\ k=n-1,\dots,1$$

1.3 Программная реализация

Данная реализация в ходе алгоритма осуществляет проверку корректности СЛАУ, подаваемой на вход программе. Если СЛАУ корректа, то следующими шагами выполняется каждое из описанных действий пункта 1.1 с последующим выведением шагов, показывающих выполняемые действия по преобразованию матрицы.

Импорт модулей

```
1 import numpy as np         # для работы с матрицами и веторами
2 from typing import Union   # для работы с типизацией
3 import warnings            # для работы с ошибками
4 import sympy as sp         # для красивого вывода промежуточных результатов
5 from IPython.display import Markdown, display  # для красивого вывода текста
```

Рисунок 1 — Импортирование необходимых модулей

Алгоритм

```python
def gaussian_elimination(A_arg: np.matrix, f_arg: Union[np.matrix, np.array]) -> Union[np.matrix, np.array]:
    A, f = np.copy(A_arg), np.copy(f_arg)  # копируем аргументы, чтобы их не 'пачкать'
    display(Markdown('<text style=font-weight:bold;font-size:16px;font-family:serif>Исходные данные<text>'),
            sp.BlockMatrix([sp.Matrix(A.round(decimals=10)), sp.Matrix(f.round(decimals=10))]))
    for i in range(len(A)):
        column = np.abs(A[i:, i])      # берём i-ую колонку по модулю
        leading_elem = np.max(column)  # методом частичного выбора находим ведущий элемент
        if leading_elem == 0.:  # проверяем определитель (if ведущий элемент == 0, то det(A) = 0 => решений нет)
            warnings.warn("Определитель равен 0")  # печатаем ошибку
            return  # заканчиваем выполнение программы
        if np.where(column == leading_elem)[0][0] != 0:  # нужно ли нам менять строки (?)
            pos_max = column.argmax() + i      # узнаём номер строки ведущего элемента
            A[[i, pos_max]] = A[[pos_max, i]]  # меняем строки местами в матрице A
            f[[i, pos_max]] = f[[pos_max, i]]  # меняем строки местами в матрице f
        for j in range(i+1, len(A)):   # делаем верхний треугольник
            coef = -(A[j, i]/A[i, i])  # считаем коэффициент
            A[j] = coef * A[i] + A[j]  # домножаем `i` строку и прибавляем `j`
            f[j] = coef * f[i] + f[j]
        display(Markdown(f'<text style=font-weight:bold;font-size:16px;font-family:serif>{i+1} итерация<text>'),
                sp.BlockMatrix([sp.Matrix(A.round(decimals=10)), sp.Matrix(f.round(decimals=10))]))  # выводим промежуточный результат
    n = f.shape[0]  # размерность нашего ответа
    X = np.zeros(shape=f.shape)   # заполняем наше будущее решение нулями
    X[n-1] = f[n-1]/A[n-1, n-1]   # решаем последнее уравнение
    for i in range(n-2, -1, -1):  # рассчитывает значения начиная с конца
        sum_elem = sum(A[i, j] * X[j] for j in range(i+1, n))  # для известных `x` суммируем коэффициенты
        X[i] = (f[i] - sum_elem)/A[i, i]  # находим `x`
    display(Markdown('<text style=font-weight:bold;font-size:16px;font-family:serif>Ответ<text>'),
            sp.Matrix(X.round(decimals=10)))  # выводим ответ
    return X  # возвращаем ответ для проверки результата
```

Рисунок 2 — Реализация метода Гаусса с частичным выбором ведущего элемента

1.4 Анализ результатов

Входные и тестовые данные представлены на рисунках. (см. рисунки 3, 4)

Входные данные

```python
A = np.matrix([[1.00, 0.17, -0.25, 0.54],
               [0.47, 1.00, 0.67, -0.32],
               [-0.11, 0.35, 1.00, -0.74],
               [0.55, 0.43, 0.36, 1.00]],
              dtype=np.dtype(np.float64))
```

```python
f = np.array([0.3, 0.5, 0.7, 0.9],
             dtype=np.dtype(np.float64))
```

Рисунок 3 — Входные данные

Тестовые наборы данных

```python
test_A1 = np.matrix([[ 2. , -1.4,  0. ],
                     [-0.6,  0.4,  1.2],
                     [ 1. , -0.2,  1. ]],
                    dtype=np.dtype(np.float64))

test_f1 = np.array([1.4, 0.8, 1.2],
                   dtype=np.dtype(np.float64))
```

```python
test_A2 = np.matrix([[0.25, 0.5 ],
                     [0.75, 1.  ]],
                     dtype=np.dtype(np.float64))

test_f2 = np.matrix([[0.75, 1.25],
                     [1.25, 2.25]],
                     dtype=np.dtype(np.float64))
```

Рисунок 4 — Тестовые наборы данных

Результат работы программы представлен на рисунках (см. рисунки 5-7)

```python
np.testing.assert_allclose(np.linalg.solve(A, f),
                           gaussian_elimination(A, f))
```

Исходные данные

$$\left[\left[\begin{array}{cccc} 1.0 & 0.17 & -0.25 & 0.54 \\ 0.47 & 1.0 & 0.67 & -0.32 \\ -0.11 & 0.35 & 1.0 & -0.74 \\ 0.55 & 0.43 & 0.36 & 1.0 \end{array}\right] \left[\begin{array}{c} 0.3 \\ 0.5 \\ 0.7 \\ 0.9 \end{array}\right]\right]$$

1 итерация

$$\left[\left[\begin{array}{cccc} 1.0 & 0.17 & -0.25 & 0.54 \\ 0.0 & 0.9201 & 0.7875 & -0.5738 \\ 0.0 & 0.3687 & 0.9725 & -0.6806 \\ 0.0 & 0.3365 & 0.4975 & 0.703 \end{array}\right] \left[\begin{array}{c} 0.3 \\ 0.359 \\ 0.733 \\ 0.735 \end{array}\right]\right]$$

2 итерация

$$\left[\left[\begin{array}{cccc} 1.0 & 0.17 & -0.25 & 0.54 \\ 0.0 & 0.9201 & 0.7875 & -0.5738 \\ 0.0 & 0.0 & 0.6569351157 & -0.4506684056 \\ 0.0 & 0.0 & 0.2094946201 & 0.9128507771 \end{array}\right] \left[\begin{array}{c} 0.3 \\ 0.359 \\ 0.5891424845 \\ 0.6037061189 \end{array}\right]\right]$$

3 итерация

$$\left[\left[\begin{array}{cccc} 1.0 & 0.17 & -0.25 & 0.54 \\ 0.0 & 0.9201 & 0.7875 & -0.5738 \\ 0.0 & 0.0 & 0.6569351157 & -0.4506684056 \\ 0.0 & 0.0 & 0.0 & 1.0565675677 \end{array}\right] \left[\begin{array}{c} 0.3 \\ 0.359 \\ 0.5891424845 \\ 0.4158303637 \end{array}\right]\right]$$

4 итерация

$$\left[\left[\begin{array}{cccc} 1.0 & 0.17 & -0.25 & 0.54 \\ 0.0 & 0.9201 & 0.7875 & -0.5738 \\ 0.0 & 0.0 & 0.6569351157 & -0.4506684056 \\ 0.0 & 0.0 & 0.0 & 1.0565675677 \end{array}\right] \left[\begin{array}{c} 0.3 \\ 0.359 \\ 0.5891424845 \\ 0.4158303637 \end{array}\right]\right]$$

Ответ

$$\left[\begin{array}{c} 0.4408885509 \\ -0.3630309901 \\ 1.1667983323 \\ 0.3935672231 \end{array}\right]$$

Рисунок 5 — Проверка работы программы на примере №1

```
np.testing.assert_allclose(np.linalg.solve(test_A1, test_f1),
                           gaussian_elimination(test_A1, test_f1))
```

Исходные данные

$$\left[\begin{bmatrix} 2.0 & -1.4 & 0.0 \\ -0.6 & 0.4 & 1.2 \\ 1.0 & -0.2 & 1.0 \end{bmatrix} \quad \begin{bmatrix} 1.4 \\ 0.8 \\ 1.2 \end{bmatrix}\right]$$

1 итерация

$$\left[\begin{bmatrix} 2.0 & -1.4 & 0.0 \\ 0.0 & -0.02 & 1.2 \\ 0.0 & 0.5 & 1.0 \end{bmatrix} \quad \begin{bmatrix} 1.4 \\ 1.22 \\ 0.5 \end{bmatrix}\right]$$

2 итерация

$$\left[\begin{bmatrix} 2.0 & -1.4 & 0.0 \\ 0.0 & 0.5 & 1.0 \\ 0.0 & 0.0 & 1.24 \end{bmatrix} \quad \begin{bmatrix} 1.4 \\ 0.5 \\ 1.24 \end{bmatrix}\right]$$

3 итерация

$$\left[\begin{bmatrix} 2.0 & -1.4 & 0.0 \\ 0.0 & 0.5 & 1.0 \\ 0.0 & 0.0 & 1.24 \end{bmatrix} \quad \begin{bmatrix} 1.4 \\ 0.5 \\ 1.24 \end{bmatrix}\right]$$

Ответ

$$\begin{bmatrix} 0.0 \\ -1.0 \\ 1.0 \end{bmatrix}$$

Рисунок 6 — Проверка работы программы на примере №2

```
np.testing.assert_allclose(np.linalg.solve(test_A2, test_f2),
                           gaussian_elimination(test_A2, test_f2))
```

Исходные данные

$$\left[\begin{bmatrix} 0.25 & 0.5 \\ 0.75 & 1.0 \end{bmatrix} \quad \begin{bmatrix} 0.75 & 1.25 \\ 1.25 & 2.25 \end{bmatrix}\right]$$

1 итерация

$$\left[\begin{bmatrix} 0.75 & 1.0 \\ 0.0 & 0.1666666667 \end{bmatrix} \quad \begin{bmatrix} 1.25 & 2.25 \\ 0.3333333333 & 0.5 \end{bmatrix}\right]$$

2 итерация

$$\left[\begin{bmatrix} 0.75 & 1.0 \\ 0.0 & 0.1666666667 \end{bmatrix} \quad \begin{bmatrix} 1.25 & 2.25 \\ 0.3333333333 & 0.5 \end{bmatrix}\right]$$

