МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных систем

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

Дата сдачи на проверку:

«__»__________ 2025 г.

Проверено:

«__»__________2025 г.

Вариант 18

Отчет по лабораторной работе № 2

по дисциплине

«Вычислительная математика»

Разработал студент гр. ИВТб 2302-05-00 ________________ /Соловьев А.С./

(Подпись)

Проверил заведующий кафедры ЭВМ ________________ /Старостин П.А./

(Подпись)

Работа защищена «__»____________2025 г.

Киров 2025

1 Задания

Задание 1

Решить систему линейных уравнений 4-го порядка методом Гаусса с точностью e=0,001. Уравнение системы:

$$\begin{cases} 0,17*x1-0,13*x2-0,11*x3-0,12*x4=0,22 \\ 1,00*x1-1,00*x2-0,13*x3+0,13*x4=0,11 \\ 0,35*x1+0,33*x2+0,12*x3+0,13*x4=0,12 \\ 0,13*x1+0,11*x2-0,13*x3-0,11*x4=1,00 \end{cases}$$

Задание 2

Решить систему линейных уравнений 4-го порядка с точностью e=0,0001: методом простой итерации. Уравнение системы:

$$\begin{cases} x1=0,23*x1-0,04*x2+0,21*x3-0,18*x4+1,24 \\ x2=0,45*x1-0,23*x2+0,06*x3-0,88 \\ x3=0,26*x1+0,34*x2-0,11*x3+0,62 \\ x4=0,05*x1-0,26*x2+0,34*x3-0,12*x4-1,17 \end{cases}$$

Задание 3

Решить систему линейных уравнений 3-го порядка методом обратной матрицы с точностью до e=0.001

$$\begin{cases} 4*x1+8*x2+7*x3=9 \\ x1+2*x2+2*x3=2 \\ 2*x1+3*x2+x3=9 \end{cases}$$

Задание 4

Решить систему нелийнейных уравнений 2-го порядка метод Ньютона с точность e=0.001: Уравнение системы:

$$\begin{cases} 30*x^2+7*y^2-1=0 \\ \sin(4*x-0,5*y)+5*x=0 \end{cases}$$

2 Теорическая часть.Описание методов решения уравнений

1. **Метод Гаусса**

   Метод Гаусса состоит из 2 действий: прямого и обратного хода Гаусса. Прямой ход - привведние матрицы к ступенчатому виду. Обратный ход - последовательное нахождение корней системы.

   Прямой метод Гаусса заключается в приведении матрицы A к верхнетреугольному виду с помощью элементарных преобразований строк(сложение строк, домножение на константу).

Для каждого шага k (от 1 до $n-1$):

1. Выбираем ведущий элемент a_{kk}. 2. Обнуляем элементы ниже диагонали в k-м столбце:

$$a_{ik} = a_{ik} - \frac{a_{ik}}{a_{kk}} a_{kk}, \quad \forall i > k.$$

3. Обновляем правую часть:

$$b_i = b_i - \frac{a_{ik}}{a_{kk}} b_k, \quad \forall i > k.$$

После $n-1$ шагов матрица принимает **верхнетреугольный вид**:

$$U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ 0 & 0 & \ddots & u_{(n-1)n} \\ 0 & 0 & 0 & u_{nn} \end{bmatrix}.$$

2. **Обратный ход**

Теперь решаем **треугольную систему** сверху вниз:

$$u_{nn} x_n = b_n \Rightarrow x_n = \frac{b_n}{u_{nn}}.$$

Затем находим остальные неизвестные по формуле:

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} u_{ij} x_j}{u_{ii}}, \quad i = n-1, n-2, \dots, 1.$$

3 Метод простых итераций

Рассмотрим систему линейных уравнений:

$$A\mathbf{x} = \mathbf{b}, \tag{1}$$

где:A — квадратная матрица коэффициентов размерности $n \times n$; $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ — вектор неизвестных; $\mathbf{b} = (b_1, b_2, \ldots, b_n)^T$ — вектор свободных членов.

Метод простых итераций (метод последовательных приближений) заключается в переходе к эквивалентному виду:

$$\mathbf{x} = P\mathbf{x} + \mathbf{g}, \tag{2}$$

где P — итерационная матрица, а $\mathbf{g}$ — вектор.

