МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Московский Авиационный Институт
(национальный Исследовательский Университет)

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

Курсовой проект по дисциплине «Численные методы»

На тему: «Разработка вычислительного алгоритма и программы расчёта параметров стационарного потока идеального газа в канале переменного сечения»

Студент: Бойков П. А.
Группа: М8О-213Б-22
Руководитель: Гидаспов В. Ю.
Оценка: _____
Дата: _____

Москва, 2024

Содержание

Введение

Цель курсового проекта – изучение и программная реализация классических численных методов, исследование их точности и эффективности, а также применение в задаче математического моделирования.

Курсовой проект включает в себя четыре лабораторные работы, нацеленные на изучение определённых классов численных методов, а также прикладную задачу, направленную на применение полученных знаний для математического моделирования физического процесса.

Лабораторная работа №1 направлена на изучение численных методов линейной алгебры. В неё включены алгоритмы прямого решения систем линейных алгебраических уравнений – метод Гаусса и метод прогонки, и алгоритмы итерационного решения – метод простых итераций и метод Зейделя. Также в ней изучаются итерационный и степенной методы нахождения собственных значений и собственных векторов матрицы.

Лабораторная работа №2 сосредоточена на решениях нелинейных уравнений и систем. В ней рассматриваются итерационный метод, метод половинного деления, метод Ньютона и метод секущих.

Лабораторная работа №3 посвящена методам приближения функций, их численному дифференцированию и интегрированию. Она состоит из задач на построение интерполяционных многочленов Лагранжа и Ньютона, сплайн-интерполяции и приближающего многочлена методом наименьших квадратов. Для решения задачи интегрирования рассматриваются методы прямоугольников, трапеций, Симпсона и Рунге-Ромберга-Ричардсона.

Лабораторная работа №4 включает в себя одно задание – задачу Коши для обыкновенного дифференциального уравнения, для решения которой используются методы Эйлера, Рунге-Кутты, Адамса.

Прикладная задача состоит в использовании одного из методов численного решения дифференциальных уравений для расчёта зависимости скорости, давления и плотности стационарного потока воздуха в конусообразном канале от продольной координаты.

Все перечисленные выше алгоритмы реализованы на языке программирования C++, а для динамического построения графиков в прикладной задаче используется язык программирования Python.

Лабораторная работа №1

Задача №1

Реализовать алгоритм Гаусса (с выбором главного элемента). Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

СЛАУ:

$$\begin{cases} 9x_1 - 5x_2 - 6x_3 + 3x_4 = -8 \\ x_1 - 7x_2 - x_3 = 38 \\ 3x_1 - 4x_2 + 9x_3 = 47 \\ 6x_1 - x_2 + 9x_3 + 8x_4 = -8 \end{cases}$$

Решение

Метод Гаусса позволяет решать системы линейных уравнений вида $Ax = b$, где A – квадратная матрица коэффициентов, x – вектор искомых, а b – вектор свободных членов. Метод состоит из двух этапов – прямого хода, приводящего матрицу к верхнетреугольному виду, и обратного хода, в котором с помощью подстановок определяются неизвестные.

На i-том шаге прямого хода выполняются следующие действия:

1. Выбор в i-том столбце максимального по модулю элемента, именуемого главным, и его перемещение с помощью перестановки строк на место элемента a_{ii};
2. Сокращение всех элементов a_{ji}, таких где $j > i$, используя формулы

$$a_{jk} = a_{jk} - a_{ik}\frac{a_{ji}}{a_{ii}}, k = \overline{i \dots n}$$

$$b_j = b_j - b_i\frac{a_{ji}}{a_{ii}}$$

Обратный ход заключается в вычислении корней в обратном порядке с помощью формулы

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

Определитель матрицы A может быть вычислен как произведение диагональных элементов верхнетреугольной матрицы. Однако следует учесть, что каждая перестановка строк в матрице меняет знак определителя, поэтому его формула выглядит следующим образом

$$detA = (-1)^k \prod_{i=1}^{n} a_{ii},$$

где k – количество перестановок строк в матрице.

Для нахождения обратной матрицы размера $n \times n$ нужно решить n СЛАУ, в правой части которых находятся столбцы единичной матрицы. Решениями этих уравений будут являться соответствующие столбцы обратной матрицы. Стоит заметить, что для ускорения расчётов, n уравнений вида $Ax = b$ можно заменить на одно уравнение $Ax = E$, таким образом удастся избежать повторения одни и тех же вычислений при прямом ходе.

Листинг 1. Прямой ход метода Гаусса

```
void process_gaussian_elimination_first_phase(
        std::vector<std::vector<double>> &augm_matrix,
        double eps)
{
    size_t n = augm_matrix.size();
    size_t m = augm_matrix[0].size();
    size_t swap_count = 0;

    for (size_t i = 0; i < n; ++i)
    {
        size_t idx_to_swap = search_leading_element(augm_matrix, i);
        swap_count += (idx_to_swap > i) ? (idx_to_swap - i)
                                        : (i - idx_to_swap);
        std::swap(augm_matrix[i], augm_matrix[idx_to_swap]);

        if (std::abs(augm_matrix[i][i]) < eps)
        {
            throw std::invalid_argument("Invalid matrix");
        }
        for (size_t j = i+1; j < n; ++j)
        {
            double mult = augm_matrix[j][i] / augm_matrix[i][i];

            for (size_t k = i; k < m; ++k)
            {
                augm_matrix[j][k] -= augm_matrix[i][k] * mult;
            }
        }
    }
    if (swap_count & 1)
    {
        for (size_t i = 0; i < n+1; ++i)
        {
            augm_matrix[0][i] *= -1;
        }
    }
```

```
}
```

Для упрощения вычисления определителя, в случае если количество перестановок строк оказалось нечётным, первая строка матрицы умножается на -1, чтобы восстановить правильный знак определителя и в будущем не учитывать -1 в степени количества перестановок.

Листинг 2. Обратный ход метода Гаусса

```
std::vector<std::vector<double>>
process_gaussian_elimination_second_phase(
        std::vector<std::vector<double>> &augm_matrix)
{
    size_t n = augm_matrix.size();
    size_t m = augm_matrix[0].size();
    std::vector<std::vector<double>> roots(n,std::vector<double>(m-n));

    for (size_t k = 0; k < m - n; ++k)
    {
        for (size_t i = n; i > 0; --i)
        {
            roots[i-1][k] = augm_matrix[i-1][n+k];
            for (size_t j = i; j < n; ++j)
            {
                roots[i-1][k] -= augm_matrix[i-1][j]*roots[j][k];
            }
            roots[i-1][k] /= augm_matrix[i-1][i-1];
        }

    }
    return roots;
}
```

Листинг 3. Вычисление определителя

```
double calc_determinant(
        std::vector<std::vector<double>> &augmented_matrix)
{
    size_t n = augmented_matrix.size();
    double det = 1;

    for (size_t i = 0; i < n; ++i)
    {
        det *= augmented_matrix[i][i];
    }

    return det;
}
```

Листинг 4. Вычисление обратной матрицы

```
std::vector<std::vector<double>>
inverse_matrix(
    std::vector<std::vector<double>> matrix,
    double eps)
```

```
{
    size_t n = matrix.size();

    std::vector<std::vector<double>> inv_matrix(n,
std::vector<double>(n));
    std::vector<std::vector<double>> augmented_matrix(n,
std::vector<double>(2*n));

    for (size_t i = 0; i < n; ++i)
    {
        for (size_t j = 0; j < n; ++j)
        {
            augmented_matrix[i][j] = matrix[i][j];
            augmented_matrix[i][n+j] = (i == j) ? 1 : 0;
        }
    }

    process_gaussian_elimination_first_phase(augmented_matrix, eps);
    auto roots =
process_gaussian_elimination_second_phase(augmented_matrix);

    return roots;
}
```

Листинг 5. Вывод решения СЛАУ методом Гаусса

```
x1 = -0.000000
x2 = -5.000000
x3 = 3.000000
x4 = -5.000000
Solution is verified

Determinant = -4239.000000

Inverse matrix:
0.111347 -0.149328 0.132578 -0.041755
0.011323 -0.167728 0.030432 -0.004246
-0.032083 -0.024770 0.080444 0.012031
-0.046001 0.118896 -0.186129 0.142251

A * A^(-1)
1.000000 0.000000 0.000000 0.000000
-0.000000 1.000000 0.000000 -0.000000
-0.000000 -0.000000 1.000000 -0.000000
0.000000 0.000000 0.000000 1.000000
The inverse matrix is verified
```

Задача №2

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трёхдиагональной матрицей.

СЛАУ:

$$\begin{cases} 13x_1 - 5x_2 = -66 \\ -4x_1 + 9x_2 - 6x_3 = -47 \\ -x_2 - 12x_3 - 6x_4 = -43 \\ 6x_3 + 20x_4 - 5x_5 = -74 \\ 4x_4 + 5x_5 = 14 \end{cases}$$

Решение

Метод прогонки, также известный как метод Томаса, представляет собой численный алгоритм для решения СЛАУ с трёхдиагональными матрицами, т.е. такими матрицами, у которых ненулевые элементы есть только на главной диагонали, а также на соседних к ней верхней и нижней диагоналях.

Форма такой системы:

$$\begin{cases} b_1x_1 + c_1x_2 = d_1 \\ a_2x_1 + b_2x_2 + c_2x_3 = d_2 \\ \dots \\ a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1} \\ a_nx_{n-1} + b_nx_n = d_n \end{cases}$$

Метод прогонки так же, как и метод Гаусса состоит из прямого и обратного хода. При прямом ходе вычисляются прогоночные коэффициенты P_i и Q_i.

$$P_1 = -\frac{c_1}{b_1}, \quad Q_1 = -\frac{d_1}{b_1}$$

$$P_1 = -\frac{c_1}{b_1 + a_iP_{i-1}}, \quad Q_1 = -\frac{d_i - a_iQ_{i-1}}{b_i - a_iQ_{i-1}}$$

$$P_n = 0$$

Обратный ход представляет собой вычисление корней в обратном порядке по формуле

$$x_i = P_ix_{i+1} + Q_i$$

Листинг 6. Прямой ход метода прогонки

```
std::pair<std::vector<double>, std::vector<double>>
process_thomas_alg_first_phase(
        std::vector<double> const &coefs_a,
        std::vector<double> const &coefs_b,
        std::vector<double> const &coefs_c,
        std::vector<double> const &coefs_d,
```

```
        double eps)
{
    size_t n = coefs_a.size();
    std::vector<double> coefs_p(n, 0), coefs_q(n, 0);
    if (abs(coefs_b[0]) < eps)
    {
        throw std::invalid_argument("Incorrect system");
    }
    coefs_p[0] = -coefs_c[0] / coefs_b[0];
    coefs_q[0] = coefs_d[0] / coefs_b[0];
    for (size_t i = 1; i < n; ++i)
    {
        double div = coefs_b[i] + coefs_a[i] * coefs_p[i - 1];
        if (abs(div) < eps)
        {
            throw std::invalid_argument("Incorrect system");
        }
        coefs_p[i] = -coefs_c[i] / div;
        coefs_q[i] = (coefs_d[i] - coefs_a[i] * coefs_q[i - 1]) / div;
    }
    return std::make_pair(coefs_p, coefs_q);
}
```