Ответ

$$\begin{bmatrix} -1.0 & -1.0 \\ 2.0 & 3.0 \end{bmatrix}$$

Рисунок 7 — Проверка работы программы на примере №3

2. ГРАДИЕНТНЫЕ МЕТОДЫ. МЕТОД НАИСКОРЕЙШЕГО СПУСКА

2.1 Описание метода

Пусть A — симметричная и положительно определённая матрица. Тогда можно ввести функцию ошибки $F(x)$ (см. формулу 4), которая принимает наименьшее нулевое значение на решении системы, но поскольку неизвестна сама функция ошибки, то нельзя выяснить её характеристики.

$$F(x)=(Ay,y)\geq 0 \tag{4}$$

Введём функционал $H(x)$, который отличается от функции ошибки на константу (см. формулу 5). Таким образом получается, что минимум функционала $H(x)$ достигается в той же точке, что и минимум функционала $F(x)$.

$$H(\boldsymbol{x})=(\boldsymbol{Ax},\boldsymbol{x})-2(\boldsymbol{f},\boldsymbol{x})=F(\boldsymbol{x})-(\boldsymbol{f},\boldsymbol{x}_*)\Rightarrow min\,H(\boldsymbol{x})=H(\boldsymbol{x}_*) \tag{5}$$

Имея начальное приближение можно минимизировать функционал $H(x)$ двигаясь от текущей точки x к следующему приближению. Таким образом, находим градиент функции $H(x)$, а как следствие, находим следующее приближение (см. формулу 6). В итоге получается, что значение градиента функции $H(x)$ равно $-2r$, где r— вектор невязки.

$$gradH(x)=-2(Ax-f)=-2r \tag{6}$$

Минимизация осуществляется переходом к новому приближению x' когда к текущему приближению добавляеться вектор антиградиента умноженный на параметр α и минимизация будет осуществляться по этому параметру. (см. формулу 7)

$$x'=x+\alpha r \tag{7}$$

Далее нужно найти значение производной по α и приравнять это выражение к нулю. Тогда получим выражение для нахождения следующего параметра α. (см. формулу 8)

$$\frac{d}{d\alpha} H(x') = 0 \Rightarrow \alpha = \frac{(r, r)}{(Ar, r)} \tag{8}$$

В итоге, мы получаем итерационную формулу для нахождения следующего значения x. (см. формулу 9)

$$x_{k+1} = x_k + \alpha_k r_k \tag{9}$$

$$r_k = f - Ax_k$$

$$\alpha_k = \frac{(r_k, r_k)}{(Ar_k, r_k)}$$

Стоит отметить, что на каждом шаге нам требуется вычисление вектора невязки, умножение матрица A на вектор невязки и вычисление двух скалярных произведений.

2.2 Программная реализация

Данная реализация проверяет СЛАУ в соответствии с необходимыми требованиями, а именно проверят матрицу коэффициентов на положительно определённость и симметричность. Если СЛАУ корректа, то следующими шагами выполняется каждое из описанных действий пункта (2.1) с последующим выведением шагов, показывающих выполняемые действия по преобразованию матрицы.

Импорт модулей

```
import numpy as np         # для работы с матрицами и веторами
import warnings            # для работы с ошибками
import sympy as sp         # для красивого вывода промежуточных результатов
from IPython.display import Markdown, display  # для красивого вывода текста
```

Рисунок 8 — Импортирование модулей

Проверяет положительно определённая ли матрица

```
def is_positive_definite(A: np.matrix) -> bool:
    return np.all(np.linalg.eigvals(A) > 0)  # считаем собственные числа и проверяет, что все они больше 0
```

```
is_positive_definite(A)
```

```
True
```

Проверяет симметричная ли матрица

```
def is_symmetric(A: np.matrix) -> bool:
    return np.allclose(A, A.T)  # сравниваем обычную матрицу и транспонированную
                                # если они совпадают, то матрица симметричная
```

```
is_symmetric(A)
```

```
True
```

Рисунок 9 — Проверка матрицы коэффициентов

Алгоритм

```
def steepest_descent_method(A_arg: np.matrix, f_arg: np.array, K_max: int) -> np.array:
    A, f = np.copy(A_arg), np.copy(f_arg)  # копируем аргументы, чтобы их не 'пачкать'
    display(Markdown('<text style=font-weight:bold;font-size:16px;font-family:serif>Исходные данные<text>'),
        sp.BlockMatrix([sp.Matrix(A.round(decimals=10)), sp.Matrix(f.round(decimals=10))]))
    if not is_positive_definite(A):  # проверяем положительна ли определена матрица `A`
        warnings.warn("Матрица не является положительно определённой")  # печатаем ошибку
        return
    elif not is_symmetric(A):  # проверям симметричная ли матрица `A`
        warnings.warn("Матрица не является симметричной")  # печатаем ошибку
        return
    elif K_max < 0:  # проверяем `K_max`
        warnings.warn("Количество итераций не может быть отрицательным числом")  # печатаем ошибку
        return
    x = np.zeros(f.shape, dtype=np.dtype(np.float64))  # начальное приближение
    for k in range(K_max):  # итерируемся до `K_max`
        display(Markdown(f'<text style=font-weight:bold;font-size:16px;font-family:serif>{k+1} итерация<text>'),
                sp.Matrix(x.round(decimals=10)))
        r = np.squeeze(np.asarray(f - np.matmul(A, x)))  # находим вектор невязки
        alpha = (np.dot(r, r)/np.dot(np.matmul(A, r), r)).item(0)  # находим alpha
        x = x + alpha*r  # находим `k`-ое приближённое решение
    display(Markdown(f'<text style=font-weight:bold;font-size:16px;font-family:serif>Ответ<text>'),
            sp.Matrix(x.round(decimals=10)))
    return x
```

Рисунок 10 — Реализация метода наискорейшего спуска

2.3 Анализ результатов

Входные и тестовые данные представлены на рисунках. (см. рисунки 10, 11)

Входные данные

```
A = np.matrix([[4.33, -1.12, -1.08, 1.14],
               [-1.12, 4.33, 0.24, -1.22],
               [-1.08, 0.24, 7.21, -3.22],
               [1.14, -1.22, -3.22, 5.43]],
               dtype=np.dtype(np.float64))
```

```
f = np.array([0.3, 0.5, 0.7, 0.9],
             dtype=np.dtype(np.float64))
```

Рисунок 11 — Входные данные

Тестовые наборы данных

```
test_A1_Err = np.matrix([[1.00, 0.17, -0.25, 0.54],
                         [0.47, 1.00, 0.67, -0.32],
                         [-0.11, 0.35, 1.00, -0.74],
                         [0.55, 0.43, 0.36, 1.00]],
                         dtype=np.dtype(np.float64))
```

```
test_A2_Err = np.matrix([[-11.00, 6.00],
                         [6.00, -11.00]],
                         dtype=np.dtype(np.float64))
test_f2_Err = np.array([0.3, 0.5],
             dtype=np.dtype(np.float64))
```

Рисунок 12 — Тестовые наборы данных

Результат работы программы представлен на рисунках (см. рисунки 13-18)

Тесты

```
x = steepest_descent_method(A, f, 10)
```

Исходные данные

$$\left[\begin{bmatrix} 4.33 & -1.12 & -1.08 & 1.14 \\ -1.12 & 4.33 & 0.24 & -1.22 \\ -1.08 & 0.24 & 7.21 & -3.22 \\ 1.14 & -1.22 & -3.22 & 5.43 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.5 \\ 0.7 \\ 0.9 \end{bmatrix}\right]$$

1 итерация

$$\begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

2 итерация

$$\begin{bmatrix} 0.1159775588 \\ 0.1932959314 \\ 0.2706143039 \\ 0.3479326764 \end{bmatrix}$$

3 итерация

$$\begin{bmatrix} 0.0985107399 \\ 0.2228601566 \\ 0.2605456268 \\ 0.3451615732 \end{bmatrix}$$

Рисунок 13 — Пример работы программы на примере №1 (часть 1)

4 итерация

$$\begin{bmatrix} 0.099772049 \\ 0.2233106841 \\ 0.2589100115 \\ 0.347960791 \end{bmatrix}$$

5 итерация

$$\begin{bmatrix} 0.1000197017 \\ 0.2250170915 \\ 0.2607752527 \\ 0.34866444 \end{bmatrix}$$

6 итерация

$$\begin{bmatrix} 0.1003772689 \\ 0.2250728728 \\ 0.260373851 \\ 0.3494673588 \end{bmatrix}$$

7 итерация

$$\begin{bmatrix} 0.1004387563 \\ 0.2254805811 \\ 0.2609386966 \\ 0.3496940339 \end{bmatrix}$$

8 итерация

$$\begin{bmatrix} 0.1005313895 \\ 0.2254993065 \\ 0.2608236673 \\ 0.3499218645 \end{bmatrix}$$

P

Рисунок 14 — Пример работы программы на примере №1 (часть 2)

9 итерация

$$\begin{bmatrix} 0.1005401597 \\ 0.2256156715 \\ 0.2609812639 \\ 0.3499883034 \end{bmatrix}$$

10 итерация

$$\begin{bmatrix} 0.1005660379 \\ 0.2256202222 \\ 0.2609492325 \\ 0.3500528971 \end{bmatrix}$$

Ответ

$$\begin{bmatrix} 0.1005680733 \\ 0.2256523012 \\ 0.2609940632 \\ 0.3500720528 \end{bmatrix}$$

Рисунок 15 — Пример работы программы на примере №1 (часть 3)

```
steepest_descent_method(test_A1_Err, f, 10)
```

Исходные данные

$$\left[\left[\begin{array}{cccc} 1.0 & 0.17 & -0.25 & 0.54 \\ 0.47 & 1.0 & 0.67 & -0.32 \\ -0.11 & 0.35 & 1.0 & -0.74 \\ 0.55 & 0.43 & 0.36 & 1.0 \end{array}\right] \left[\begin{array}{c} 0.3 \\ 0.5 \\ 0.7 \\ 0.9 \end{array}\right]\right]$$

```
<ipython-input-119-f9ccc6ac74a8>:9: UserWarning: Матрица не является симметричной
  warnings.warn("Матрица не является симметричной")  # печатаем ошибку
```