1. **Итерационный процесс**

   Рекуррентная формула метода имеет вид:

   $$\mathbf{x}^{(k+1)} = P\mathbf{x}^{(k)} + \mathbf{g}, \quad k = 0, 1, 2, \ldots \tag{3}$$

   Процесс продолжается до выполнения условия остановки:

   $$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \varepsilon, \tag{4}$$

   где ε — заданная точность.

2. **Алгоритм метода** 1. Выбираем начальное приближение $\mathbf{x}^{(0)}$. 2. Вычисляем следующее приближение:

   $$\mathbf{x}^{(k+1)} = P\mathbf{x}^{(k)} + \mathbf{g}.$$

3. Проверяем условие сходимости:

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \varepsilon.$$

Если оно выполнено, процесс завершается. 4. Если условие не выполнено, увеличиваем k и повторяем шаги 2-3.

4 Метод LU разложения с перестановками

LU-разложение представляет собой представление квадратной матрицы A в виде произведения трех матриц:

$$A = PLU,$$

где P — матрица перестановок, L — нижняя треугольная матрица, а U — верхняя треугольная матрица. Это разложение позволяет решить систему линейных уравнений и вычислять определители и обратные матрицы.

Метод LU-разложения с перестановками используется для приведения матрицы A к верхней и нижней треугольной форме с учетом возможных проблем с числовой устойчивостью. В отличие от стандартного метода LU-разложения, при котором матрица A может не быть разложена без ошибок, в методе с перестановками строки матрицы переставляются так, чтобы в процессе преобразований не возникали деления на ноль или слишком малые числа.

Процесс разложения можно описать следующим образом:

1. Перестановка строк: Для обеспечения стабильности выбираются такие строки, которые минимизируют числовые погрешности при делении. Это

делается с помощью матрицы перестановок P, которая переставляет строки исходной матрицы A.

2. Приведение к верхней треугольной форме: Затем с использованием метода Гаусса, но с учетом перестановок строк, матрица A приводится к верхней треугольной форме U.

3. Получение нижней треугольной матрицы: После приведения к верхней треугольной матрице элементы, которые не являются главной диагональю, записываются в нижнюю треугольную матрицу L. При этом на главной диагонали матрицы L стоят единицы.

Таким образом, результатом разложения является три матрицы:

$$A = PLU,$$

где:

$$P = \text{матрица перестановок}, \quad L = \text{нижняя треугольная матрица}, \quad U = \text{верхняя треу}$$

Решение системы линейных уравнений Рассмотрим систему линейных уравнений:

$$A\mathbf{x} = \mathbf{b},$$

где A — матрица коэффициентов, $\mathbf{x}$ — вектор неизвестных, $\mathbf{b}$ — вектор правых частей.

Для решения этой системы с использованием LU-разложения с перестановками, сначала необходимо разложить матрицу A на P, L и U, так что:

$$A = PLU.$$

Затем можно записать систему как:

$$PLU\mathbf{x} = \mathbf{b}.$$

$$Ly = P\mathbf{b}$$

Находим значения для каждого y_i по формуле

$$y_i = b_i - \sum_{k=0}^{i-1} L_{ik} y_k$$

Найдем знаечние для каждого x_i по формуле

$$x_i = y_i - \sum_{k=0}^{i-1} \frac{U_{ik} x_k}{U_{ii}}$$

Мы получили решение для данной системы.

4.1 Вычисление обратной матрицы с использованием LU-разложения

Обратная матрица A^{-1} может быть вычислена с использованием LU-разложения с перестановками. Поскольку $A = PLU$, то для вычисления A^{-1} необходимо решить систему линейных уравнений для каждого столбца единичной матрицы.

Процесс нахождения обратной матрицы включает следующие шаги:

Для каждого столбца единичной матрицы I решается система:

$$A\mathbf{x}_i = \mathbf{e}_i,$$

где $\mathbf{e}_i$ — i-й столбец единичной матрицы.

. После разложения $A = PLU$, эта система может быть записана как:

$$PLU\mathbf{x}_i = \mathbf{e}_i.$$

После того, как мы получим все столбцы $\mathbf{x}_i$, они составляют столбцы обратной матрицы A^{-1}.