Листинг 7. Обратный ход метода прогонки

```
std::vector<double> process_thomas_alg_second_phase(
        std::vector<double> const &coefs_p,
        std::vector<double> const &coefs_q)
{
    size_t n = coefs_p.size();
    std::vector<double> roots(n, 0);

    roots[n - 1] = coefs_q[n - 1]
    for (size_t i = n - 1; i > 0; --i)
    {
        roots[i - 1] = coefs_p[i - 1] * roots[i] + coefs_q[i - 1];
    }
    return roots;
}
```

Листинг 8. Нахождение детерминанта

```
double calc_determinant(
        std::vector<double> const &coefs_a,
        std::vector<double> const &coefs_b,
        std::vector<double> const &coefs_p)
{
    size_t n = coefs_a.size();
    double det = coefs_b[0];
    for (size_t i = 1; i < n; ++i)
    {
        det *= coefs_b[i] + coefs_a[i] * coefs_p[i - 1];
    }
    return det;
```

```
}
```

Листинг 9. Вывод решения СЛАУ методом прогонки

```
The matrix is tridiagonal
The method is stable for the given system
x1 = -7.00000000
x2 = -5.00000000
x3 = 6.00000000
x4 = -4.00000000
x5 = 6.00000000
Determinant = -130020.00000000
```

Задача №3

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

СЛАУ:

$$\begin{cases} -23 \cdot x_1 - 7 \cdot x_2 + 5 \cdot x_3 + 2 \cdot x_4 = -26 \\ -7 \cdot x_1 - 21 \cdot x_2 + 4 \cdot x_3 + 9 \cdot x_4 = -55 \\ 9 \cdot x_1 + 5 \cdot x_2 - 31 \cdot x_3 - 8 \cdot x_4 = -58 \\ x_2 - 2 \cdot x_3 + 10 \cdot x_4 = -24 \end{cases}$$

Решение

Метод простых итераций представляет собой численный метод для решения систем линейных уравнений. Основная идея заключается в разложении матрицы системы на сумму двух матриц и итеративном вычислении значений неизвестных.

Систему уравнений представим в виде: $Ax = b$

Переписывая для метода итераций: $x^{(k+1)} = Cx^{(k)} + g$, где C – матрица с коэффициентами, и g — вектор. Условие на сходимость – это диагональное доминирование матрицы.

Метод Зейделя считается более улучшенной версией метода простых итераций, поскольку в ходе вычислений он использует уже обновленные значения, что обычно ускоряет сходимость. Формулы обновляются схожим образом, но при вычислении используется больше свежей информации.

Систему можно представить следующим образом:

$$x_i^{(k+1)} = 1/(a_{ii})(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^k)$$

Основное отличие заключается в использовании новых значений x_j в правой части.

Листинг 10. Метод простых итераций

```
solve_with_iterations(
    matrixNxN const &matrix,
    vecN const &const_terms,
    double eps)
{
    if (eps < 0)
    {
        throw std::invalid_argument("Invalid epsilon");
    }
    auto [tr_matrix, tr_const_terms] = jacobi_transform(matrix,
const_terms);
    vecN prev_roots(0);
    vecN roots = tr_const_terms;
    size_t k = 0;
    double tr_matrix_c_norm = tr_matrix.continuous_norm();
    double eps_k_coef = 1;

    if (tr_matrix_c_norm < 1.0 - std::numeric_limits<double>::epsilon())
    {
        eps_k_coef = tr_matrix_c_norm / (1 - tr_matrix_c_norm);
    }

    double eps_k = std::numeric_limits<double>::infinity();
    double eps_k_prev = std::numeric_limits<double>::infinity();
    while (eps_k > eps)
    {
        ++k;
        prev_roots = std::move(roots);
        roots = tr_const_terms + tr_matrix * prev_roots;
        eps_k_prev = eps_k;
        eps_k = eps_k_coef * (roots - prev_roots).continuous_norm();
        if (eps_k > eps_k_prev)
        {
            throw std::invalid_argument("The method does not converges
                                        for the system");
        }
    }
}
```

Листинг 11. Метод Зейделя

```
std::pair<vecN, size_t>
```

```
solve_with_seidel(
    matrixNxN const &matrix,
    vecN const &const_terms,
    double eps)
{
    if (eps < 0)
    {
        throw std::invalid_argument("Invalid epsilon");
    }

  auto [tr_matrix, tr_const_terms] = jacobi_transform(matrix,
const_terms);
    matrixNxN b(matrix.size(), 0);
    matrixNxN c(matrix.size(), 0);
    for (size_t i = 0; i < matrix.size(); ++i)
    {
        for (size_t j = 0; j < matrix.size(); ++j)
        {
            if (i < j)
            {
                b[i][j] = tr_matrix[i][j];
            }
            else
            {
                c[i][j] = tr_matrix[i][j];
            }
        }
    }
    matrixNxN tmp = (matrixNxN::identical(matrix.size()) - b).inversed();
    matrixNxN modified_matrix = tmp * c;
    vecN modified_const_terms = tmp * tr_const_terms;
    vecN prev_roots(0);
    vecN roots = tr_const_terms;
    size_t k = 0;

    double modified_matrix_c_norm = modified_matrix.continuous_norm();
    double eps_k_coef = 1;

    if (modified_matrix_c_norm < 1.0 -
std::numeric_limits<double>::epsilon())
    {
        eps_k_coef = c.continuous_norm() / (1 –
modified_matrix.continuous_norm());
    }

    double eps_k = std::numeric_limits<double>::infinity();
    double eps_k_prev = std::numeric_limits<double>::infinity();
    while (eps_k > eps)
    {
        ++k;
        prev_roots = roots;
```

```
        for (size_t i = 0; i < roots.size(); ++i)
        {
            roots[i] = tr_const_terms[i];
            for (size_t j = 0; j < i; ++j)
            {
                roots[i] += tr_matrix[i][j] * roots[j];
            }
            for (size_t j = i; j < roots.size(); ++j)
            {
                roots[i] += tr_matrix[i][j] * prev_roots[j];
            }
        }
        eps_k_prev = eps_k;
        eps_k = eps_k_coef * (roots - prev_roots).continuous_norm();
        if (eps_k > eps_k_prev)
        {
            throw std::invalid_argument("The method does not converges
for the system");
        }
    }
    return std::make_pair(roots, k);
}
```

Листинг 12. Вывод методов простых итераций и Зейделя

```
Matrix
Iterations count: 18
x1 = 1.00000000006697
x2 = 2.00000000004111
x3 = 2.99999999994701
x4 = -1.9999999999451
Solution verified

Iterations count: 11
x1 = 0.99999999965685
x2 = 2.00000000009219
x3 = 2.99999999994026
x4 = -2.00000000002117
Solution verified
```

	e-0	e-1	e-2	e-3	e-4	e-5	e-6	e-7	e-8	e-9	e-10
Iters	5	6	8	10	12	13	15	17	18	20	22
Seidel	2	3	5	5	6	8	8	10	11	12	13

Задача №4

Реализовать метод вращений и степенной метод в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и

собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Матрица:

$$\begin{pmatrix} 5 & 5 & 3 \\ 5 & -4 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

Решение

Метод Якоби — численный метод для нахождения собственных значений симметричной матрицы. Он работает за счет последовательного "обнуления" недиагональных элементов с помощью ортогональных преобразований (вращений). Основная идея заключается в сведении матрицы к диагональной, при этом собственные значения будут на диагонали, а собственные векторы - в соответствующих столбцах.

Процедура включает:

1. Выбор наибольшего по абсолютной величине недиагонального элемента.
2. Построение вращательной матрицы (т.е., выбор угла вращения, который минимизирует данный элемент).
3. Применение этой матрицы для обновления текущей матрицы.
4. Повторение итераций дотех пор, пока все недиагональные элементы не станут меньше заданной точности.

Степенной метод — это итерационный способ приближенного нахождения доминирующего собственного значения матрицы, как правило, абсолютного максимума среди всех ее собственных значений. Метод заключается в последовательном выполнении матричных векторных умножений и нормализации векторов.

Основные этапы:

1. Начать с произвольного вектора и нормализовать его.
2. Выполнить умножение матрицы на вектор.
3. Нормализовать новый вектор.
4. Определить связанное собственное значение и проверять сходимость.
5. Повторять дотех пор, пока разница между собственными значениями не станет меньше заданной точности.

Листинг 13. Метод Якоби

```
std::pair< std::vector< std::pair<double, vecN> >, size_t>
calc_eigen_with_rotation_iterations(
    matrixNxN matrix,
```

```
    double eps)
{
    if (eps < 0) throw std::invalid_argument("Invalid epsilon");
    if (matrix.size() == 0) throw std::invalid_argument("Empty matrix");
    if (matrix.size() == 1)
    {
        return std::make_pair(std::vector<std::pair<double, vecN>>(
            1, std::make_pair(matrix[0][0], vecN(1, 1))), 0);
    }
    if (!matrix.is_symmetric(eps))
         throw std::invalid_argument("Matrix is not symmetric");

    matrixNxN transformation_matrix =
                    matrixNxN::identical(matrix.size());
    size_t iter = 0;
    while (calc_check_sum(matrix) > eps)
    {
        size_t mx_i = 0, mx_j = 1;

        for (size_t i = 1; i < matrix.size(); ++i)
        {
            for (size_t j = 0; j < i; ++j)
            {
                if (std::abs(matrix[i][j]) >
                        std::abs(matrix[mx_i][mx_j]))
                {
                    mx_i = i;
                    mx_j = j;
                }
            }
        }

        double phi = 0.5 * std::atan(2 * matrix[mx_i][mx_j] /
                      (matrix[mx_i][mx_i] - matrix[mx_j][mx_j]));
        matrixNxN rotate_matrix = matrixNxN::identical(matrix.size());
        rotate_matrix[mx_i][mx_i] =  std::cos(phi);
        rotate_matrix[mx_i][mx_j] = -std::sin(phi);
        rotate_matrix[mx_j][mx_i] =  std::sin(phi);
        rotate_matrix[mx_j][mx_j] =  std::cos(phi);
        transformation_matrix *= rotate_matrix;
        matrix = rotate_matrix.transposed() * matrix * rotate_matrix;
        ++iter;
    }
    transformation_matrix = transformation_matrix.transposed();
    std::vector< std::pair<double, vecN> > ans(matrix.size());
    for (size_t i = 0; i < matrix.size(); ++i)
    {
        ans[i] = std::make_pair(matrix[i][i], transformation_matrix[i]);
    }
    return std::make_pair(ans, iter);
}
```

Листинг 14. Степенной метод

```
std::pair< std::pair<double, vecN> , size_t>
calc_eigen_radius_with_power_iterations(
    matrixNxN const &matrix,
    double eps)
{
    if (eps < 0) throw std::invalid_argument("Invalid epsilon");
    if (matrix.size() == 0) throw std::invalid_argument("Empty matrix");
    if (matrix.size() == 1)
    {
        return std::make_pair(std::make_pair(matrix[0][0], vecN(1, 1)),
                              0);
    }

    size_t iter = 0;
    double x, x_prev;
    vecN h(matrix.size(), 1);
    h /= h.continuous_norm();
    do
    {
        double tmp = h[0];
        h = matrix * h;
        x_prev = x;
        x = h[0] / tmp;
        h /= h.continuous_norm();
        ++iter;
    } while (std::abs(x - x_prev) > eps);
    return std::make_pair(std::make_pair(x, h), iter);
}
```

Листинг 15. Вывод метода Якоби и степенного метода

```
Rotation iterations count: 6
x1 = 8.7054674 : h = (0.83032442, 0.36022501, 0.425205)^T
x2 = -6.2393735 : h = (-0.41520774, 0.90880771, 0.040878778)^T
x3 = 0.53390615 : h = (-0.37170403, -0.21049106, 0.90417345)^T
Solution verified

Rotation iterations count: 41
x_rad = 8.7054714 : h = (1, 0.43383595, 0.5120949)^T
Solution verified
```

Лабораторная работа №2

Задача №1

Реализовать методы (простой итерации, дихотомии, Ньютона и секущих) решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

Нелинейное уравнение:

$$\sqrt{1-x^2}-e^x+0{,}1=0\,.$$

Решение

1. Метод дихотомии:

Метод дихотомии — это численный метод для решения уравнений и поиска корней функций, который заключается в делении отрезка пополам и выборе той его части, в которой происходит смена знака функции. Он всегда сходится, если функция непрерывна и пределы начального отрезка выбраны правильно. Начальное приближение определяем графически.