Рисунок 16 — Пример работы программы на неправильных входных данных

```
steepest_descent_method(test_A2_Err, test_f2_Err, 10)
```

Исходные данные

$$\left[\left[\begin{array}{cc} -11.0 & 6.0 \\ 6.0 & -11.0 \end{array}\right] \left[\begin{array}{c} 0.3 \\ 0.5 \end{array}\right]\right]$$

```
<ipython-input-119-f9ccc6ac74a8>:6: UserWarning: Матрица не является положительно определённой
  warnings.warn("Матрица не является положительно определённой")  # печатаем ошибку
```

Рисунок 17 — Пример работы программы на неправильных входных данных

Оценка точности полученного решения

```
r = f - np.matmul(A, x)
display(Markdown(f'<text style=font-weight:bold;font-size:16px;font-family:serif>Вектор невязки<text>'),
        sp.Matrix(r.T))
```

Вектор невязки

$$\left[\begin{array}{c} 6.22681101437594 \cdot 10^{-5} \\ 1.11072297926951 \cdot 10^{-5} \\ -7.82185327168339 \cdot 10^{-5} \\ 0.000157840633184247 \end{array}\right]$$

Рисунок 18 — Оценка точности полученного результата

3. МЕТОД СЕКУЩИХ ДЛЯ СИСТЕМЫ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

3.1 Описание метода

Задача заключается в нахождении корней нелинейной системы уравнений: (см. формулу 10)

$$f(x)=0 \tag{10}$$

Стоит отметить, что метод секущих является дискретной модификацией метода Ньютона. Сначала надо найти матрицу Якоби — матрицы $f(x)$. Она может быть представлена аппроксимацией, а именно: (см. формулу 11)

$$f'_k = f'(x_k) \approx H_k^{-1}\Gamma_k \Leftrightarrow f_k^{'-1} \approx \Gamma_k^{-1} H_k \tag{11}$$

Где Γ_k — матрица, столбцы которой являются приращением значений левых частей нелинейного уравнения, а H_k — матрица, столбцы которой показывают приращение векторного аргумента.

Все частные производные функции $f_i(x_i)$ по x_j находятся как отношения приращения функции по компонентам с номерам j умноженное на h и на h делённое. Таким образом, h общее для номера нелинейного уравнения и для номера неизвестного. (см. формулу 12)

$$\frac{df_i(x_k)}{dx_j} \approx \frac{f_i(x_k + h e_j) - f_i(x_k)}{h} \tag{12}$$

Тогда матрица H является диагональной. (см. формулу 13)

$$H_k = hI \tag{13}$$

Матрицу Γ можно представить следующим образом. (см. формулу 14)

$$\gamma_{ij}^k = f_i(x_k - h e_j) - f_i(x_k) \tag{14}$$

$$\Gamma_k = \{\gamma_{ij}^{(k)}\}_{i,j}^{s}$$

После нахождения матрицы Якоби можно найти следующее приближение по формуле. (см. формулу 15)

$$x_{k+1} = x_k - f_k^{'-1} f_k, k = 1 \dots K_{max} \quad (15)$$

В данном случае используется два критерия остановки итерационного процесса:

1. K_{max}— критерий прекращения итерационного процесса по числу итераций
2. δ— критерий прекращения итерационного процесса по малости двух соседних приближений

3.2 Программная реализация

На рисунках представлена программная реализация метода секущих для системы нелинейных уравнений. (см. рисунок 19)

Алгоритм

```
Clear[secantMethod]
secantMethod[F_Symbol, x0_List, Kmax_Integer, δ_Real, h_Real] := Module[
  {
   x = {x0},
   k = 1,
   dRes = δ,
   H, G, fDInv, e, n
  },
  n = Length[F[x[[1]]]]; (* количество уравнений в системе *)
  H = h * IdentityMatrix[n]; (* матрица H *)
  While[k ≤ Kmax ∧ δ ≤ dRes, (* критерии остановки: число итераций и малось соседних приближений*)
   G = Table[ (* рассчитываем матрицу Г *)
     e = ConstantArray[0, n];
     e[[j]] = 1;
     F[x[[k]] + h * e][[i]] - F[x[[k]]][[i]], (* считаем γij *)
     {i, n}, {j, n}];
   fDInv = Inverse[G].H; (* считаем fk'-1*)
   AppendTo[x, x[[k]] - fDInv.F[x[[k]]]]; (* считаем xk+1 *)
   dRes = Norm[x[[k + 1]] - x[[k]]]; (* считаем значение соседних приближений *)
   k += 1
  ];
  x[[-1]] (* выводим результат *)
 ]
```

Рисунок 19 — Реализация метода секущих для системы нелинейных уравнений

3.3 Анализ результатов

Входные данные представлены в формуле (см. формулу 16) и на рисунке. (см. рисунки 20)

$$f(x)=\begin{pmatrix} x_1^2-x_2^2-1 \\ x_1 x_2^3-x_2-1 \end{pmatrix} \tag{16}$$

$$x_0=\begin{pmatrix} 1.8 \\ -0.3 \end{pmatrix}$$

$$K_{max}=10$$

$$\delta=10^{-5}$$

$$h=10^{-10}$$

Входные данные:

```
h = 10.^-10;
x0 = {1.5, 1.5};
δ = 10.^-5;
Kmax = 10;

Clear[f1]
f1[x_] := x[[1]]^2 - x[[2]]^2 - 1

Clear[f2]
f2[x_] := x[[1]] x[[2]]^3 - x[[2]] - 1

Clear[F]
F[x_] := {f1[x], f2[x]}
```

Рисунок 20 — Входные данные

Результаты

```
secantMethod[F, x0, Kmax, δ, h]
{1.50284, 1.12185}
```

Результат встроенной функции

```
NSolve[{x1^2 - x2^2 - 1 == 0, x1 x2^3 - x2 - 1 == 0}, {x1, x2}, Reals]
{{x1 → 1.50284, x2 → 1.12185}, {x1 → -1.19726, x2 → -0.658357}}
```

Рисунок 21 — Пример работы программы

Оценка точности

```
r = F[x]
```

$$\{-3.77476 \times 10^{-15},\ 3.44169 \times 10^{-14}\}$$

Рисунок 22 — Оценка точности полученного решения

4. МЕТОДЫ РЕШЕНИЯ ЧАСТИЧНОЙ ПРОБЛЕМЫ СОБСТВЕННЫХ ЗНАЧЕНИЙ. QR-АЛГОРИТМ СО СДВИГОМ

4.1 Описание метода

QR-алгоритм — это численный метод в линейной алгебре, предназначенный для решения полной проблемы собственных значений, то есть отыскивания всех собственных чисел матрицы. Он был разработан в конце 1950-х годов независимо В. Н. Кублановской и Дж. Фрэнсисом. Открытию QR-алгоритма предшествовал LR-алгоритм, который использовал LU-разложение вместо QR-разложения. В настоящее время LR-алгоритм используется очень редко ввиду своей меньшей эффективности, однако он был важным шагом на пути к открытию QR-алгоритма. В настоящее время под QR-алгоритмом понимается не только исходно разработанный В. Н. Кублановской и Дж. Фрэнсисом алгоритм, но и всю совокупность более поздних приёмов его ускорения.

4.1.1 Базовый QR-алгоритм

Суть базового QR-алгоритма заключается в итерационном приведении матрицы A к некоторой унитарно подобной ей матрице A_n при помощи QR-разложения. Матрица A_n является правой верхней треугольной матрицей, а значит ее диагональ содержит собственные значения. В силу подобия матриц A и A_n их наборы собственных значений совпадают. Таким образом задача поиска собственных значений матрицы сводится к задаче выведения матрицы A_n и поиска собственных значений для неё.

Первым шагом является нахождение QR-разложения исходной матрицы A . Это можно сделать несколькими способами:

- Методом ортогонализации Грамма-Шмидта
- Методом Гивенса
- Методом Хаусхолдера

Отражение Хаусхолдера (или преобразование Хаусхолдера) — это преобразование, которое берет вектор и отражает его относительно некоторой плоскости или гиперплоскости. Использование преобразований Хаусхолдера по своей сути является наиболее простым из численно устойчивых алгоритмов QR-разложения из-за использования отражений в качестве механизма для получения нулей в матрице R. Однако алгоритм отражения Хаусхолдера требует большой полосы пропускания и не поддается распараллеливанию, поскольку каждое отражение, которое создает новый нулевой элемент, полностью изменяет матрицы Q и R.

QR-разложения можно вычислить с помощью серии вращений Гивенса. Каждое вращение обнуляет элемент в поддиагонали матрицы, образуя матрицу R. Объединение всех вращений Гивенса образует ортогональную матрицу Q. QR-разложение с помощью вращений Гивенса является наиболее сложным для реализации. Однако он имеет значительное преимущество в том, что каждый новый нулевой элемент a_{ij} влияет только на строку с обнуляемым элементом i и строку выше j. Это делает алгоритм вращения Гивенса более эффективным и распараллеливаемым, чем метод отражения Хаусхолдера.