Таким образом, метод LU-разложения с перестановками не только позволяет решать системы линейных уравнений, но и вычислять обратную матрицу. При этом процесс нахождения обратной матрицы сводится к решению системы линейных уравнений для каждого столбца единичной матрицы, что делает метод эффективным и численно стабильным.

5 Алгоритм решения СНАУ методом Ньютона

Задача состоит в нахождении корней системы нелинейных алгебраических уравнений:

$$F(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{pmatrix} = 0$$

где $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ — вектор переменных, а $F(\mathbf{x})$ — вектор функций.

Метод Ньютона для системы нелинейных уравнений представляет собой итерационный процесс, который записывается следующим образом:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - J^{-1}(\mathbf{x}_k)F(\mathbf{x}_k)$$

где $J(\mathbf{x}_k)$ — якобиан системы, то есть матрица частных производных:

$$J(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

Алгоритм решения:

1. Выберите начальное приближение $\mathbf{x}_0$.
2. Для каждого шага k выполните следующие действия:
   (a) Вычислите $F(\mathbf{x}_k)$ и $J(\mathbf{x}_k)$.
   (b) Решите линейную систему:

$$\Delta\mathbf{x}_k = -J^{-1}(\mathbf{x}_k)F(\mathbf{x}_k)$$

(c) Обновите значение вектора $\mathbf{x}_k$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$$

3. Повторяйте шаги до тех пор, пока не будет достигнута желаемая точность:

$$\|F(\mathbf{x}_{k+1})\| < \epsilon$$

где ϵ — заданная точность.

6 Практическая часть

Задание 1

Решить систему линейных уравнений 4-го порядка методом Гаусса с точностью e=0,001. Уравнение системы:

$$\begin{cases} 0,17 * x1 - 0,13 * x2 - 0,11 * x3 - 0,12 * x4 = 0,22 \\ 1,00 * x1 - 1,00 * x2 - 0,13 * x3 + 0,13 * x4 = 0,11 \\ 0,35 * x1 + 0,33 * x2 + 0,12 * x3 + 0,13 * x4 = 0,12 \\ 0,13 * x1 + 0,11 * x2 - 0,13 * x3 - 0,11 * x4 = 1,00 \end{cases}$$

Эту систему можно записать в матричной форме:

$$A \cdot \mathbf{x} = \mathbf{b}$$

где A — матрица коэффициентов, $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$ — вектор переменных, а $\mathbf{b} = \begin{pmatrix} 0.22 \\ 0.11 \\ 0.12 \\ 1.00 \end{pmatrix}$ — вектор правых частей уравнений.

Метод Гаусса состоит из двух этапов:

1. **Прямой ход**: Приведение матрицы коэффициентов A к верхнетреугольному виду с помощью элементарных преобразований строк. В ходе этого этапа мы находим коэффициенты, которые позволяют обнулить элементы ниже главной диагонали.
2. **Обратный ход**: После приведения матрицы к верхнетреугольному виду, начинаем вычисление переменных системы, начиная с последней строки. Для каждой строки вычисляется переменная как:

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{ij} x_j}{a_{ii}}$$

Прямой ход На первом шаге мы применяем метод выбора ведущего элемента, чтобы минимизировать погрешности вычислений. В ходе этого шага мы находим максимальные элементы по столбцам и меняем строки, если это необходимо.

Затем, используя элементарные преобразования строк, мы приводим систему к верхнетреугольному виду. Для каждой строки i вычисляется коэффициент:

$$k_{ij} = \frac{a_{ij}}{a_{ii}}$$

и вычитаем k_{ij} умноженное на строку i из строки j, чтобы обнулить элементы ниже ведущего.

Обратный ход После приведения матрицы к верхнетреугольному виду, мы начинаем обратный ход. Сначала решаем для последней переменной, затем для остальных, двигаясь вверх по строкам.

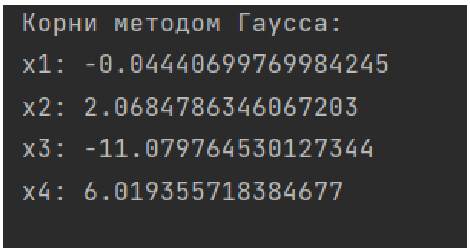
Решение для переменной x_4 на основе последней строки:

$$x_4 = \frac{b_4 - (a_{43}x_3 + a_{44}x_4)}{a_{44}}$$

Затем, по аналогии, решаем для остальных переменных x_3, x_2, и x_1.