2. Метод простой итерации:

Этот метод, также известный как метод последовательных приближений, основывается на замене функции на простую в приближении итерационную функцию g(x), и повторении этого процесса до тех пор, пока результат не "стабилизируется". Метод сходится, если функция g(x) является сжимающей, то есть ее производная должна быть меньше 1 по модулю.

3. Метод Ньютона:

Также известный как метод касательных, он представляет собой итерационный процесс, использующий производную функции для нахождения корня. При быстрой сходимости он показывает хорошую эффективность, но может не сходиться, если начальное приближение слишком далеко от истинного корня.

4. Метод секущих:

Это численный метод нахождения корней, похожий на метод Ньютона. Он использует линейную аппроксимацию через наклонные прямые (секущие) и не требует вычисления производных, делая его более применимым в ситуациях, где производная найти затруднительно.

Листинг 16. Метод дихотомии

```
std::pair<double, size_t>
solve_equation_with_dichotomy(double (*f)(double), double a, double b,
double eps)
{
    checks::throw_if_invalid_eps(eps);
    checks::throw_if_invalid_interval(a, b, eps);
    size_t iter = 0;
    do
    {
        double mid = (a + b) / 2;
        double a_sign = f(a) > 0 ? 1 : -1;
        double b_sign = f(b) > 0 ? 1 : -1;
        double mid_sign = f(mid) > 0 ? 1 : -1;
        if (a_sign != mid_sign && mid_sign == b_sign)
            b = mid;
        else if (b_sign != mid_sign && mid_sign == a_sign)
            a = mid;
        else if (a_sign != mid_sign && mid_sign != b_sign)
            throw std::invalid_argument("TWO roots on the interval");
        else if (a_sign == mid_sign && mid_sign == b_sign)
            throw std::invalid_argument("NO roots on the interval");
        ++iter;
    } while (b - a > 2 * eps);
    return std::make_pair((a + b) / 2, iter);
}
```

Листинг 17. Метод простой итерации

```
std::pair<double, size_t>
solve_equation_with_iterations(double (*phi)(double), double q, double
x0, double eps)
{
    checks::throw_if_invalid_eps(eps);
    double x = x0;
    double x_prev;
    double eps_mult = q * (1 - q);
    size_t iter = 0;
    do
    {
        x_prev = x;
        x = phi(x);
        ++iter;
    } while (eps_mult * std::abs(x - x_prev) > eps);
    return std::make_pair(x, iter);
}
```

Листинг 18. Метод Ньютона

```
std::pair<double, size_t>
solve_equation_with_newton(
    double (*func)(double),
    double (*func_first_deriv)(double),
```

```
    double (*func_second_deriv)(double),
    double x0, double eps)
{
    checks::throw_if_invalid_eps(eps);
    if (func(x0) * func_second_deriv(x0) <
std::numeric_limits<double>::epsilon())
        throw std::invalid_argument("Newton method will not converge with
                                        the given approximation");
    double x = x0;
    double x_prev;
    size_t iter = 0;
    do
    {
        x_prev = x;
        x -= func(x) / func_first_deriv(x);
        ++iter;
    } while (std::abs(x - x_prev) > eps);
    return std::make_pair(x, iter);
}
```

Листинг 19. Метод секущих

```
std::pair<double, size_t>
solve_equation_with_secant(
    double (*func)(double),
    double (*func_second_deriv)(double),
    double x0, double x1, double eps)
{
    checks::throw_if_invalid_eps(eps);
    if (func(x0) * func_second_deriv(x0) <
                 std::numeric_limits<double>::epsilon() ||
            func(x1) * func_second_deriv(x1) <
                 std::numeric_limits<double>::epsilon())
    {
        throw std::invalid_argument("Newton method will not converge with
                                       the given approximation");
    }
    double x = x1;
    double x_prev = x0;
    size_t iter = 0;
    do
    {
        double tmp = x;
        x -= func(x) * (x - x_prev) / (func(x) - func(x_prev));
        x_prev = tmp;
        ++iter;
    } while (std::abs(x - x_prev) > eps);

    return std::make_pair(x, iter);
}
```

Листинг 20. Вывод методов простой итерации, дихотомии, Ньютона и секущих.

```
          eps = 0.000000000001000
```

```
Iterations: x = 0.091490148635518  (10)
Dichotomy:  x = 0.091490148634875  (39)
Newton:     x = 0.091490148635777  ( 4)
Secant:     x = 0.091490148635777  ( 5)
```

Задача №2

Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

Параметр a = 4. Система уравнений:

$$\begin{cases}(x_1^2 + a^2)x_2 - a^3 = 0,\\ (x_1 - a/2)^2 + (x_2 - a/2)^2 - a^2 = 0.\end{cases}$$

Решение

1. Метод простой итерации:

Этот метод ищет приближенное решение системы нелинейных уравнений $F(x) = 0$ путем последовательного вычисления решений $x^{(k+1)} = g\left(x^{(k)}\right)$. Функция $g(x)$ должна обеспечивать сходимость процесса к искомому решению. Выбор функции $g(x)$ и проверка условий Липшица для сходимости метода — ключевые моменты этого подхода.

2. Метод Ньютона для систем:

Метод Ньютона для систем линейных уравнений использует якобиан системы. Это итеративный метод, который находит корректировки решению, используя градиентный подход. Для каждой итерации вычисляется и решается система линейных уравнений, представленная якобианом, что требует вычисления частных производных функций. Начальное приближение определяется графически.

Листинг 21. Метод простой итерации

```
std::pair<std::pair<double, double>, size_t>
solve_equation_system_with_iterations(
    double (*phi1)(double, double),
    double (*phi2)(double, double),
    double q, double x1_0, double x2_0, double eps)
{
    checks::throw_if_invalid_eps(eps);
```

```
    double x1 = x1_0;
    double x2 = x2_0;
    double x1_prev;
    double x2_prev;
    double eps_mult = q * (1 - q);
    double check;
    size_t iter = 0;
    do
    {
        x1_prev = x1;
        x2_prev = x2;
        double x1_tmp = phi1(x1, x2);
        double x2_tmp = phi2(x1, x2);
        x1 = x1_tmp;
        x2 = x2_tmp;

        check = std::max(std::abs(x1 - x1_prev), std::abs(x2 - x2_prev));
        ++iter;
    } while (eps_mult * check > eps);
    return std::make_pair(std::make_pair(x1, x2), iter);
}
```

Листинг 22. Метод Ньютона

```
std::pair<std::pair<double, double>, size_t>
solve_equation_system_with_newton(
    double (*f1)(double, double),
    double (*f2)(double, double),
    double (*f1_x1_deriv)(double, double),
    double (*f1_x2_deriv)(double, double),
    double (*f2_x1_deriv)(double, double),
    double (*f2_x2_deriv)(double, double),
    double x1_0, double x2_0, double eps)
{
    checks::throw_if_invalid_eps(eps);
    double x1 = x1_0;
    double x2 = x2_0;
    double x1_prev;
    double x2_prev;
    double check;
    size_t iter = 0;
    do
    {
        x1_prev = x1;
        x2_prev = x2;
        matrixNxN coefs(2);
        vecN constant_terms(2);
        coefs[0][0] = f1_x1_deriv(x1, x2);
        coefs[0][1] = f1_x2_deriv(x1, x2);
        coefs[1][0] = f2_x1_deriv(x1, x2);
        coefs[1][1] = f2_x2_deriv(x1, x2);
        constant_terms[0] = -f1(x1, x2);
```

```
        constant_terms[1] = -f2(x1, x2);

        vecN delta = algorithms::solve_linear_equation_system(coefs,
                                                              constant_terms);
        x1 += delta[0];
        x2 += delta[1];
        check = std::max(std::abs(x1 - x1_prev), std::abs(x2 - x2_prev));
        ++iter;
    } while (check > eps);
    return std::make_pair(std::make_pair(x1, x2), iter);
}
```

Листинг 23. Вывод метода Ньютона и простых итераций

```
           eps = 0.000000000001000  0.000000000001000
Iterations: x = (5.929262756906704, 1.251071307010412)
Newton:     x = (5.929262756907046, 1.251071307010313)
```

Лабораторная работа №3

Задача №1

Используя таблицу значений Y_i функции $y = f(x)$, вычисленных в точках $X_i, i = 0, \dots, 3$ построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки $\{X_i, Y_i\}$. Вычислить значения погрешности интерполяции в точке X^*.

Функция:

$y = tg(x)$; а) $X_i = 0, \frac{\pi}{8}, \frac{2\pi}{8}, \frac{3\pi}{8}$; б) $X_i = 0, \frac{\pi}{8}, \frac{\pi}{3}, \frac{3\pi}{8}$; $X^* = \frac{\pi}{4}$

Решение

Интерполяция — это метод нахождения новых точек данных в пределах диапазона дискретного набора известных точек. В контексте использования многочленов, интерполяция строит многочлен, который проходит через заданные точки.

Многочлен Лагранжа имеет форму:

$P(x) = \sum_{i=0}^{n} y_i L_i(x)$,

где $L_i(x)$ – это базисные многочлены Лагранжа, вычисляемые как

$L_i(x) = \prod_{j=0, j\neq i}^{n} \left(\frac{x - x_j}{x_i - x_j}\right)$

Многочлены Ньютона строятся на основе разделенных разностей.

Они имеют такую форму:

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots +$$
$$+ a_n(x - x_0)(x - x_1)\dots(x - x_{n-1})$$

Погрешность интерполяции для многочленов степени n может быть оценена с помощью формулы:

$$E(x) = \frac{(f^{((n+1)}(\xi))}{(n+1)!} \prod_{i=0}^{n}(x - x_i)$$

где ξ — некоторая точка внутри интервала интерполяции.