В данном случае, стоит рассмотреть более подробно QR-разложение методом ортогонализации Грамма-Шмидта, потому что именно этот метод будет использоваться в реализации алгоритма. Представим матричное произведение Q и R в виде: (см. формулу 17)

$$A = \sum_{i=1}^{n} cQ_i \, rR_i^T \tag{17}$$

Заметим, что первые $i - 1$ столбец нашей матрицы равны нулю, потому что R — верхнетреугольная матрица: (см. рисунок 23)

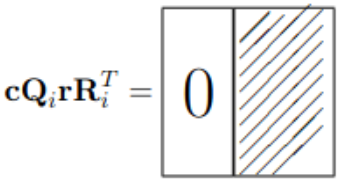

Рисунок 23 — Первые $i-1$ столбцы нашей матрицы

Определим матрицы, которые имеют ту же форму, что и на рисунке (23) (первые $k-1$ столбцов равны нулю): (см. формулу 18)

$$A^{(k)}=A-\sum_{i=1}^{k-1} cQ_i rR_i^T,\ k=1,\dots,n+1 \tag{18}$$

Что эквивалентно данной формуле: (смотреть формулу 19)

$$A^{(k)}=\sum_{i=k}^{n} cQ_i rR_i^T \tag{19}$$

Ясно, что если продолжить процесс, то мы получим следующие формулы (см. формулы 20)

$$A^{(1)}=A \tag{20}$$
$$A^{(k+1)}=A(k)-cQ_k rR_k^T,\ k=1,\dots,n$$
$$A^{(n+1)}=0$$

Воспользуясь формулами (19) и (20) мы получим, что:

$$A^{(k)}e_k=cA_k^{(k)}=\sum_{i=k}^{n} cQ_i\left(rR_i^T e_k\right)=cQ_k r_{kk} \tag{21}$$

Следовательно: (см. формулу 22)

$$r_{kk}=\|cA_k^{(k)}\| \tag{22}$$
$$cQ_k=\frac{cA^{(k)}}{r_{kk}}$$

Тогда из следующих рассуждений можно получить шаги, необходимые для расчёта QR-разложения по модифицированному методу ортогонализации Грамма-Шмидта. (см. формулы 23)

$$s=0 \tag{23}$$

$$s=s+a_{jk}^2,\ j=1,\dots,n$$

$$r_{kk}=\sqrt{s}$$

$$q_{jk}=\frac{a_{jk}}{r_{kk}},\ j=1,\dots,n$$

$$s=0;\ s=s+a_{ji}\ *\ q_{jk},\ j=1,\dots,n;\ r_{ki}=s;\ a_{ji}=a_{ji}-r_{ki}\ *\ q_{jk},\ i=k+1,\dots,n$$

$$k=1,\dots,n$$

Итак, первым шаг сделан, найдено QR-разложение исходной матрицы. (см. формулу 24)

$$A_k=Q_k R_k \tag{24}$$

На следующем шаге мы должны найти новое приближение матрицы A, а именно перемножить в обратном порядке матрицы R и Q. (см. формулу 25)

$$A_{k+1}=R_k Q_k \tag{25}$$

Поскольку $A_{k+1}=R_k Q_k=Q_k^* A_k Q_k$, то матрицы A_{k+1} и A_kунитарно подобны, для любого k. Поэтому матрицы $A_1, A_2,\dots$ унитарно подобны исходной матрице A и имеют те же собственные значения.

Сходимость у данного метода является линейной.

4.1.2 QR-алгоритм со сдвигом

QR-алгоритм со сдвигами позволяет сократить количество итераций, необходимых для сходимости. Пусть есть матрица A_k, тогда процесс перехода к матрице A_{k+1} выглядит следующим образом. (см. формулу 26) Теперь на каждом шаге подбирается число t_k.

$$A_k - t_k = Q_k R_k \tag{26}$$

$$A_{k+1} = R_k Q_k + t_k I$$

$$t_{k+1} = a_{nn}^{(k)}$$

$$k = 1, \ldots, K_{max}$$

При этом сохраняется свойство подобия матриц.

Сходимость данного способа — квадратичная.

4.2 Программная реализация

На рисунках представлена программная реализация QR-алгоритма со сдвигом. (см. рисунок 24) Критерием остановки итерационного процесса является количество итераций и близость двух приближений.

Импорт модулей

```
import numpy as np          # для работы с матрицами и веторами
import warnings             # для работы с ошибками
import sympy as sp          # для красивого вывода промежуточных результатов
from IPython.display import Markdown, display  # для красивого вывода текста
```

Рисунок 24 — Импортирование модулей

Модифицированный алгоритм Грама-Шмидта для нахождения QR-разложения

```
def qr_mod_gram_schmidt(A_arg: np.matrix):
    A = np.copy(A_arg)
    n = A.shape[0]
    R, Q = np.zeros(A.shape), np.zeros(A.shape)
    for k in range(n):
        s = 0
        for j in range(n):
            s += A[j, k]**2
            R[k, k] = np.sqrt(s)
        for j in range(n): Q[j, k] = A[j, k]/R[k, k]
        for i in range(k, n):
            s = 0
            for j in range(n):
                s += A[j, i] * Q[j, k]
                R[k, i] = s
            for j in range(n): A[j, i] = A[j, i] - R[k, i] * Q[j, k]
    return np.asmatrix(Q), np.asmatrix(R)
```

Рисунок 25 — Реализация модифицированного алгоритма Грама-Шмидта

QR-алгоритм

```
def qr_mod_algorithm(A: np.matrix, Kmax: int, delta: float) -> np.array:
    if Kmax < 1:
        warnings.warn("Количество итераций должно быть положительным числом")
        return
    Ak = np.copy(A)
    t = 0
    I = np.identity(A.shape[0])
    d = delta
    eigvals = []
    k = 0
    while k < Kmax and d >= delta:
        Q, R = qr_mod_gram_schmidt(Ak - t * I)
        Ak = np.matmul(R,Q) + t * I if k else np.matmul(R, Q)
        t = Ak[-1, -1]
        eigvals.append(np.diagonal(Ak))
        d = np.linalg.norm(eigvals[-1] - eigvals[-2]) if k else delta
        k += 1
    display(Markdown(f"""<text style=font-weight:bold;font-size:16px;font-family:serif>
                            Количество итераций, которое потребовалось для нахождения решения: {k}
                        <text>"""))
    return eigvals[-1]
```

Рисунок 26 — Реализация QR-алгоритма со сдвигом

4.3 Анализ результатов

Входные данные представлены на рисунке. (см. рисунок 27)

Входные данные

```
A1 = np.matrix([[4.33, -1.12, -1.08, 1.14],
                [-1.12, 4.33, 0.24, -1.22],
                [-1.08, 0.24, 7.21, -3.22],
                [1.14, -1.22, -3.22, 5.43]],
                dtype=np.dtype(np.float64))
```

```
A2 = np.matrix([[1.00, 0.42, 0.54, 0.66],
                [0.42, 1.00, 0.32, 0.44],
                [0.54, 0.32, 1.00, 0.22],
                [0.66, 0.44, 0.22, 1.00]],
                dtype=np.dtype(np.float64))
```

Рисунок 27 — Входные данные

Результаты

```
real_res = qr_mod_algorithm(A1, 200, 10**-20)
np_res = np.linalg.eigvals(A1)
display(Markdown('<text style=font-weight:bold;font-size:16px;font-family:serif>Полученный ответ<text>'),
            sp.Matrix(real_res.round(decimals=10)))
display(Markdown('<text style=font-weight:bold;font-size:16px;font-family:serif>Встроенная функция<text>'),
            sp.Matrix(np_res.round(decimals=10)))
```

Количество итераций, которое потребовалось для нахождения решения: 19

Полученный ответ

$$\begin{bmatrix} 10.3267786405 \\ 5.1025199601 \\ 3.3389380551 \\ 2.5317633444 \end{bmatrix}$$

Встроенная функция

$$\begin{bmatrix} 10.3267786405 \\ 5.1025199601 \\ 3.3389380551 \\ 2.5317633444 \end{bmatrix}$$

Рисунок 28 — Пример работы программы на примере №1

```
real_res = qr_mod_algorithm(A2, 200, 10**-20)
np_res = np.linalg.eigvals(A2)
display(Markdown('<text style=font-weight:bold;font-size:16px;font-family:serif>Полученный ответ<text>'),
            sp.Matrix(real_res.round(decimals=10)))
display(Markdown('<text style=font-weight:bold;font-size:16px;font-family:serif>Встроенная функция<text>'),
            sp.Matrix(np_res.round(decimals=10)))
```

Количество итераций, которое потребовалось для нахождения решения: 59

Полученный ответ

$$\begin{bmatrix} 2.3227488001 \\ 0.7967066889 \\ 0.6382838028 \\ 0.2422607083 \end{bmatrix}$$

Встроенная функция

$$\begin{bmatrix} 2.3227488001 \\ 0.2422607083 \\ 0.6382838028 \\ 0.7967066889 \end{bmatrix}$$

Рисунок 29 — Пример работы программы на примере №2

5. ИНТЕРПОЛИРОВАНИЕ ФУНКЦИИ ОДНОЙ ИЛИ ДВУХ ПЕРЕМЕННЫХ. ПЕРВАЯ ИНТЕРПОЛЯЦИОННАЯ ФОРМУЛА НЬЮТОНА (В НАЧАЛЕ ТАБЛИЦЫ)

5.1 Описание метода

Первый шаг заключается в том, что нужно построить интерполяционную сетку узлов. (см. формулу 27) Интерполяционная сетка строиться в предположении, что узлов должно быть большое количество, а для достижения требуемой точности необходимо, чтобы узлов было меньше.

$$x_i = x_0 + ih,\ i = 0, \ldots, N \tag{27}$$

Если воспользоваться формулами, выражающими разностные отношения через конечные разности, то получается следующая формула: (см. формулу 28)

$$P_n(x) = y_0 + q\Delta y_0 + \frac{q(q-1)}{2!}\Delta^2 y_0 + \ldots + \frac{q(q-1)\ldots(q-n+1)}{n!}\Delta^n y_0 \tag{28}$$

Где $\Delta^k y_0$ — конечные разности для первого узла (см. формулу 29), а $q = \frac{x - x_0}{h}$, где x_0 — значение функции в первом узле, h — шаг интерполяции.

$$\Delta^k y_0 = \Delta(\Delta y_0) = \Delta^{k-1} y_1 - \Delta^{k-1} y_0 \tag{29}$$

Оценка остаточного члена будет выглядеть следующим образом: (см. формулу 30)

$$R_n(x_0 + th; f) = \frac{t^{\underline{n+1}} h^{n+1}}{(n+1)!} f^{(n+1)}(\xi) \tag{30}$$

5.2 Программная реализация

Для реализации данного метода были написаны три функции. Первая функция — *finite_differences* — составляет таблицу конечных разностей (см. рисунок 30), вторая функция — *newton_function*— выводит на экран интерполяционный полином (см. рисунок 31) и третья функция — *newton_interpolation*— рассчитывает значение полинома в точке *x* (см. рисунок 31).