Решение После применения метода Гаусса система уравнений была решена. Корни, полученные методом Гаусса:

$$x_1 = -0.044 \pm 0.001, \quad x_2 = 2.068 \pm 0.001, \quad x_3 = -11.080 \pm 0.001, \quad x_4 = 6.019 \pm 0.001$$



```
Корни методом Гаусса:
x1: -0.04440699769984245
x2: 2.0684786346067203
x3: -11.079764530127344
x4: 6.019355718384677
```

Рис. 1: Решение методом Гаусса

Проверим найденные корни в Maxima

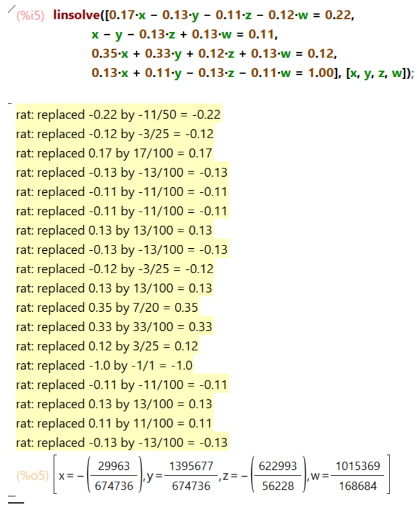


Рис. 2: Решение СЛАУ в Максима

Найденные корни совпадают с корнями, найденными в программе python.

Листинг программы:

```
import numpy as np


def select_lead_elem(matrix):
    matrix = matrix.astype(float)
    num_rows, num_cols = matrix.shape

    for col in range(min(num_rows, num_cols)):  #
        max_index = col + np.argmax(abs(matrix[col:, col]))  #

        if max_index != col:  #                                  ,
            matrix[[col, max_index]] = matrix[[max_index, col]]
    return matrix


def direct_Gausse(matrix):
    matrix = select_lead_elem(matrix)  #

    lead_elem = 0

    for row_1 in matrix:
        for i in range(lead_elem + 1, len(matrix)):
            sub_k = matrix[i][lead_elem] / row_1[lead_elem]
```

```
            matrix[i] = np.subtract(matrix[i], row_1 * sub_k)
        lead_elem += 1
        if lead_elem == len(matrix) - 1:
            break
    return matrix


def redirect_Gause(matrix):
    root = np.zeros(len(matrix))
    for i in range(len(matrix) - 1, -1, -1):
        sum_ax = sum(matrix[i][j] * root[j] for j in range(i + 1, len(matrix)))
        root[i] = (matrix[i][-1] - sum_ax) / matrix[i][i]
    return root


def Gausse(matrix):
    matrix = select_lead_elem(matrix)  #


    matrix = direct_Gausse(matrix)
    root = redirect_Gause(matrix)
    return root


'''
                                  4-                                  e =
   10^(-3):
                  :
            0,17*x1-0,13*x2-0,11*x3-0,12*x4=0,22
            1,00*x1-1,00*x2-0,13*x3+0,13*x4=0,11
            0,35*x1+0,33*x2+0,12*x3+0,13*x4=0,12
            0,13*x1+0,11*x2-0,13*x3-0,11*x4=1,00
            '''
A = np.array([[0.17, -0.13, -0.11, -0.12, 0.22], [1, -1, -0.13, 0.13, 0.11],
   [0.35, 0.33, 0.12, 0.13, 0.12],
               [0.13, 0.11, -0.13, -0.11, 1.00]])
X = Gausse(A)
i = 0
print("                    :")
for x in X:
```

```python
    i += 1
    print(f"x{i}: {x}")
```

Листинг 1: Python код программы

Задание 2

Решить систему линейных уравнений 4-го порядка с точностью e=0,0001: методом простой итерации. Уравнение системы:

$$\begin{cases} x1 = 0,23 * x1 - 0,04 * x2 + 0,21 * x3 - 0,18 * x4 + 1,24 \\ x2 = 0,45 * x1 - 0,23 * x2 + 0,06 * x3 - 0,88 \\ x3 = 0,26 * x1 + 0,34 * x2 - 0,11 * x3 + 0,62 \\ x4 = 0,05 * x1 - 0,26 * x2 + 0,34 * x3 - 0,12 * x4 - 1,17 \end{cases}$$

Метод простых итераций используется для численного решения данной системы. В методе простых итераций на каждом шаге обновляются значения переменных, пока разность между старым и новым значением не станет меньше заданной погрешности.