Листинг 24. интерполяционный многочлен Лагранжа

```
polynomial algorithms::interpolate_with_lagrange(
    std::vector<double> const &points,
    std::vector<double> const &values,
    bool trace_flag,
    size_t trace_precision)
{
    if (points.size() != values.size())
        throw std::invalid_argument("Point and values count arnt equal");
```

```
    if (points.size() == 0) return polynomial();

    polynomial interpolation;
    for (size_t i = 0; i < points.size(); ++i)
    {
        polynomial li({1});
        for (size_t j = 0; j < points.size(); ++j)
        {
            if (i == j) continue;
            li *= polynomial({-points[j], 1});
            li /= points[i] - points[j];
        }
        interpolation += li * values[i];
    }
    if (trace_flag) {
        std::ios_base::fmtflags former_flags = std::cout.flags();
        size_t former_precision = std::cout.precision();
        std::cout.precision(trace_precision);
        std::cout.setf(std::cout.fixed);
        std::cout << "TRACE: ";
        for (size_t i = 0; i < points.size(); ++i)
        {
            if (i > 0 && values[i] > 0) std::cout << "+";
            double divisor = 1;
            for (size_t j = 0; j < points.size(); ++j)
                divisor *= (i == j) ? 1 : (points[i] - points[j]);

            std::cout << values[i] / divisor;

            for (size_t j = 0; j < points.size(); ++j)
            {
                if (i == j) continue;
                std::cout << "(x" << (-points[j] > 0 ? "+" : "") <<
                             -points[j] << ")";
            }
        }
        std::cout << std::endl;
        std::cout.flags(former_flags);
        std::cout.precision(former_precision);
    }
    return interpolation;
}
```

Листинг 25. интерполяционный многочлен Ньютона

```
polynomial algorithms::interpolate_with_newton(
    std::vector<double> const &points,
    std::vector<double> const &values,
    bool trace_flag,
    size_t trace_precision)
{
    if (points.size() != values.size())
```

```
    {
        throw std::invalid_argument("Point and values count arnt equal");
    }
    if (points.size() == 0) return polynomial();

    std::vector<std::vector<double>> div_diff(points.size(),
                                      std::vector<double>(points.size()));
    for (size_t i = 0; i < points.size(); ++i)
        div_diff[i][0] = values[i];

    for (size_t j = 1; j < points.size(); ++j)
    {
        for (size_t i = 0; i+j < points.size(); ++i)
        {
            div_diff[i][j] = (div_diff[i+1][j-1] - div_diff[i][j-1]) /
                             (points[i+j] - points[i]);
        }
    }
    polynomial interpolation, mult({1});
    if (trace_flag) {
        std::ios_base::fmtflags former_flags = std::cout.flags();
        size_t former_precision = std::cout.precision();
        std::cout.precision(trace_precision);
        std::cout.setf(std::cout.fixed);
        std::cout << "TRACE: ";
        for (size_t i = 0; i < points.size(); ++i)
        {
            if (div_diff[0][i] > 0) std::cout << "+";
            std::cout << div_diff[0][i];

            for (size_t j = 0; j < i; ++j)
            {
                std::cout << "(x" << (-points[j] > 0 ? "+" : "") <<
                             -points[j] << ")";
            }
        }
        std::cout << std::endl;
        std::cout.flags(former_flags);
        std::cout.precision(former_precision);
    }
    for (size_t i = 0; i < points.size(); ++i)
    {
        interpolation += mult * div_diff[0][i];
        mult *= polynomial({-points[i], 1});
    }
    return interpolation;
}
```

Листинг 26. Вывод интерполяционных многочленов

```
INTERPOLATION A
Points : 0.000000 0.392699 0.785398 1.178097
```

```
Values : 0.000000 0.414214 1.000000 2.414214

TRACE: -0.000(x-0.393)(x-0.785)(x-1.178)+3.420(x-0.000)(x-0.785)(x-
1.178)+-8.256(x-0.000)(x-0.393)(x-1.178)+6.644(x-0.000)(x-0.393)(x-0.785)
Lagrange P(x) = 1.808x^3 - 1.573x^2 + 1.394x

TRACE: 0.000+1.055(x-0.000)+0.556(x-0.000)(x-0.393)+1.808(x-0.000)(x-
0.393)(x-0.785)
Newton P(x) = 1.808x^3 - 1.573x^2 + 1.394x

tan(x*)      = 0.668179
newton P(x*) = 0.644607
error        = 0.023572


INTERPOLATION B
Points : 0.000000 0.392699 1.047198 1.178097
Values : 0.000000 0.414214 1.732051 2.414214

TRACE: -0.000(x-0.393)(x-1.047)(x-1.178)+2.052(x-0.000)(x-1.047)(x-
1.178)+-19.306(x-0.000)(x-0.393)(x-1.178)+19.933(x-0.000)(x-0.393)(x-
1.047)
Lagrange P(x) = 2.679x^3 - 2.942x^2 + 1.797x

TRACE: 0.000+1.055(x-0.000)+0.916(x-0.000)(x-0.393)+2.679(x-0.000)(x-
0.393)(x-1.047)
Newton P(x) = 2.679x^3 - 2.942x^2 + 1.797x

tan(x*)      = 0.668179
Newton P(x*) = 0.585251
error        = 0.082928
```

Задача №2

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при x=x0 и x=x4. Вычислить значение функции в точке x=X*. Построить график.

Данные: X*=1.5

i	0	1	2	3	4
x_i	0.0	0.9	1.8	2.7	3.6
f_i	0.0	0.36892	0.85408	1.7856	6.3138

Решение

Кубический сплайн — это набор соединенных гладких полиномов третьей степени, используемый для интерполяции данных. Они создаются так, чтобы корректно соединять каждый кусочный полином на основе заданных данных.

Особенности кубического сплайна:

1. Каждый сегмент сплайна между двумя соседними узлами описывается кубическим многочленом:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

2. Требуются условия гладкости: непрерывность функции и ее первой и второй производных.

3. В задаче предполагается, что сплайн имеет нулевую кривизну на концах, что соответствует условиям $S''(x_0) = 0, \quad S''(x_4) = 0$

Листинг 27. Кубический сплайн

```
piecewise_polynomial algorithms::build_spline(
    std::vector<double> const &points,
    std::vector<double> const &values)
{
    if (points.size() != values.size())
        throw std::invalid_argument("Point and values count arnt equal");
    if (points.size() == 0) return piecewise_polynomial();

    std::vector<std::pair<double, double>> pv(points.size());
    for (size_t i = 0; i < pv.size(); ++i)
    {
        pv[i].first = points[i];
        pv[i].second = values[i];
    }
    std::sort(pv.begin(), pv.end(),
            [](std::pair<double, double> a, std::pair<double, double> b)
            {
                return a.first < b.first;
            });

    std::vector<double> h(points.size() - 1);
    matrixNxN eq_coefs(points.size() - 2);
    vecN eq_const_terms(points.size() - 2);
    for (size_t i = 0; i < h.size(); ++i)
    {
        h[i] = pv[i+1].first - pv[i].first;
    }
    for (size_t i = 0; i < eq_coefs.size(); ++i)
    {
        if (i > 0) eq_coefs[i][i-1] = h[i];
        eq_coefs[i][i] = 2 * (h[i] + h[i+1]);

        if (i + 1 < eq_coefs.size()) eq_coefs[i][i+1] = h[i+1];

        eq_const_terms[i] = 3*((pv[i+2].second - pv[i+1].second) / h[i+1]
                               - (pv[i+1].second - pv[i].second) / h[i]);
    }
    vecN roots = solve_tridiagonal_linear_equation_system(eq_coefs,
                                                          eq_const_terms);
```

```
    std::vector<double> c(points.size() - 1, 0);
    for (size_t i = 0; i < roots.size(); ++i) c[i+1] = roots[i];

    piecewise_polynomial spline;
    for (size_t i = 0; i < c.size(); ++i)
    {
        double a = pv[i].second;
        double b = (pv[i+1].second - pv[i].second) / h[i] -
                (i + 1 < c.size()
                ? h[i] * (c[i+1] + 2*c[i]) / 3
                : 2 * h[i] * c[i] / 3);
        double d = ((i + 1 < c.size() ? c[i+1] : 0) - c[i]) / (3 * h[i]);

        polynomial p({-pv[i].first, 1});
        polynomial q = polynomial({a}) + p*b + p*p*c[i] + p*p*p*d;
        if (i == 0)
            spline = piecewise_polynomial(pv[i].first, pv[i+1].first, q);
        else
            spline.push_back(pv[i+1].first, q);
    }
    return spline;
}
```

Листинг 28. Вывод кубического сплайна

```
0.000 <= x <= 0.900 :  0.087077x^3 + 1.339379x
0.900 <  x <= 1.800 : -0.275935x^3 + 0.980132x^2 - 0.542740x + 0.264636
1.800 <  x <= 2.700 :  1.469501x^3 - 8.445222x^2 + 16.422898x - 9.914747
2.700 <  x <= 3.600 : -1.280644x^3 + 13.830952x^2 - 43.722774x + 44.21635

F(1.5) = 0.724542857142857
```

Для построения графиков следует добавить точки, данные из задачи, и ввести все функции (рис. 1).

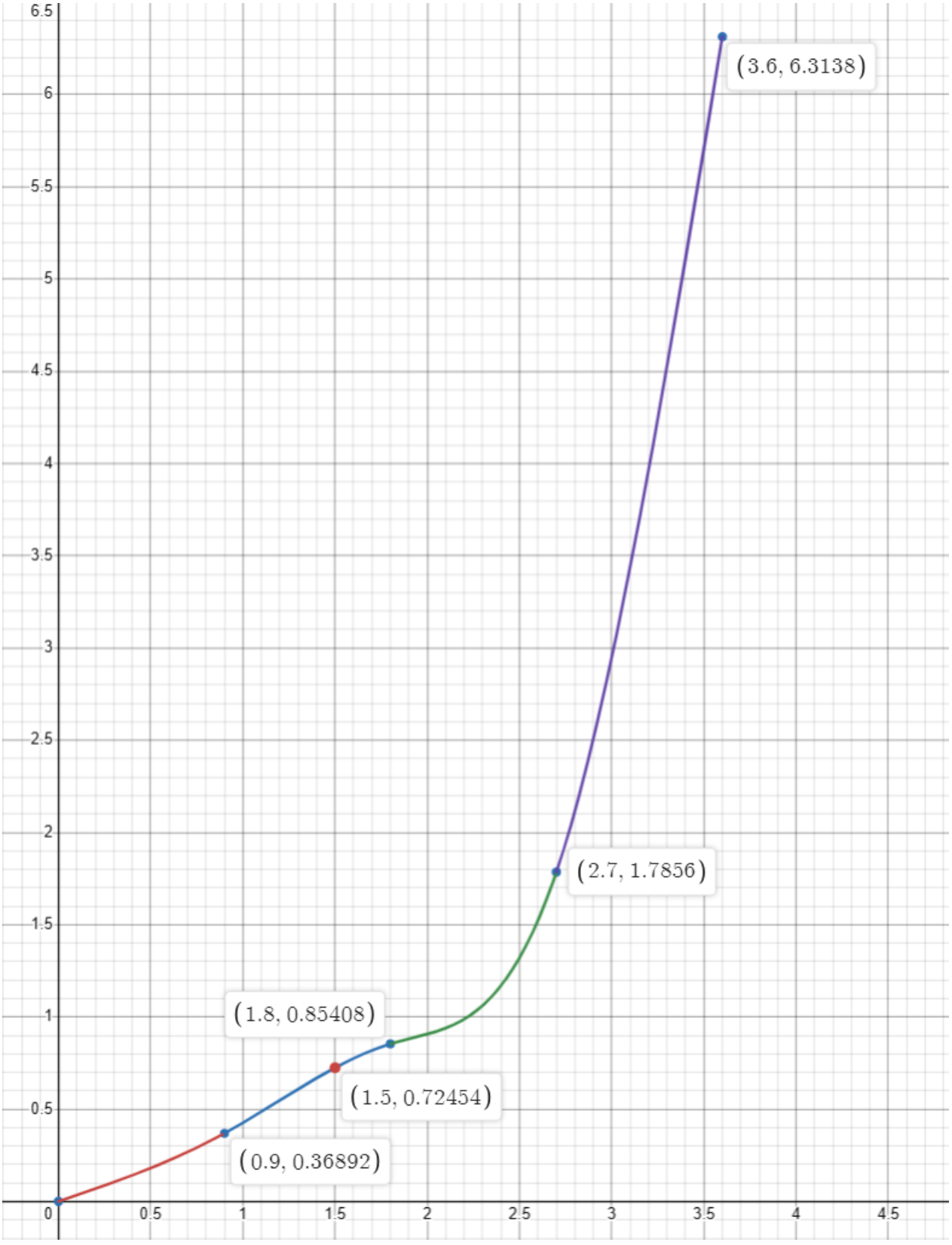

Рисунок 1. График кубического сплайна

Задача №3

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

Данные:

i	0	1	2	3	4	5
x_i	-0.9	0.0	0.9	1.8	2.7	3.6
y_i	-0.36892	0.0	0.36892	0.85408	1.7856	6.3138

Решение

Метод наименьших квадратов (МНК) — это метод описания зависимости между переменными с использованием аппроксимирующих функций. Его целью является минимизация суммы квадратов отклонений наблюдаемых значений от значений, предсказанных моделью. В контексте полиномиальной аппроксимации, мы пытаемся найти полином, который "наилучшим образом" (в терминах наименьших квадратов) приблизит заданную функцию.