Импорт модулей

```
import numpy as np                                # для работы с матрицами и веторами
import matplotlib.pyplot as plt                   # для отрисовки графиков
import sympy as sp                                # для красивого вывода математических объектов
from math import factorial                        # для нахождения факториала
import functools                                  # немного функционального программирования
from IPython.display import Markdown, display     # для красивого вывода текста

%matplotlib inline
```

```
plt.style.use('seaborn-poster')  # стиль графиков
```

Рисунок 30 — Импортирование модулей

Находим конечные разности

```
def finite_differences(y: np.array) -> np.matrix:
    ''' finite_differences - составляет таблицу конечных разностей

        Аргументы:
            * y: np.array - список значений `f(x)`

        Возвращает:
            np.matrix - таблица конечных разностей
    '''

    n = len(y)               # длина списка f(x)
    coef = np.zeros([n, n])  # заполняем матрицу конечных разностей нулями
    coef[:, 0] = y           # пусть первая колонка у нас будет f(x)
    for j in range(1, n):    # итерируемся начиная со второй колонки
        for i in range(n-j):
            coef[i][j] = coef[i+1][j-1]-coef[i][j-1]  # по формуле находим все конечные разности
    return coef
```

Рисунок 31 — Программная реализация нахождения конечных разностей

Интерполяция

Вывод полинома

```
def newton_function(x: np.array, y: np.array, *, var: bool = False) -> None:
    ''' newton_function - выводит на экран интерполяционный полином

        Аргументы:
            * x: np.array - список значений `x`
            * y: np.array - список значений `f(x)`
            ** var: bool - определяет от какого аргумента будет полином от `x` или от `q`
                True - `x`
                False - `q`, где `q` = (x - x_0)/h

        Возвращает:
            None - но печатает на экран интерполяционный полином
    '''

    coef = finite_differences(y)                    # таблица конечных разностей
    q = f'(x-{x[0]})/{x[1]-x[0]}' if var else 'q'   # определяем какой аргумент будет у полинома
    arg = 'x' if var else 'q'
    variables = [list(map(lambda x: f'({q} - {x})', np.arange(0, i))) for i in range(len(y))]  # находим аргументы при коэффициентах
    lst = []  # список со слагаемыми
    for i in range(len(variables)):
        if len(variables[i]) > 0:  # если у нас есть аргумент в слагаемом
            lst.append(str(coef[0, i]) + '*' + functools.reduce((lambda x, y: f'{x}*{y}'), variables[i]) + '/' + str(factorial(i)))  # True: рассчитываем по формуле слагаемое
        else:
            lst.append(str(coef[0, i]))  # False: добавляем коэффициент
    expression = functools.reduce((lambda x, y: f'{x} + {y}'), lst)  # получаем наше выражение
    display(Markdown(f"<text style=font-size:18px;font-family:serif>P({arg}) = </text> {sp.latex(sp.simplify(expression), mode='inline')}"))  # выводим его
```

Рисунок 32 — Программная реализация вывода интерполяционного полинома

Процесс интерполяции

```
def newton_interpolation(x: float, x_data: np.array, y: np.array) -> float:
    ''' newton_interpolation - находим значение полинома в точке `x`

        Аргументы:
            * x: float - точка в которой ищется значение полинома
            * x_data: np.array - список исходных значений `x`
            * y: np.array - список исходных значений `f(x)`

        Возвращает:
            float - значение полинома в точке `x`
    '''

    coefs = finite_differences(y)  # таблица конечных разностей
    h = x_data[1] - x_data[0]      # `h` - шаг
    q = (x - x_data[0])/h          # находим `q`
    mul = 1
    s = 0
    for i in range(len(y)):
        mul = 1
        for j in range(i):
            mul *= q-j             # находим значение аргумента в точке `x`
        s += coefs[0, i]*mul/factorial(i)  # находим сумму всех слагаемых
    return s
```

Рисунок 33 — Программная реализация метода Ньютона в начале таблицы

5.3 Анализ результатов

Входные данные представлены в формуле (31) и на рисунке (34). (см. рисунок 34)

$$[1,17]\text{- интервал} \tag{31}$$

$$h=2.5\text{- шаг интерполирования}$$

$$f(x)=\ln(x)+\sqrt{1+x}\text{- функция}$$

Входные данные

```
def f(x):
    return np.log(x) + np.sqrt(1+x)
```

```
x_d = sp.Symbol('x')
display(Markdown(f"<text style=font-size:16px;font-family:serif>f(x) = </text> {sp.latex(sp.log(x_d) + sp.sqrt(1+x_d), mode='inline')}"))
```

f(x) = $\sqrt{x+1} + \log(x)$

```
h = 2.5
x_0 = 1
x_1 = 17
```

```
x = np.arange(x_0, x_1, h)
y = f(x)
display(Markdown(f"<text style=font-size:16px;font-family:serif>x = {list(x)}<br>f(x) = {list(y)} </text>"))
```

x = [1.0, 3.5, 6.0, 8.5, 11.0, 13.5, 16.0]
f(x) = [1.4142135623730951, 3.3740833120550104, 4.437510780292646, 5.222273164980759, 5.861996887936125, 6.410576238376338, 6.895694347857441]

Рисунок 34 — Входные данные

```
a = finite_differences(y)
sp.Matrix(a)
```

1.4142135623731	1.95986974968192	−0.89644228144428	0.617777197894757	−0.484150776077982	0.404418643478801	−0.350897668541171
3.37408331205501	1.06342746823764	−0.278665083549523	0.133626421816775	−0.0797321325991809	0.0535209749376295	0.0
4.43751078029265	0.784762384688113	−0.145038661732747	0.0538942892175944	−0.0262111576615514	0.0	0.0
5.22227316498076	0.639723722955366	−0.0911443725151528	0.0276831315560431	0.0	0.0	0.0
5.86199688793613	0.548579350440213	−0.0634612409591098	0.0	0.0	0.0	0.0
6.41057623837634	0.485118109481103	0.0	0.0	0.0	0.0	0.0
6.89569434785744	0.0	0.0	0.0	0.0	0.0	0.0

Рисунок 35 — Расчёт таблицы конечных разностей для примера

```
newton_function(x, y)
```

P(q) = $-0.000487357872973849 03q^6 + 0.010680523456931076q^5 - 0.095299921829259811q^4 + 0.45161151943572122q^3 - 1.2810560040162616q^2 + 2.8744209905077583q + 1.4142135623730951$

```
newton_function(x, y, var=True)
```

P(x) = $-1.9962178477008866 \cdot 10^{-6}x^6 + 0.00012134586728517958x^5 - 0.0030164640675394364x^4 + 0.039795459198146133x^3 - 0.30744006923693995x^2 + 1.6567332613233629x + 0.028022025506627992$

Рисунок 36 — Вывод интерполяционного полинома для примера

Результаты

```
res = [newton_interpolation(i, x, y) for i in x]
comp_res = [np.isclose(res[i], y[i]) for i in range(len(res))]
display(Markdown(f"<text style=font-size:16px;font-family:serif>Результат: {res}<br>Значения функции: {list(y)}<br>Сравнение значений в заданных узлах: {comp_res} </text>"))
```

Результат: [1.4142135623730951, 3.3740833120550104, 4.437510780292646, 5.222273164980759, 5.861996887936125, 6.41057623837634, 6.895694347857444]
Значения функции: [1.4142135623730951, 3.3740833120550104, 4.437510780292646, 5.222273164980759, 5.861996887936125, 6.410576238376338, 6.895694347857441]
Сравнение значений в заданных узлах: [True, True, True, True, True, True, True]

Рисунок 37 — Сравнение результата интерполирования со значениями функции

Графики

```
newtonxs = newton_interpolation(x, x, y)
```

```
plt.plot(x, y, color='blue', label='f(x)', linewidth=3)
plt.plot(x, newtonxs, linestyle='dashed', color='cyan', linewidth=2, label='P(x)')
plt.plot(x, newtonxs, 'bo', markersize=8, color='red', label='узлы')
plt.rcParams['figure.figsize'] = (10, 10)
plt.legend();
```

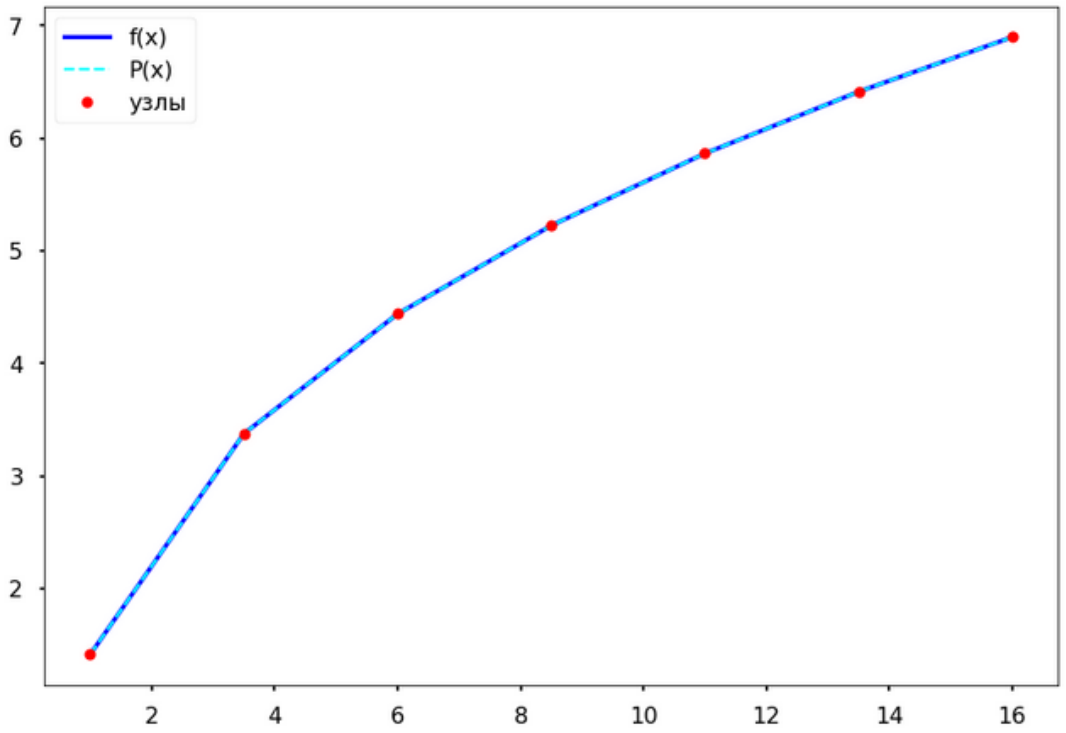

Рисунок 38 — Визуализация решения с узлами

```
xs = np.linspace(x_0, x_1, 50)
ys = f(xs)
newtonxs = [newton_interpolation(p, xs, ys) for p in xs]
```

```
plt.plot(xs, ys, color='blue', label='f(x)', linewidth=3)
plt.plot(xs, newtonxs, linestyle='dashed', color='cyan', linewidth=2, label='P(x)')
plt.rcParams['figure.figsize'] = (10,10)
plt.legend();
```