Шаги решения

1. Представление системы уравнений в виде матрицы и вектора свободных членов:

Матрица коэффициентов системы:

$$P = \begin{bmatrix} 0.23 & 0.04 & 0.21 & -0.18 \\ 0.45 & 0.23 & 0.06 & 0 \\ 0.26 & 0.34 & -0.11 & 0 \\ 0.05 & -0.26 & 0.34 & -0.11 \end{bmatrix}$$

Вектор свободных членов:

$$g = \begin{bmatrix} 1.24 \\ -0.88 \\ 0.62 \\ -1.17 \end{bmatrix}$$

2. Метод простых итераций:

Для решения системы с использованием метода простых итераций применяется следующая формула обновления для каждого элемента:

$$x_i^{(k+1)} = \sum_{j=1}^{n} P_{ij} \cdot x_j^{(k)} + g_i$$

где $x_i^{(k)}$ — это значение переменной на k-м шаге, а $x_i^{(k+1)}$ — новое значение на $(k+1)$-м шаге.

Процесс продолжается до тех пор, пока разница между старыми и новыми значениями переменных не станет меньше заданной погрешности (в данном случае 0.001).

Функция работает следующим образом:

- Инициализируется вектор x с нулевыми значениями.
- В цикле происходит итерационное обновление значений вектора x согласно формуле метода простых итераций.
- Итерации продолжаются до тех пор, пока разница между новыми и старыми значениями переменных не станет меньше заданной погрешности.

4. Решение системы:

Применив метод простых итераций с матрицей коэффициентов P и вектором свободных членов g, мы получаем решение системы уравнений.

$$\text{Решение: } [2.070 \pm 0.001, 0.151 \pm 0.001, 1.090 \pm 0.001, -0.663 \pm 0.001]$$

Листинг программы:

```
import numpy as np

def norm_infinity(P):
    return max(np.sum((P), axis=0))

# Функция для вычисления нормы по максимальному элементу в строках
def norm_onee(P):
    for i in range(len(P)):
        if max(np.sum((P),axis = i):

```

```python
    return max(np.sum((P), axis=1))

# Функция для вычисления евклидовой нормы
def norm_twoo(P):
    return max(np.sqrt(np.sum(P**2, axis=1)))


def simple_iteration(P, g, error=0.001):
    n = len(g)
    x = [0] * n

    norm_inf = norm_infinity(P)
    norm_one = norm_onee(P)
    norm_two = norm_twoo(P)

    print(f"Норма по максимальному элементу в столбцах: {norm_inf}")
    print(f"Норма по максимальному элементу в строках: {norm_one}")
    print(f"Норма по квадратам: {norm_two}")

    while(True):
        x_new = [sum(P[i][j] * x[j] for j in range(n)) + g[i] for i in range(n)
            ]

        if max(abs(x_new[i] - x[i]) for i in range(n)) < error:
            return x_new

        x = x_new[:]


P = [
    [0.23, -0.04, 0.21, -0.18],
    [0.45, -0.23, 0.06, 0],
    [0.26, 0.34, -0.11, 0],
    [0.05, -0.26, 0.34, -0.11]
]

g = [1.24, -0.88, 0.62, -1.17]


solution = simple_iteration(np.array(P), g)
```

```python
print("Решение:", solution)
```

Листинг 2: Python код программы

7 LU-разложение

Задание 3

Решить систему линейных уравнений 3-го порядка методом обратной матрицы с точностью до e=0.001

$$\begin{cases} 4*x1 + 8*x2 + 7*x3 = 9 \\ x1 + 2*x2 + 2*x3 = 2 \\ 2*x1 + 3*x2 + x3 = 9 \end{cases}$$

Преобразуем матрицу A в разложение $PA = LU$, где:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.25 & -1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 8 & 7 \\ 0 & -1 & -2.5 \\ 0 & 0 & 0.5 \end{bmatrix}.$$

7.1 Вычисление обратной матрицы

Вычислим обратную матрицу по формуле

$$PLU\mathbf{x}_i = \mathbf{e}_i.$$

Для решения системы сначала находим y_i с использованием прямого хода, затем решаем для x_i с использованием обратного хода.