Основные шаги:

1. Построение системы линейных уравнений: для каждого коэффициента многочлена составляются уравнения. Эти уравнения образуются путем суммирования произведений заданных данных и соответствующих степеней x.
2. Решение нормальной системы: обычно для этого используется метод Гаусса или LU-разложение. Он позволяет найти коэффициенты аппроксимирующего многочлена.
3. Оценка качества аппроксимации: исчисляется сумма квадратов ошибок, представляющая собой сумму квадратов разностей между фактическими и предсказанными значениями.

Листинг 29. МНК заданной степени

```
polynomial algorithms::approximate_with_least_squares_method(
    std::vector<double> const &points,
    std::vector<double> const &values,
    size_t degree)
{
    if (points.size() != values.size())
        throw std::invalid_argument("Point and values count arnt equal");
    if (points.size() == 0) return polynomial();

    matrixNxN coefs(degree + 1, 0);
    vecN constant_terms(degree + 1, 0);
    std::vector<double> powered_points(points.size(), 1);
    std::vector<double> powered_right_side(values);
    for (size_t i = 0; i <= degree; ++i)
```

```
    {
        for (size_t j = 0; j < points.size(); ++j)
        {
            coefs[0][i] += powered_points[j];
            powered_points[j] *= points[j];
        }
    }
    for (size_t i = 0; i < points.size(); ++i)
    {
        constant_terms[0] += powered_right_side[i];
        powered_right_side[i] *= points[i];
    }
    for (size_t k = 1; k <= degree; ++k)
    {
        for (size_t i = 0; i < degree; ++i)
        {
            coefs[k][i] = coefs[k-1][i+1];
        }
        for (size_t i = 0; i < points.size(); ++i)
        {
            coefs[k][degree] += powered_points[i];
            powered_points[i] *= points[i];

            constant_terms[k] += powered_right_side[i];
            powered_right_side[i] *= points[i];
        }
    }
    vecN roots = solve_linear_equation_system(coefs, constant_terms);
    std::vector<double> polynomial_coefs(degree + 1);
    for (size_t i = 0; i <= degree; ++i)
    {
        polynomial_coefs[i] = roots[i];
    }
    return polynomial(polynomial_coefs);
}
```

Листинг 30. Вывод МНК

```
P1(x) = 1.2462x - 0.1901
Error = 8.679022

P2(x) = 0.5081x^2 - 0.1256x - 0.4645
Error = 2.355749
```

Для построения графика следует взять данные задачи и данные результата (рис. 2).

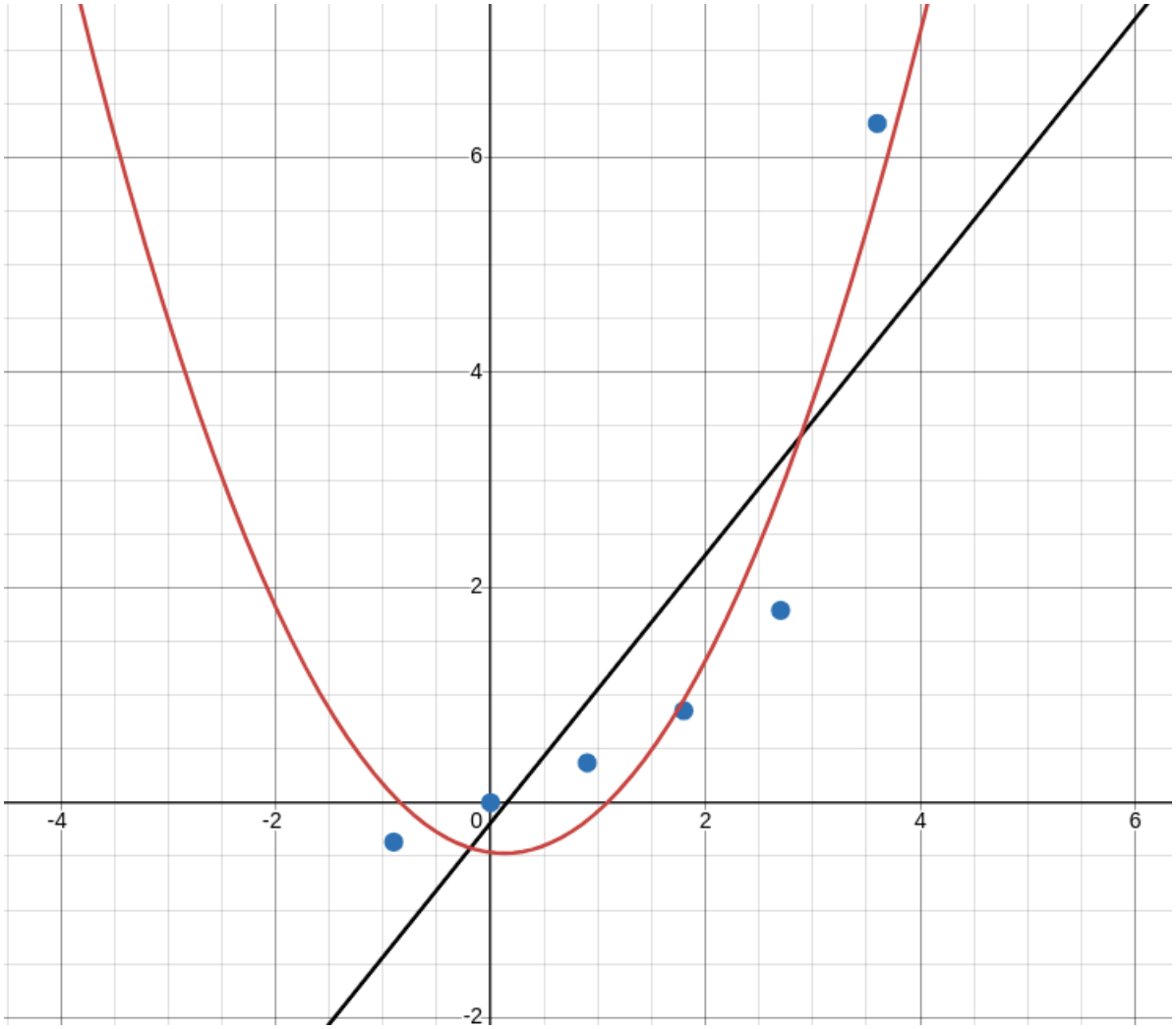

Рисунок 2. График приближающих многочленов

Задача №4

Вычислить первую и вторую производную от таблично заданной функции yi=f(xi), i=0,1,2,3,4 в точке x=X*.

Данные:

X*=2.0

i	0	1	2	3	4
x_i	1.0	1.5	2.0	2.5	3.0
y_i	0.0	0.40547	0.69315	0.91629	1.0986

Решение

Численное дифференцирование — это техника приближенного нахождения производных функции, основанная на известных данных. Когда у нас есть лишь табличные данные, мы можем использовать методы конечных разностей для приближенного вычисления производных.

1. Аппроксимация первой производной первого порядка точности:
$y'(x) = \frac{y_{i+1}-y_i}{x_{i+1}-x_i},\ x \in [x_i, x_{i+1}]$
2. Аппроксимация первой производной первого порядка точности:
$y'(x) = \frac{y_{i+1}-y_i}{x_{i+1}-x_i} + \frac{\frac{y_{i+2}-y_{i+1}}{x_{i+2}-x_{i+1}} - \frac{y_{i+1}-y_i}{x_{i+1}-x_i}}{x_{i+2}-x_i}(2x - x_i - x_{i+1}),\ x \in [x_i, x_{i+1}]$
3. Аппроксимация второй производной первого порядка точности:
$y'(x) = 2\frac{\frac{y_{i+2}-y_{i+1}}{x_{i+2}-x_{i+1}} - \frac{y_{i+1}-y_i}{x_{i+1}-x_i}}{x_{i+2}-x_i},\ x \in [x_i, x_{i+1}]$

Листинг 31. Первая производная первого порядка точности

```
piecewise_polynomial calc_first_derivative1(
    std::vector<double> const &points,
    std::vector<double> const &values)
{
    if (points.size() != values.size())
        throw std::invalid_argument("Point and values count arnt equal");
    if (points.size() < 2)
        return piecewise_polynomial(0, 0, polynomial({0}));

    piecewise_polynomial deriv(points[0], points[1], polynomial(
            {(values[1] - values[0]) / (points[1] - points[0])}));

    for (size_t i = 2; i < points.size(); ++i)
    {
        deriv.push_back(points[i], polynomial(
                {(values[i] - values[i-1]) /
                 (points[i] - points[i-1])}));
    }
    return deriv;
}
```

Листинг 32. Первая производная второго порядка точности

```
piecewise_polynomial calc_first_derivative2(
    std::vector<double> const &points,
    std::vector<double> const &values)
{
    if (points.size() != values.size())
        throw std::invalid_argument("Point and values count arnt equal");
    if (points.size() < 3)
        return piecewise_polynomial(0, 0, polynomial({0}));

    piecewise_polynomial deriv;

    for (size_t i = 2; i < points.size(); ++i)
    {
        double coef_A = (values[i  ] - values[i-1]) /
             (points[i] - points[i-1]);
```

```
        double coef_B = (values[i-1] - values[i-2]) /
              (points[i-1] - points[i-2]);
        double coef_C = points[i] - points[i-2];
        double coef = (coef_A - coef_B) / coef_C;

        polynomial p({-points[i-2] - points[i-1], 2});
        p *= polynomial({coef});
        p += polynomial({coef_B});
        if (i == 2)
            deriv = piecewise_polynomial(points[i-2], points[i-1], p);
        else
            deriv.push_back(points[i-1], p);
    }
    return deriv;
}
```