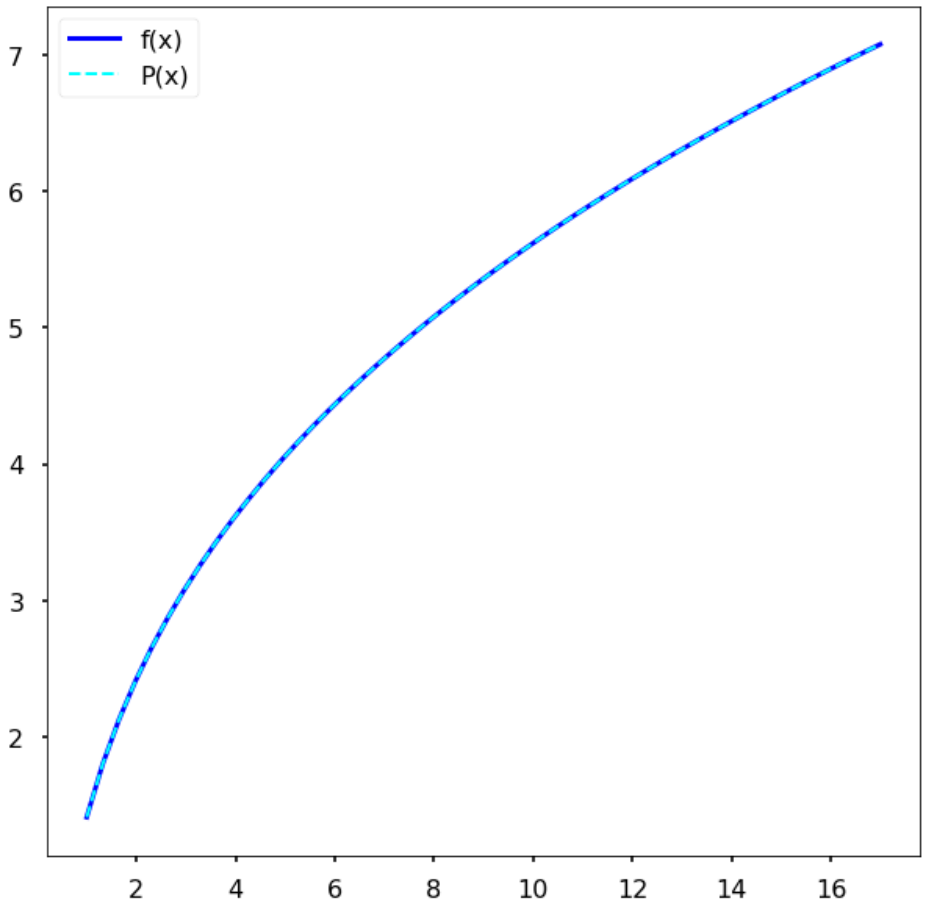

Рисунок 39 — Визуализация решения без узлов

6. ПОСТРОЕНИЕ НАИЛУЧШЕГО ПРИБЛИЖЕНИЯ ФУНКЦИИ. ПАДЕ-АППРОКСИМАЦИЯ [N+2/N]

6.1 Описание метода

На первом шаге паде-аппроксимации мы раскладываем исходную функцию $f(x)$в ряд Тейлора в окрестности начала координат и ищем приближение в виде дробно-рациональной функции, числитель которой является алгебраическим многочленом степени n, а знаменатель тоже является алгебраическим многочленом степени n. (см. формулу 32)

$$f(x)=\sum_{k=0}^{\infty} c_k x^k \tag{32}$$

Разложим получившуюся дробь в ряд Тейлора в окрестности нуля и потребуем, чтобы максимальное количество коэффициентов из разложения функции и разложения дроби — совпадали. (см. формулу 33)

$$[n+2/n]_f=\frac{a_0+a_1 x+...+a_n x^n}{b_0+b_1 x+...+b_n x_n}=\sum_{k=0}^{\infty} d_k x^k,\ d_i=c_i,\ где i=0...2n+2 \tag{33}$$

Следующим шагом является нахождение коэффициентов a_iи b_i. Для этого воспользуемся соотношением (33) и приведём его к общему знаменателю. Таким образом получиться следующая формула. (см. формулу 34)

$$\sum_{k=0}^{n+2} a_k x^k=\sum_{i=0}^{2n+2} c_i x^i \sum_{j=0}^{n} b_j x^j \tag{34}$$

В итоге мы получаем систему линейных алгебраических уравнений. Число уравнений совпадает с числом свободных параметром, но так как коэффициентов в исходном представлении на единицу больше, то можно зафиксировать какой-то из коэффициентов числителя и знаменатели и придать ему какое-то значение. В данном случае, пусть $b_0=1$, то есть свободный член равен единице. Фиксирование коэффициента b_0— ничему не помешает. Таким образом, получается система где число уравнений и неизвестных совпадают. (см. формулу 35)

$$\underbrace{\begin{bmatrix} c_3 & c_4 & \cdots & c_{n+1} & c_{n+2} \\ c_4 & \cdots & \cdots & \cdots & c_{n+3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_{n+1} & \cdots & \cdots & \cdots & c_{2n} \\ c_{n+2} & c_{n+3} & \cdots & c_{2n} & c_{2n+1} \end{bmatrix}}_{\textit{Ганкелева матрица}} \begin{bmatrix} b_n \\ b_{n-1} \\ \vdots \\ b_2 \\ b_1 \end{bmatrix} = - \begin{bmatrix} c_{n+3} \\ c_{n+2} \\ \vdots \\ c_{2n+1} \\ c_{2n+2} \end{bmatrix} \quad (35)$$

Матрица коэффициентов представляется в виде ганкелевой матрицы, то есть на n-побочных диагоналях у данной матрицы представлены одинаковые элементы.

После того как коэффициенты b_i— найдены, то теперь можно перейти к нахождению коэффициентов a_i. Они находятся по следующей формуле. (см. формулу 36)

$$a_0 = c_0 b_0 \quad (36)$$

$$a_1 = c_1 b_0 + c_0 b_1$$

$$\vdots$$

$$a_{n-1} = c_{n-1} b_0 + c_{n-2} b_1 + \ldots + c_0 b_{n-1}$$

$$a_n = c_n b_0 + c_{n-1} b_1 + \ldots + c_0 b_n$$

6.2 Программная реализация

Данный метод реализуется посредством создания функции — *pade*, которая по функции и значению степени результирующего полинома применяет вышеописанные алгоритм. Реализация функции показана на рисунке. (см. рисунок 40)

```
Clear[pade]
pade[f_, n_Integer] := Module[
  {
   a, b,
   c,
   res, hankelMatrix
  },
  c = CoefficientList[Series[f, {x, 0, 2 n + 2}], x];
  hankelMatrix = Table[c[[i + j]], {j, 4, n + 3}, {i, 0, n - 1}];
  res = Table[c[[i]], {i, n + 4, 2 n + 3}];
  b = Composition[Abs, Prepend[#, 1] &, Reverse][LinearSolve[hankelMatrix, res]];
  a = Dot[Take[b, {1, #}], Reverse[Take[c, {1, #}]]] & /@ Range[n + 1];
  Sum[a[[i + 1]] x^i, {i, 0, n}] / Sum[b[[i + 1]] x^i, {i, 0, n}]
 ]
```

Рисунок 40 — Программная реализация паде-аппроксимации [n+2/n]

6.3 Анализ результатов

Входные данные продемонстрированы в следующих формулах. (см. формулы 37-40)

$$f_1 = \frac{1}{1+\sin\left(x^2\right)} \quad , \quad n_1 = 6 \tag{37}$$

$$f_2(x) = \sqrt{\frac{1+\frac{1}{2}x}{1+2x}}, \; n_2 = 2 \tag{38}$$

$$\ln(1+x), \; n_3 = 2 \tag{39}$$

$$f_4(x) = \frac{e^{-x}}{1+x}, \; n_4 = 4 \tag{40}$$

```
f1 = 1/(1 + Sin[x^2])
n1 = 6;
res1 = pade[f1, n1]
```

$$\frac{1}{1+\sin\left[x^2\right]}$$

$$\frac{1+\frac{11\,x^2}{686}+\frac{3\,x^4}{49}+\frac{8647\,x^6}{41160}}{1+\frac{697\,x^2}{686}+\frac{53\,x^4}{686}+\frac{4307\,x^6}{41160}}$$

```
Plot[{res1, f1}, {x, -1.5, 1.5},
 PlotLegends → {"Оригинал", "Паде"},
 AxesLabel → Automatic,
 Filling → {1 → {2}},
 FillingStyle → Directive[Opacity[0.1], Yellow],
 PlotStyle → {{Red}, {Blue}},
 ImageSize → Large]
```

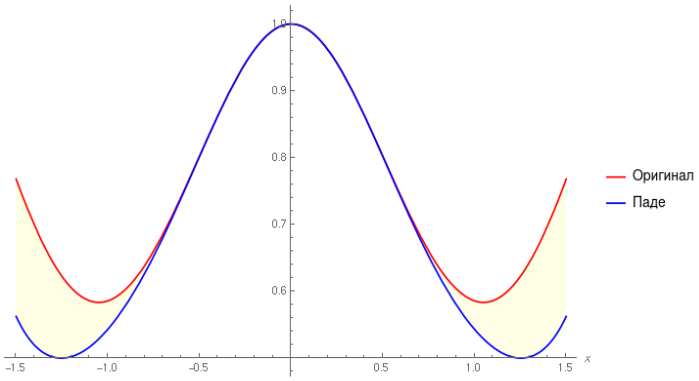

Рисунок 41 — Визуализация решения для примера №1

```
f2 = Sqrt[(1 + 1/2 x)/(1 + 2 x)]
n2 = 2;
res2 = pade[f2, n2]
```

$$\sqrt{\frac{1+\frac{x}{2}}{1+2\,x}}$$

$$\frac{1+\frac{76577\,x}{32486}+\frac{387441\,x^2}{259888}}{1+\frac{201883\,x}{64972}+\frac{1192703\,x^2}{519776}}$$

```
Plot[{res2, f2}, {x, 0, 4},
 PlotLegends → {"Оригинал", "Паде"},
 AxesLabel → Automatic,
 Filling → {1 → {2}},
 FillingStyle → Directive[Yellow],
 PlotStyle → {{Red}, {Blue}},
 ImageSize → Large]
```