После того, как мы получим все столбцы $\mathbf{x}_i$, они составляют столбцы обратной матрицы A^{-1}.

$$A^{-1} = \begin{bmatrix} -4 & 13 & 2 \\ 3 & -10 & -1 \\ -1 & 4 & 0 \end{bmatrix}$$

Таким образом, метод LU-разложения с перестановками позволяет эффективно вычислить обратную матрицу.

7.2 Найдем корни системы

По формуле вычисления корней через обратную матрциу

$$x = A^{-1} * b$$

Где b - свободные члены системы

Подставив в формулу получаем корни системы

$$\mathbf{x} = \begin{bmatrix} 8 \\ -2 \\ -1 \end{bmatrix}.$$

7.3 Листинг программы:

```
import numpy as np


# LU разложение с перестановками
```

```python
def LU_with_pivoting(A):
    n = len(A)
    L = np.eye(n)
    U = np.copy(A).astype(float)
    P = np.eye(n)
    for i in range(n):
        # Строка с максимальным элементом в i столбце
        max_row = np.argmax(np.abs(U[i:n, i])) + i
        if i != max_row:
            U[[i, max_row], :] = U[[max_row, i], :]
            L[[i, max_row], :i] = L[[max_row, i], :i]
            P[[i, max_row], :] = P[[max_row, i], :]

        for j in range(i + 1, n):
            L[j][i] = U[j][i] / U[i][i]  # Вычисляем элементы L
            U[j, i:n] -= L[j][i] * U[i, i:n]

    return P, L, U  # Возвращаем матрицы P, L и U


# Решает Ly = b (прямой ход)
def forward_substitution(L, b):
    n = len(b)
    y = np.zeros(n)
    for i in range(n):
        y[i] = b[i] - sum(L[i][k] * y[k] for k in range(i))
    return y


# Решает обратный ход Ux = y
def backward_substitution(U, y):
    n = len(y)
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):  # Идчм снизу вверх
        if U[i][i] == 0:
            raise ValueError(f"Нулевая диагональ в U на шаге {i}, невозможно продолжить решение!")
        x[i] = (y[i] - sum(U[i][k] * x[k] for k in range(i + 1, n))) / U[i][i]
    return x

```

```
# Нахождение обратной матрицы через LU-разложение
def inverse_matrix(A):
    n = len(A)
    P, L, U = LU_with_pivoting(A)
    I = np.eye(n)
    inv_A = np.zeros_like(A)

    for i in range(n):
        # Решаем систему P * L * U * x = I_i (i-й столбец единичной матрицы)
        y = forward_substitution(L, np.dot(P, I[:, i]))  # Ly = P * I_i
        inv_A[:, i] = backward_substitution(U, y)  # Ux = y => x = U^-1 * y

    return inv_A


A = np.array([[4, 8, 7], [1, 2, 2], [2, 3, 1]], dtype=float)

inv_A = inverse_matrix(A)

print("Обратная матрица A^{-1}:\n", inv_A)
```

Листинг 3: Python код программы

7.4 Проверка в Maxima

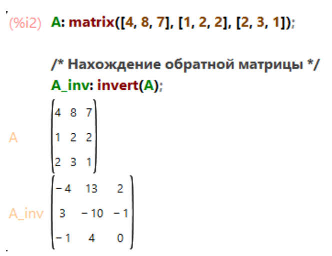


Рис. 3: Правильное решение в Maxima

8 Задание 4

Задание 4

Решить систему нелийнейных уравнений 2-го порядка метод Ньютона с точность e=0.001: Уравнение системы:

$$\begin{cases} 30 * x^2 + 7 * y^2 - 1 = 0 \\ \sin(4 * x - 0,5 * y) + 5 * x = 0 \end{cases}$$