Листинг 33. Вторая производная первого порядка точности

```
piecewise_polynomial calc_second_derivative(
    std::vector<double> const &points,
    std::vector<double> const &values)
{
    if (points.size() != values.size())
        throw std::invalid_argument("Point and values count arnt equal");
    if (points.size() < 3)
        return piecewise_polynomial(0, 0, polynomial({0}));

    piecewise_polynomial deriv(points[0], points[1], polynomial(
            {(values[1] - values[0]) / (points[1] - points[0])}));

    for (size_t i = 2; i < points.size(); ++i)
    {
        double coef_A = (values[i  ] - values[i-1]) / (points[i  ] -
points[i-1]);
        double coef_B = (values[i-1] - values[i-2]) / (points[i-1] -
points[i-2]);
        double coef_C = points[i] - points[i-2];
        double coef = (coef_A - coef_B) / coef_C;
        polynomial p({2 * coef});

        if (i == 2)
            deriv = piecewise_polynomial(points[i-2], points[i-1], p);
        else
            deriv.push_back(points[i-1], p);
    }
    return deriv;
}
```

Листинг 34. Вывод производных

```
1th-order accurate first derivative
1.0 <= x <= 1.5 :  0.810940
1.5 <  x <= 2.0 :  0.575360
2.0 <  x <= 2.5 :  0.446280
```

```
2.5 <  x <= 3.0 :  0.364620
F(X*) = 0.575360 / 0.446280

2th-order accurate first derivative
1.0 <= x <= 1.5 : -0.471160x + 1.399890
1.5 <  x <= 2.0 : -0.258160x + 1.027140
2.0 <  x <= 2.5 : -0.163320x + 0.813750
F(X*) = 0.510820 / 0.487110

1th-order accurate second derivative
1.0 <= x <= 1.5 : -0.471160
1.5 <  x <= 2.0 : -0.258160
2.0 <  x <= 2.5 : -0.163320
F(X*) = -0.258160 / -0.163320
```

Задача №5

Вычислить определенный интеграл методами прямоугольников, трапеций, Симпсона с шагами h1, h2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга.

Данные:

$$y = \frac{x}{(3x+4)^3} \qquad X_0 = -1, \quad X_k = 1, \quad h_1 = 0.5, \quad h_2 = 0.25$$

Решение

Численное интегрирование применяется для приближения значений определенных интегралов, когда аналитическое решение затруднительно или невозможно. Рассмотрим несколько методов численного интегрирования, используемых в коде.

Метод прямоугольников — один из простейших методов численного интегрирования. В нем интервал интегрирования делится на равные части, и в каждой части значение функции приближается к значению в середине интервала (метод средних прямоугольников).

Формула для вычисления интеграла методом прямоугольников:

$$I \approx h \cdot \sum_{i=1}^{n} f(x_i)$$

где h — ширина каждого подынтервала, а x_i — середина i-го интервала.

Метод трапеций использует линейную интерполяцию для приближения площади под кривой. Интервал интегрирования делится на части, где каждая часть представляется трапецией. Формула для метода трапеций:

$$I \approx \frac{h}{2}(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b))$$

Метод Симпсона использует параболическое интерполирование и работает с четным числом интервалов. Формула метода Симпсона:

$$I \approx \frac{h}{3}(f(a) + 4 \sum_{i(четные)} f(x_i) + 2 \sum_{i(нечетные)} (f(x_i) + f(b))$$

Метод Рунге-Ромберга позволяет получать более высокий порядок точности вычисления, если имеются результаты вычисления интеграла на двух сетках. Формула:

$$F = F_h + \frac{F_h - F_{kh}}{k^p - 1} + O(h^{p+1})$$

где F_h и F_{kh} — интегралы, посчитанные при значениях шага h и kh, p — порядок метода (2 для прямоугольников и трапеций, 4 для Симпсона).

Листинг 35. Метод прямоугольников

```
double integrate_with_rectangle_method(double (func)(double), double a,
                                       double b, double h)
{
    size_t n = std::round((b - a) / h);
    double sum = 0;
    for (size_t i = 0; i < n; ++i)
    {
        double l = a + h*i;
        double r = a + h*(i+1);
        sum += func((l + r) / 2);
    }
    return sum * h;
}
```

Листинг 36. Метод трапеций

```
double integrate_with_trapezium_method(double (func)(double), double a,
                                       double b, double h)
{
    size_t n = std::round((b - a) / h);
    double sum = 0;
    for (size_t i = 0; i < n; ++i)
    {
        double l = a + h*i;
        double r = a + h*(i+1);
        sum += func(l) + func(r);
    }
    return sum * h / 2;
}
```

Листинг 37. Метод Симпсона

```
double integrate_with_simpson_method(double (func)(double), double a,
                                     double b, double h)
{
    size_t n = std::round((b - a) / h);
    double sum = 0;
    for (size_t i = 0; i < n; i += 2)
```

```
    {
        double l = a + h*i;
        double m = a + h*(i+1);
        double r = a + h*(i+2);
        sum += func(l) + 4*func(m) + func(r);
    }
    return sum * h / 3;
}
```

Листинг 38. Уточнение интеграла методом Рунге-Ромберга

```
double integrate_with_rrr_method(double f_h, double f_kh, double k,
double p)
{
    return f_h + (f_h - f_kh) / (std::pow(k, p) - 1);
}
```

Листинг 39. Вывод итегралов

```
METHOD         VALUE      ERROR
h = 0.500000
Rectangle  -0.070910
Trapezium  -0.263769
Simpson    -0.185511
h = 0.250000
Rectangle  -0.102439
Trapezium  -0.167339
Simpson    -0.135196
RRR improvement
Rectangle  -0.112949  0.009500
Trapezium  -0.135196  0.012747
Simpson    -0.118425  0.004024
```

Лабораторная работа №4

Задача №1

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки h. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге-Ромберга и путем сравнения с точным решением.

Задача Коши:

$y''-2y-4x^2e^{x^2}=0,$

$y(0)=3,$

$y'(0)=0,$

$x\in[0,1], h=0.1$

Точное решение:

$y=e^{x^2}+e^{x\sqrt{2}}+e^{-x\sqrt{2}}$

Решение

Метод Эйлера является одним из простейших численных методов для решения дифференциальных уравнений. Он основан на идее аппроксимации кривой касательной линией, используя простой разностный подход. Это дает первое приближение для начальной задачи Коши:

$y_{n+1}=y_n+hf(x_n,y_n)$

где h — это шаг сетки, y_n и y_{n+1} – соответствующие значения функции в точках x_n и x_{n+1}.

Метод Рунге-Кутты 4-го порядка один из самых широко используемых методов для ОДУ благодаря своей точности и эффективности. Он вычисляет промежуточные оценки для уклона функции, что улучшает аппроксимацию:

$k_1=f(x_n,y_n),$

$k_2=f(x_n+(h)/2,y_n+(h)/2\cdot k_1),$

$k_3=f(x_n+(h)/2,y_n+(h)/2\cdot k_2),$

$k_4=f(x_n+h,y_n+h\cdot k_3),$

$y_{n+1}=y_n+(h)/6(k_1+2k_2+2k_3+k_4).$

Метод Адамса относится к многошаговым методам, которые используют информацию о решениях на нескольких предыдущих шагах для вычисления

следующего значения. Обычно начинает с использования другого метода для начальных значений.

Метод Рунге-Ромберга — это способ оценки погрешности численных решений, использующий два решения с различными шагами сетки. Разница между ними помогает оценить истинную ошибку.

Листинг 40. Метод Эйлера

```
std::vector<std::tuple<double, double, double>>
solve_cauchy_with_euler(
    double a, double b, double h,
    double y0, double z0,
    std::pair<double, double> (func)(double, double, double))
{
    size_t n = std::round((b - a) / h) + 1;
    std::vector<double> vx(n, a);
    std::vector<double> vy(n, y0);
    std::vector<double> vz(n, z0);
    for (size_t i = 1; i < n; ++i)
    {
        auto p = func(vx[i-1], vy[i-1], vz[i-1]);
        vx[i] = a + i*h;
        vy[i] = vy[i-1] + h * p.first;
        vz[i] = vz[i-1] + h * p.second;
    }

    std::vector<std::tuple<double, double, double>> pv(n);
    for (size_t i = 0; i < n; ++i)
    {
        pv[i] = std::make_tuple(vx[i], vy[i], vz[i]);
    }
    return pv;
}
```

Листинг 41. Метод Рунге-Кутты 4-го порядка

```
std::vector<std::tuple<double, double, double>>
solve_cauchy_with_runge_kutta4(
    double a, double b, double h,
    double y0, double z0,
    std::pair<double, double> (func)(double, double, double))
{
    size_t n = std::round((b - a) / h) + 1;
    std::vector<double> vx(n, a);
    std::vector<double> vy(n, y0);
    std::vector<double> vz(n, z0);
    for (size_t i = 1; i < n; ++i)
    {
        double const &x = vx[i-1];
        double const &y = vy[i-1];
        double const &z = vz[i-1];
```

```
        auto [k1, l1] = func(x, y, z);
        auto [k2, l2] = func(x + h/2, y + h*k1/2, z + h*l1/2);
        auto [k3, l3] = func(x + h/2, y + h*k2/2, z + h*l2/2);
        auto [k4, l4] = func(x + h, y + h*k3, z + h*l3);
        vx[i] = a + i*h;
        vy[i] = vy[i-1] + h/6 * (k1 + 2*k2 + 2*k3 + k4);
        vz[i] = vz[i-1] + h/6 * (l1 + 2*l2 + 2*l3 + l4);
    }
    std::vector<std::tuple<double, double, double>> pv(n);
    for (size_t i = 0; i < n; ++i)
        pv[i] = std::make_tuple(vx[i], vy[i], vz[i]);
    return pv;
}
```

Листинг 42. Метод Адамса

```
std::vector<std::tuple<double, double, double>>
solve_cauchy_with_adams4(
    double a, double b, double h,
    double y0, double y1, double y2, double y3,
    double z0, double z1, double z2, double z3,
    std::pair<double, double> (func)(double, double, double))
{
    size_t n = std::round((b - a) / h) + 1;
    std::vector<double> vx(n, a);
    std::vector<double> vy = { y0, y1, y2, y3 };
    std::vector<double> vz = { z0, z1, z2, z3 };
    vy.resize(n, 0);
    vz.resize(n, 0);
    for (size_t i = 1; i < 4; ++i) vx[i] = a + i*h;
    for (size_t i = 4; i < n; ++i)
    {
        auto [f0, g0] = func(vx[i-1], vy[i-1], vz[i-1]);
        auto [f1, g1] = func(vx[i-2], vy[i-2], vz[i-2]);
        auto [f2, g2] = func(vx[i-3], vy[i-3], vz[i-3]);
        auto [f3, g3] = func(vx[i-4], vy[i-4], vz[i-4]);
        vx[i] = a + i*h;
        vy[i] = vy[i-1] + h/24 * (55*f0 - 59*f1 + 37*f2 - 9*f3);
        vz[i] = vz[i-1] + h/24 * (55*g0 - 59*g1 + 37*g2 - 9*g3);
    }
    std::vector<std::tuple<double, double, double>> pv(n);
    for (size_t i = 0; i < n; ++i)
        pv[i] = std::make_tuple(vx[i], vy[i], vz[i]);
    return pv;
}
```

Листинг 43. Метод Рунге-Ромберга

```
double calc_error_with_rrr(double yh, double y2h, double p)
{
    return std::abs((yh - y2h) / (std::pow(2, p) - 1));
}
```