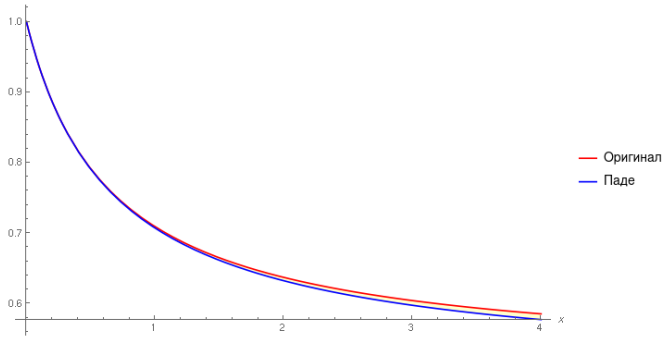

Рисунок 42 — Визуализация решения для примера №2

```
f3 = Log[1 + x]
n3 = 2;
res3 = pade[f3, n3]
```

$$\text{Log}[1 + x]$$

$$\frac{x + \frac{5x^2}{6}}{1 + \frac{4x}{3} + \frac{2x^2}{5}}$$

```
Plot[{res3, f3}, {x, 0, 5},
 PlotLegends → {"Оригинал", "Паде"},
 AxesLabel → Automatic,
 Filling → {1 → {2}},
 FillingStyle → Directive[Opacity[0.1], Yellow],
 PlotStyle → {{Red}, {Blue}},
 ImageSize → Large]
```

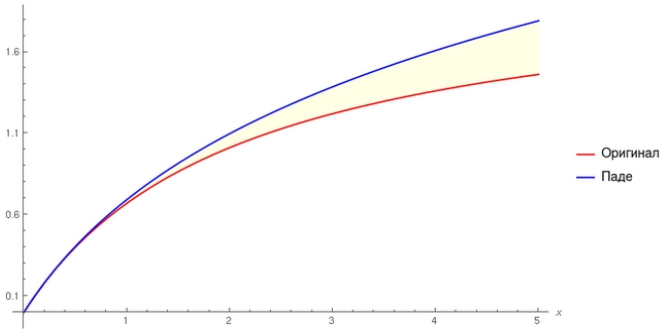

Рисунок 43 — Визуализация решения для примера №3

```
f4 = e^-x / (1 + x)
n4 = 4;
res4 = pade[f4, n4]
```

$$\frac{e^{-x}}{1 + x}$$

$$\frac{1 - \frac{385\,398\,x}{580\,625} + \frac{47\,907\,x^2}{232\,250} - \frac{40\,762\,x^3}{1\,045\,125} + \frac{62\,767\,x^4}{13\,006\,000}}{1 + \frac{775\,852\,x}{580\,625} + \frac{219\,909\,x^2}{580\,625} + \frac{232\,882\,x^3}{5\,225\,625} + \frac{1\,203\,329\,x^4}{585\,270\,000}}$$

```
Plot[{res4, f4}, {x, 0, 6},
 PlotLegends → {"Оригинал", "Паде"},
 AxesLabel → Automatic,
 Filling → {1 → {2}},
 FillingStyle → Directive[Opacity[0.1], Yellow],
 PlotStyle → {{Red}, {Blue}},
 ImageSize → Large]
```

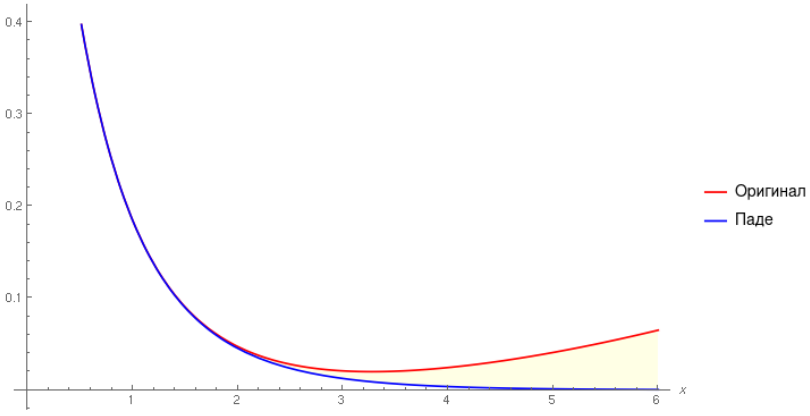

Рисунок 44 — Визуализация решения для примера №4

7. КВАДРАТУРНЫЕ ФОРМУЛЫ ТИПА ЭЙЛЕРА. ФОРМУЛА ГРЕГОРИ.

7.1 Описание метода

На первом шаге нужно вычислить значение формулы трапеции, которая может быть представлена следующим образом. (см. формулу 41)

$$\int_a^b f(x)dx \approx T_n = \frac{h}{2}\left(f_0 + 2\sum_{i=1}^{n-1} f_i + f_n\right) \quad (41)$$

Согласно этой формуле, интервал интегрирования разбивается на некоторое число равных малых интервалов, а дальше на каждом малом интервале мы применяем формулу трапеции.

На следующем шаге, с помощью конечных разностей, находим остаточный член и получаем следующее выражение. (см. формулу 42)

$$\int_a^b f(x)dx \approx T_n - \frac{h}{12}\left[\Delta f_{n-1} - \Delta f_0\right] - \frac{h}{24}\left[\Delta^2 f_{n-2} - \Delta^2 f_0\right] - \frac{19h}{720}\left[\Delta^3 f_{n-3} - \Delta f_3\right] - \frac{3l}{16} \quad (42)$$

Формула Грегори удобна тем, что не нужно искать производные подынтегральной функции.

7.2 Программная реализация

Данный метод реализуется посредством создания 3 функций — *trapezoidalFormula* , которая рассчитывает формулу трапеции, *finiteDifferences*, которая рассчитывает таблицу конечных разностей, *gregory*, которая рассчитывает формулу Грегори. Реализация функций показаны на рисунках. (см. рисунок 45-47)

```
Clear[trapezoidalFormula]
trapezoidalFormula[f_, {a_Real, b_Real}, n_Integer] := Module[
  {
   h = (b - a)/n,
   xk
  },
  xk = Table[a + h i, {i, 0, n}];
  N[h/2 ((f /. x → xk[[1]]) + 2 Sum[(f /. x → xk[[i]]), {i, 2, n}] + (f /. x → xk[[n + 1]]))]]
```

Рисунок 45 — Программная реализация расчёта формулы трапеции

```
Clear[finiteDifferences]
finiteDifferences[y_List] := Module[
  {
   n = Length[y],
   coef
  },
  coef = ConstantArray[0, {n, 6}];
  coef[[All, 1]] = y;
  Do[
   coef[[i, j]] = coef[[i + 1, j - 1]] - coef[[i, j - 1]],
   {j, 2, 6}, {i, n - j + 1}];
  coef
 ]
```

Рисунок 46 — Программная реализация расчёта таблицы конечных разностей

```
Clear[gregory]
gregory[f_, {a_Real, b_Real}, n_Integer] := Module[
  {
   h = (b - a)/n,
   xk,
   y,
   coefs
  },
  xk = Table[a + h i, {i, 0, n}];
  y = Composition[N, f /. x → #1 &][Range[n]];
  coefs = finiteDifferences[y];
  trapezoidalFormula[f, {a, b}, n] - h/12 (coefs[[n - 1, 1]] - coefs[[1, 1]]) - h/24 (coefs[[n - 2, 2]] - coefs[[1, 2]]) - 19 h/720 (coefs[[n - 3, 3]] - coefs[[1, 3]]) - 3 h/160 (coefs[[n - 4, 4]] - coefs[[1, 4]]) - 863 h/60 480 (coefs[[n - 6, 6]] - coefs[[1, 6]])
 ]
```

Рисунок 47 — Программная реализация расчёта формулы Грегори

7.3 Анализ результатов

Входные данные продемонстрированы в следующих формулах. (см. формулы 43-46).

$$f_1(x)=\sin\left(x^2\right)+1,\ [1,10],\ n_1=150\,000 \tag{43}$$

$$f_2(x)=\sqrt{\frac{1+\frac{1}{2}x}{1+2x}},\ [1,5],\ n_2=100\,000 \tag{44}$$

$$f_3(x)=\sqrt{1+\frac{1}{x}},\ [1,4],\ n_3=120\,000 \tag{45}$$

$$f_4(x)=\sin(\sin(x)),\ [0,4],\ n_4=10000 \qquad (46)$$

На графиках продемонстрированы приближения к решению в зависимости от количества разбиений. (см рисунки 48-51)

```
f1 = Sin[x^2] + 1;

NIntegrate[f1, {x, 1, 10}]

9.2734

gregory[f1, {1., 10.}, 150000]

9.2734

res1 = ConstantArray[NIntegrate[f1, {x, 1, 10}], 500 - 6];

test1 = Table[gregory[f1, {1., 10.}, i], {i, 7, 500}];

ListLinePlot[{res1, test1},
 PlotRange → {1, 12},
 PlotLegends → {"Wolfram", "Приближения"},
 PlotLabel → Style[Framed["Пример №1: Sin(x^2)+1"], 16, Bold, Black],
 PlotStyle → {{Red}, {Blue}},
 ImageSize → Large]
```