Метод Ньютона

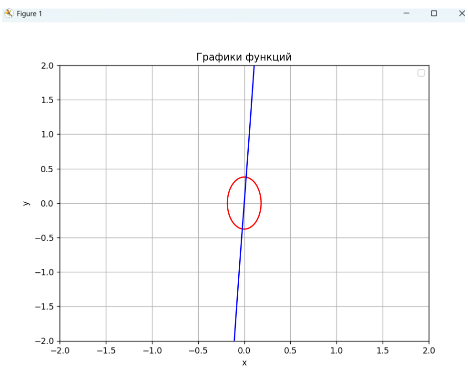


Рис. 4: График функций

Возьмем за начальное приближение $x_0 = 0.1$ и $y_0 = 0.1$ Метод Ньютона использует итерационный процесс:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - J^{-1}(x_k, y_k) \cdot \begin{bmatrix} f_1(x_k, y_k) \\ f_2(x_k, y_k) \end{bmatrix},$$

где $J(x, y)$ — якобиан:

$$J(x, y) = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix}.$$

Частные производные

$$J(x,y) = \begin{bmatrix} 60x & 14y \\ 4\cos(4x-0.5y)+5 & -2\cos(4x-0.5y) \end{bmatrix}.$$

Определитель якобиана:

$$\det J = (60x)(-2\cos(4x-0.5y)) - (14y)(4\cos(4x-0.5y)+5).$$

Обратная матрица:

$$J^{-1} = \frac{1}{\det J}\begin{bmatrix} -2\cos(4x-0.5y) & -14y \\ -(4\cos(4x-0.5y)+5) & 60x \end{bmatrix}.$$

Численное решение

Используя начальное приближение $(x_0, y_0) = (0.1, 0.1)$ и точность $\varepsilon = 10^{-3}$

Решение:

$$x \approx 0.020 \pm 0.001, \quad y \approx 0.375 \pm 0.001.$$

8.1 Листинг программы:

```
import numpy as np


# Определяем функции и их производные
def f1(x, y):
    return 30 * x ** 2 + 7 * y ** 2 - 1


def f2(x, y):
```

```
    return np.sin(4 * x - 0.5 * y) + 5 * x


def df1_dx(x, y):
    return 60 * x


def df1_dy(x, y):
    return 14 * y


def df2_dx(x, y):
    return 4 * np.cos(4 * x - 0.5 * y) + 5


def df2_dy(x, y):
    return -2 * np.cos(4 * x - 0.5 * y)


# Метод Ньютона для решения системы
def newton_method(x0, y0, tol=1e-3, max_iter=100):
    x, y = x0, y0
    for _ in range(max_iter):
        F1 = f1(x, y)
        F2 = f2(x, y)

        J11 = df1_dx(x, y)
        J12 = df1_dy(x, y)
        J21 = df2_dx(x, y)
        J22 = df2_dy(x, y)

        det_J = J11 * J22 - J12 * J21

        J_inv = (1 / det_J) * np.array([[J22, -J12], [-J21, J11]])

        F = np.array([F1, F2])

        delta = J_inv.dot(F)
        x, y = x - delta[0], y - delta[1]

```

```python
        if np.linalg.norm(F, ord=2) < tol:
            break
#Работает за 8 итераций
    return x, y
x0, y0 = 0.1, 0.1

solution = newton_method(x0, y0)
print()
print(f"Решение: x = {solution[0]}, y = {solution[1]}")
```

Листинг 4: Python код программы

8.2 Проверка в Maxima

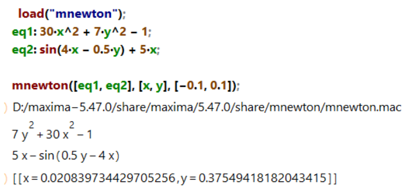


Рис. 5: Правильное решение в Maxima

9 Вывод

В ходе данной лабораторной работы, я смог ознакомлся с 3 методами решения линейных уравнений и 1 для решения нелинейных уравнений. А именно: метод Ньютона, метод Гаусса, LU разложение с перестановками и метод простых итераций. Эти методы были реализованы в программе с использованием высокого языка программирования (Python). Также вычесленные корни были проверены в Maxima, они сошлись с минимальной погрешностью.