Листинг 44. Вывод методов Эйлера, Рунге-Кутты и Адамса

```
  x     exact      euler  exact error  rrr error
0.0  3.000000   3.000000     0.000000   0.000000
0.1  3.030084   3.000000     0.030084   0.005000
0.2  3.121346   3.060000     0.061346   0.010075
0.3  3.276891   3.180404     0.096487   0.015667
0.4  3.502136   3.363673     0.138462   0.022247
0.5  3.805209   3.614490     0.190719   0.030353
0.6  4.197580   3.940090     0.257490   0.040634
0.7  4.695010   4.350821     0.344190   0.053913
0.8  5.318975   4.860993     0.457982   0.071268
0.9  6.098766   5.490175     0.608591   0.094154
1.0  7.074649   6.265126     0.809523   0.124580
  x     exact      runge  exact error  rrr error
0.0  3.000000   3.000000     0.000000   0.000000
0.1  3.030084   3.030083     0.000000   0.000000
0.2  3.121346   3.121345     0.000001   0.000000
0.3  3.276891   3.276888     0.000002   0.000000
0.4  3.502136   3.502131     0.000004   0.000000
0.5  3.805209   3.805201     0.000008   0.000000
0.6  4.197580   4.197567     0.000012   0.000001
0.7  4.695010   4.694992     0.000019   0.000001
0.8  5.318975   5.318947     0.000028   0.000002
0.9  6.098766   6.098725     0.000040   0.000003
1.0  7.074649   7.074591     0.000058   0.000004
  x     exact      adams  exact error  rrr error
0.0  3.000000   3.000000     0.000000   0.000000
0.1  3.030084   3.030083     0.000000   0.000000
0.2  3.121346   3.121345     0.000001   0.000000
0.3  3.276891   3.276888     0.000002   0.000001
0.4  3.502136   3.502031     0.000105   0.000005
0.5  3.805209   3.804795     0.000414   0.000024
0.6  4.197580   4.196721     0.000859   0.000051
0.7  4.695010   4.693408     0.001602   0.000096
0.8  5.318975   5.316225     0.002750   0.000165
0.9  6.098766   6.094234     0.004532   0.000274
1.0  7.074649   7.067381     0.007268   0.000440
```

Расчёт параметров стационарного потока идеального газа в канале переменного сечения

При различных начальных данных рассчитать параметры стационарного потока идеального газа в канале переменного сечения.

Решение

Параметры стационарного потока газа описываются уравнением скорости потока, которое в равновесном случае принимает вид

$$\left[M^2 - 1\right]\frac{du}{dx} = u\frac{F'}{F}$$

где M – число Маха, а – скорость звука в данных условиях.

$$M = u/a$$

$$\text{а} = \sqrt{\text{æ}\frac{P}{\rho}}$$

Из уравнения скорости потока следуют четыре случая

1. Если $M < 1,\ F' < 0$, то $\frac{du}{dx} > 0$, т.е. дозвуковой поток в сужающемся канале ускоряется;
2. Если $M < 1, \quad F' > 0,$ то $\frac{du}{dx} < 0,$ т.е. дозвуковой поток в расширяющемся канале замедляется;
3. Если $M > 1, \quad F' < 0,$ то $\frac{du}{dx} < 0,$ т.е. сверхзвуковой поток в сужающемся канале замедляется;
4. Если $M > 1, \quad F' > 0,$ то $\frac{du}{dx} > 0,$ т.е. сверхзвуковой поток в расширяющемся канале ускоряется.

Из уравнения скорости потока и законов сохранения следует следующая система дифференциальных уравнений

$$\begin{cases} \dfrac{du}{dx} = \dfrac{uF'}{(M^2-1)F} \\ \dfrac{dP}{dx} = \dfrac{\rho u^2 F'}{(M^2-1)F} \\ \dfrac{d\rho}{dx} = \dfrac{M^2 \rho F'}{(M^2-1)F} \end{cases}$$

Систему таких дифференциальных уравнений можно решить, например, методом Рунге-Кутта 4 порядка. Метод формулируется так: пусть задано ОДУ

вида $y' = f(x, y)$ с начальным условием $y_0 = y(x_0)$, и необходимо вычислить значение y на интервале $[x_0; x_1]$. тогда, если выбрать шаг h, т.е. расстояние между соседними точками, приближённое значение в следующих точках будет вычисляется по следующей итерационной формуле

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Коэффициенты k_1, k_2, k_3, k_4 вычисляются так

$$k_1 = f(x_n, y_n)$$
$$k_2 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right)$$
$$k_3 = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right)$$
$$k_4 = f(x_n + h, y_n + hk_3)$$

Ошибка на каждом шаге вычисления имеет порядок $O(h^5)$, тогда суммарная ошибка будет иметь порядок $O(h^4)$.

Смоделируем параметры потока воздуха, адиабатическая постоянная которого æ $= 1.4$, при начальном давлении $P_0 = 10^5$ Па, начальной плотности $\rho_0 = 1$ кг/м^3 и различных начальных скоростях u_0. Сопло имеет коническую форму, радиус которого $r = 0.3 + kx$. Выберем шаг $h = 0.00001$ м. В случае сужающегося канала, когда число Маха стремится к единице, будем считать до тех пор, пока не нарушится монотонность вычислений, т.е. если, например, число Маха непрерывно росло, а в точке x_{n+1} стало меньше, чем в точке x_n, значит где-то на отрезке $[x_n; x_{n+1}]$ достигается число Маха равное единице и вычисления нужно закончить. В случае же расширяющегося канала будем считать до заранее заданной точки $x^* = 3$ м.

Случай 1. $u_0 = 250$, $k = -0.3$, т.е. $M < 1$, $F' < 0$

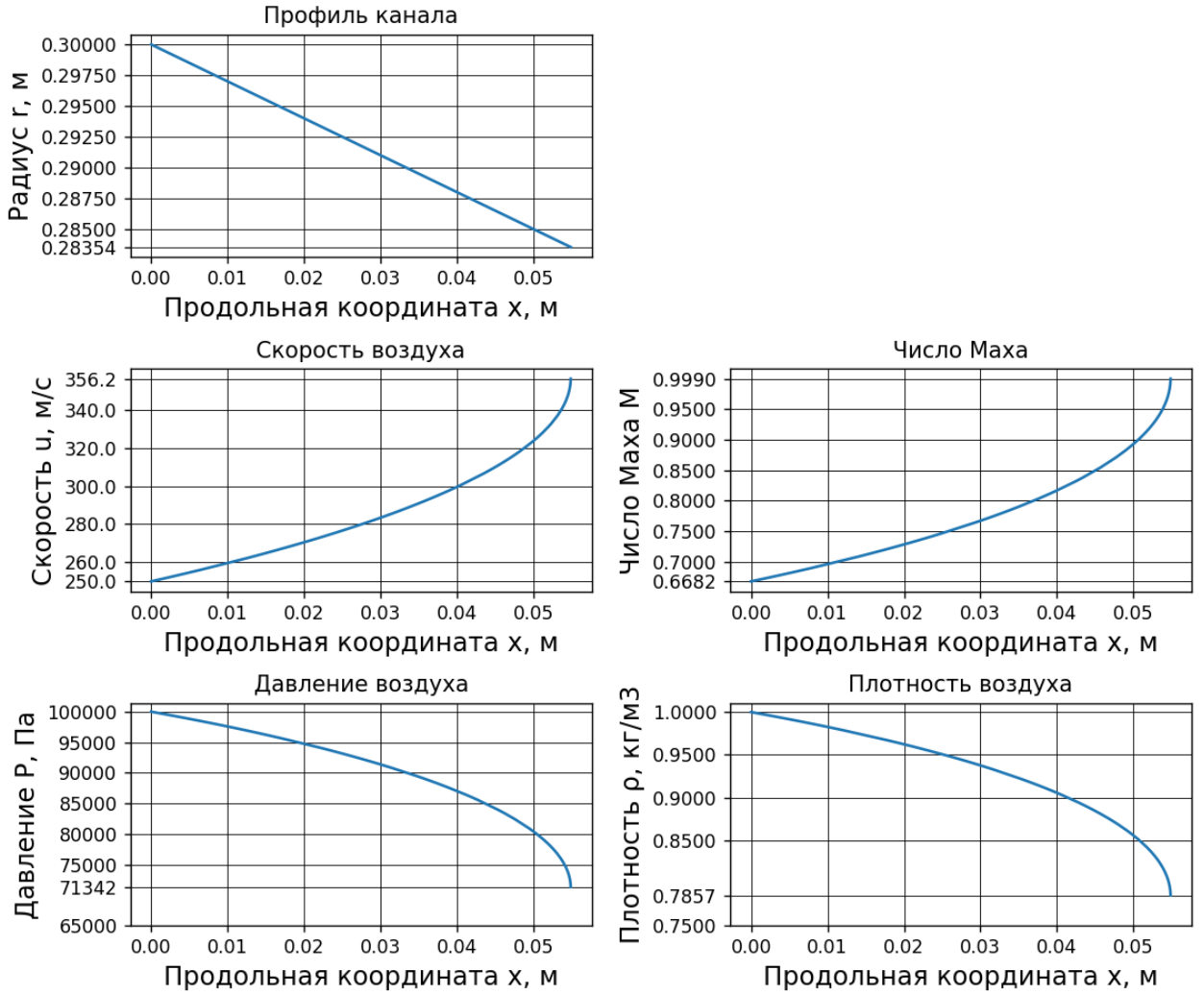

Рисунок 3. Графики параметров газа (M < 1, F’ < 0)