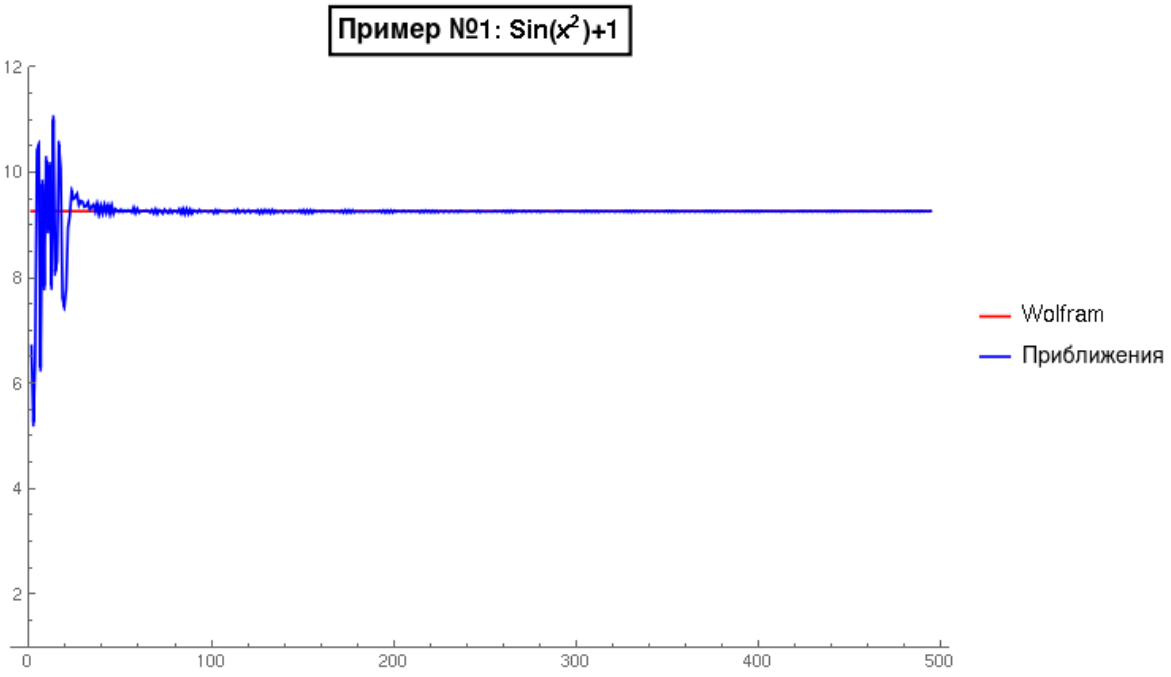

Рисунок 48 — Пример работы программной реализации для примера №1

```
f2 = x/(1 + x);
NIntegrate[f2, {x, 1, 5}]
2.90139
gregory[f2, {1., 5.}, 100 000]
2.90139
res2 = ConstantArray[NIntegrate[f2, {x, 1, 5}], 500 - 7];
test2 = Table[gregory[f2, {1., 5.}, i], {i, 7, 500}];
ListLinePlot[{res2, test2},
 PlotRange → {{2.5, 2.95}},
 PlotLegends → {"Wolfram", "Приближения"},
 PlotLabel → Style[Framed["Пример №2: x/(1 + x)"], 16, Bold, Black],
 PlotStyle → {{Red}, {Blue}},
 ImageSize → Large]
```

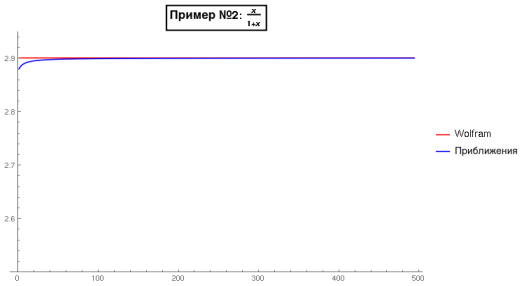

Рисунок 49 — Пример работы программной реализации для примера №2

```
f3 = Sqrt[1 + 1/x];
NIntegrate[f3, {x, 1, 4}]
3.62018
gregory[f3, {1., 4.}, 120 000]
3.62018
res3 = ConstantArray[NIntegrate[f3, {x, 1, 4}], 500 - 7];
test3 = Table[gregory[f3, {1., 4.}, i], {i, 7, 500}];
ListLinePlot[{res3, test3},
 PlotRange → {3.5, 3.7},
 PlotLegends → {"Wolfram", "Приближения"},
 PlotLabel → Style[Framed["Пример №3: Sqrt[1 + 1/x]"], 16, Bold, Black],
 PlotStyle → {{Red}, {Blue}},
 ImageSize → Large]
```

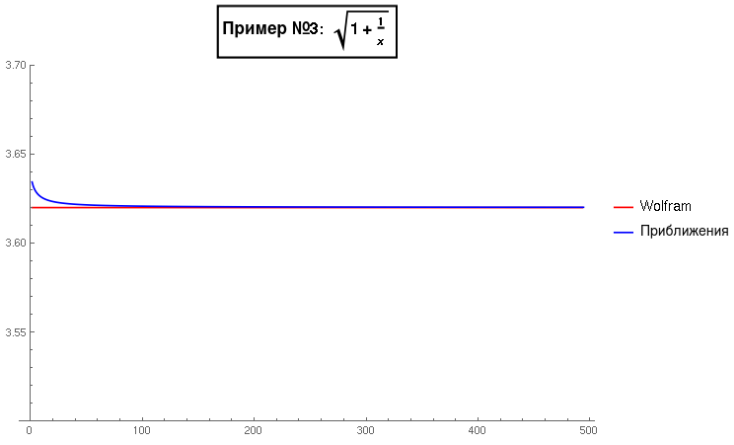

Рисунок 50 — Пример работы программной реализации для примера №3

```
f4 = Sin[Sin[x]];

NIntegrate[f4, {x, 0, 4}]

1.45747

gregory[f4, {0., 4.}, 10 000]

1.45747

res4 = ConstantArray[NIntegrate[f4, {x, 0, 4}], 1000 - 7];

test4 = Table[gregory[f4, {0., 4.}, i], {i, 7, 1000}];

ListLinePlot[{res4, test4},
 PlotRange → {1, 1.5},
 PlotLegends → {"Wolfram", "Приближения"},
 PlotLabel → Style[Framed["Пример №4: Sin(Sin(x))"], 16, Bold, Black],
 PlotStyle → {{Red}, {Blue}},
 ImageSize → Large]
```

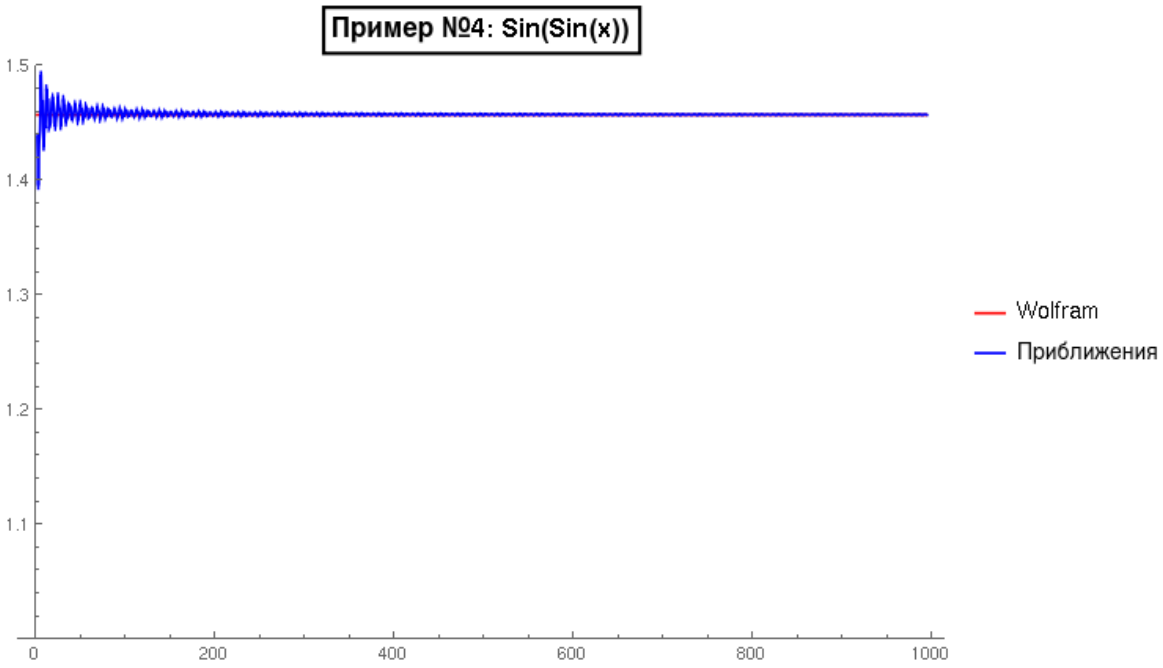

Рисунок 51 — Пример работы программной реализации для примера №4

ЗАКЛЮЧЕНИЕ

Были выполнены поставленные задачи, а именно:

- изучить теорию по данным методам;
- написать программную реализацию методов с использованием теоретических данных;
- проверить правильность работы методов;
- проанализировать полученные результаты.

В дальнейшей перспективе целесообразно освоение других вычислительных методов и применение их на практике.

СПИСОК ЛИТЕРАТУРЫ

1. Юхно Л. Х. Модификация некоторых методов типа сопряжённых направлений для решения систем линейных алгебраических уравнений / Л. Х. Юхно; Ж. вычил. матем. и матем. физ.., 2007, том 47, номер 11, 1811-1818
2. В. Б. Хазанов. Конспект лекций «Численные методы алгебры»
3. В. Б. Хазанов. Конспект лекций «Численные методы анализа»
4. В. В. Воеводин. Численные методы алгебры. Теория и алгоритмы Изд-во «Наука», Москва, 1966
5. А. Н. Пакулина. Вычислительный практикум по методам вычислений, учебное пособие. СПбГУ, математико-механический факультет. Санкт-Петербург, 2016.
6. W. Gander. Algorithms for the QR-Decomposition. Zuerich, 1980.
7. Python Programming and Numerical Methods [Электронные ресурс] / Newton’s Polynomial Interpolation; ред. Qingkai Kong, Timmy Siauw, Alexandre Bayen, 2021. – Режим доступа: https://pythonnumericalmethods.berkeley.edu/notebooks/Index.html, свободный. — Загл. с экрана — Яз. англ.
8. QR-алгоритм [электронный ресурс] / режим доступа: https://algowiki-project.org/ru/QR-%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC, свободный. — Загл. с экрана. — Яз. рус.