Случай 2. $u_0 = 450$, $k = -0.3$, т.е. $M > 1$, $F' < 0$

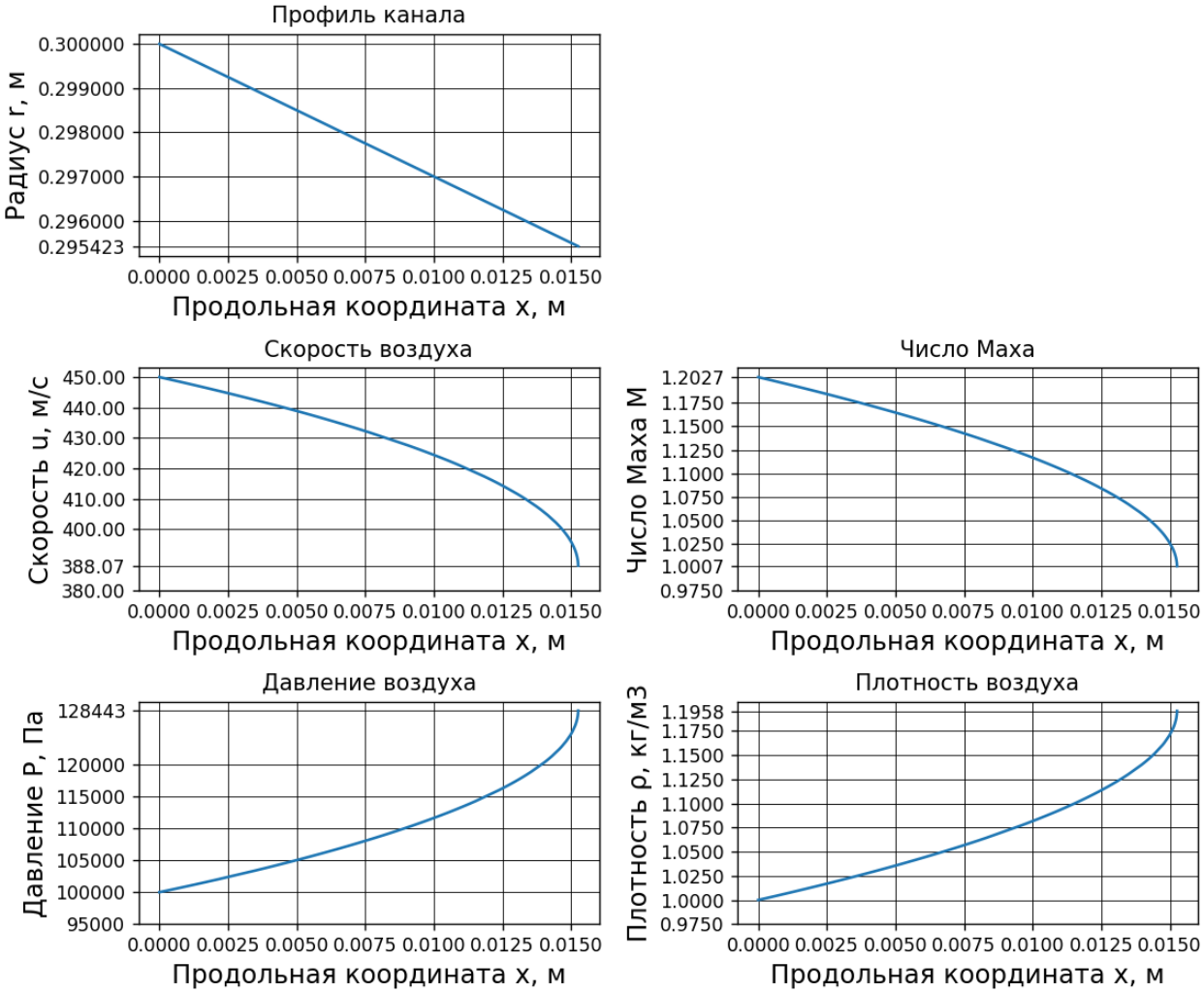

Рисунок 4. Графики параметров газа (M > 1, F’ < 0)

Случай 3. $u_0 = 250$, $k = 0.3$, $x_1 = 3$, т.е. $M < 1$, $F' > 0$

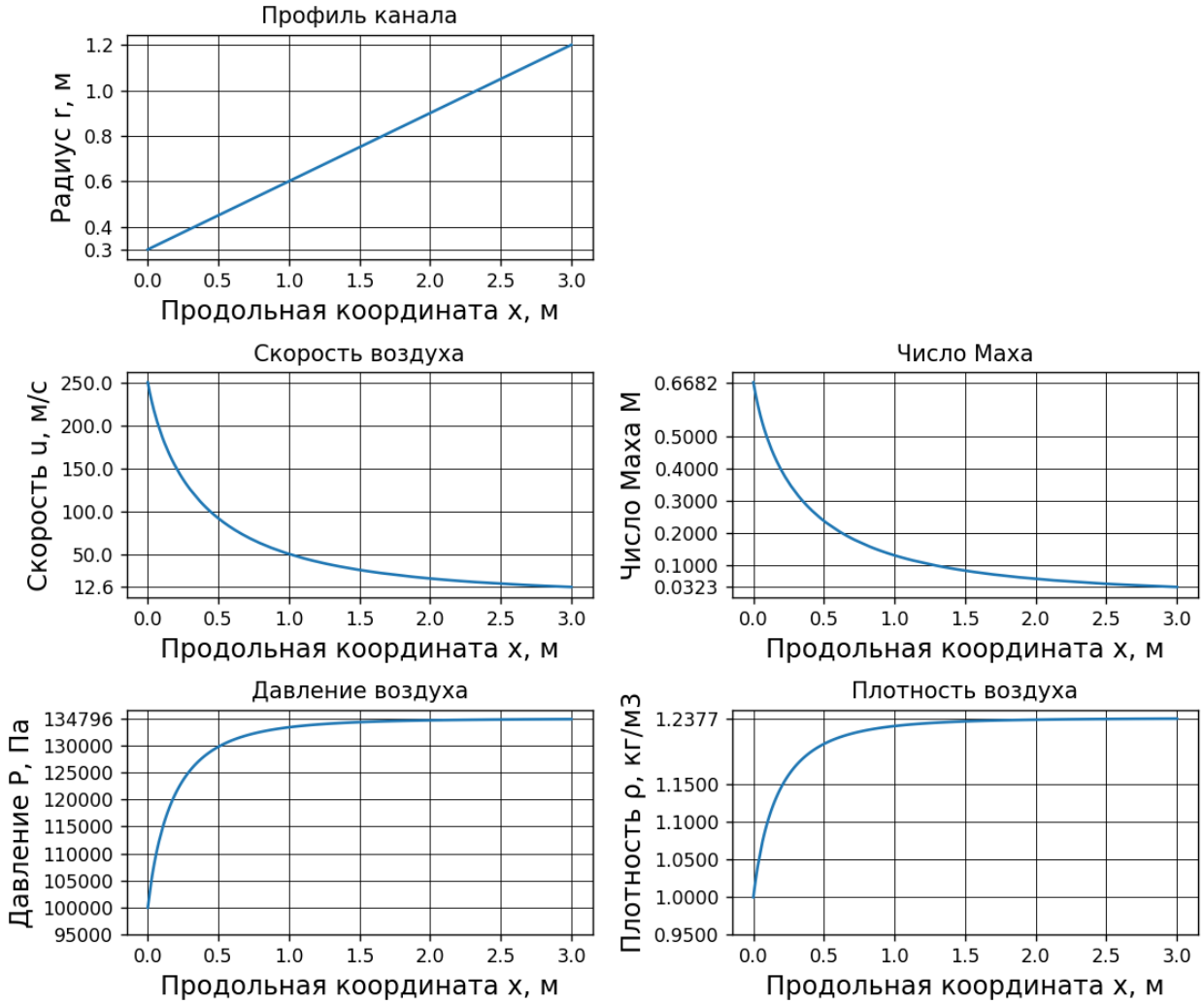

Рисунок 5. Графики параметров газа (M <1, F’ > 0)

Случай 4. $u_0 = 450, k = 0.3,\ x_1 = 3$, т.е. $M > 1, F' > 0$

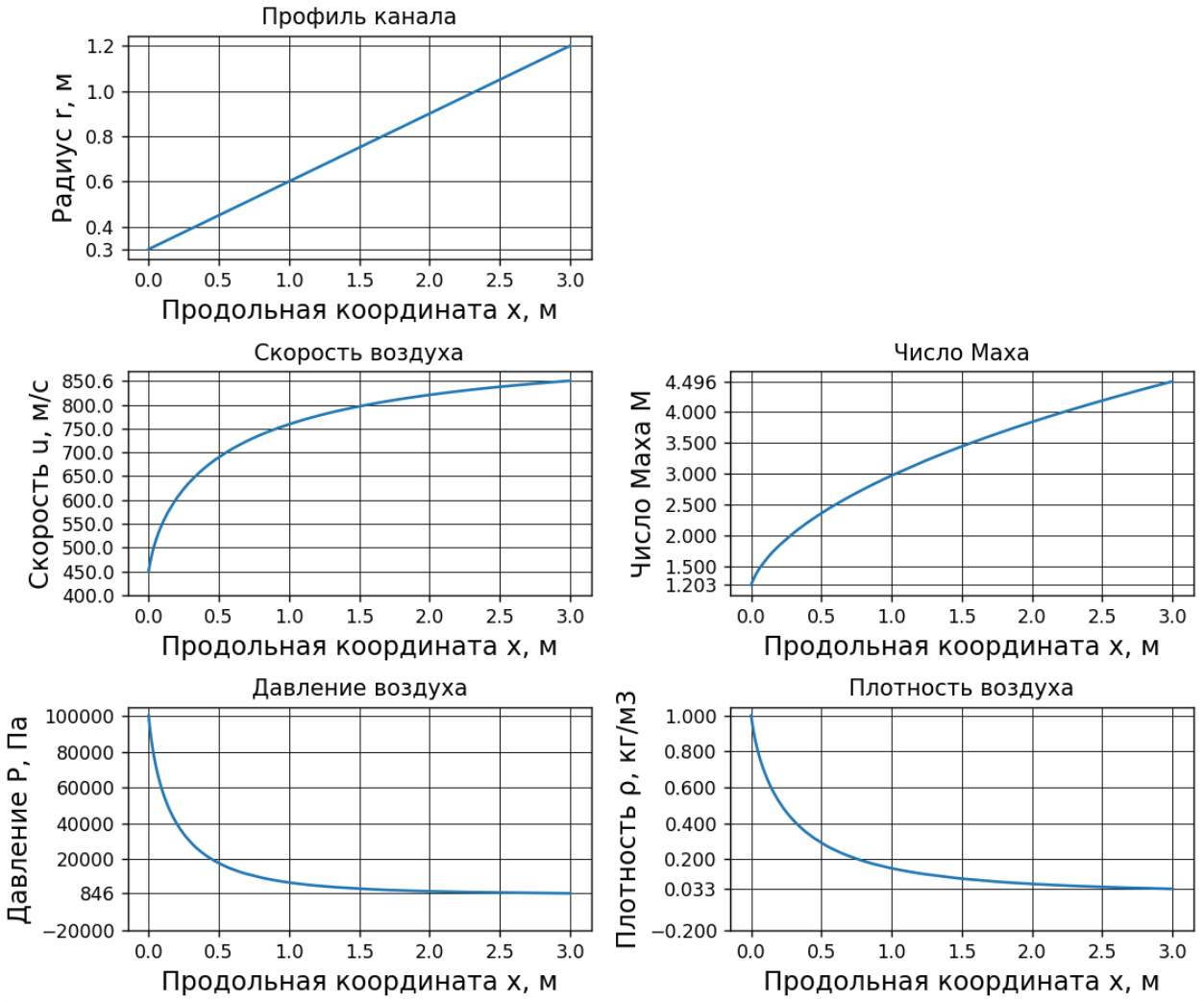

Рисунок 6. Графики параметров газа (M >1, F’ > 0)

Листинг 45. Расчёт числа Маха

```
double calc_sound_speed(double pressure, double density, double
                        adiabatic_index)
{
    return std::sqrt(adiabatic_index * pressure / density);
}

double calc_mach_number(double speed, double pressure, double density,
                        double adiabatic_index)
{
    return speed/calc_sound_speed(pressure, density, adiabatic_index);
}
```

Листинг 46. Нахождение параметров газа с помощью метода Рунге-Кутты 4-ого порядка

```
std::vector<gas_params> calc_gas_params_with_runge_kutta4(
    double x0, double x1, double h,
    double speed0, double pressure0, double density0,
    double adiabatic_index,
    std::function<double (double)> calc_square,
    std::function<double (double)> calc_square_deriv)
{
    auto calc_step = [calc_square, calc_square_deriv,
adiabatic_index](
        double x, double speed, double pressure, double density)
    {
        double m = calc_mach_number(speed, pressure, density,
adiabatic_index);

        double f = calc_square(x);
        double fd = calc_square_deriv(x);

        double k_speed = (speed * fd) / ((m*m - 1) * f);
        double k_pressure = -(density * speed * speed * fd) / ((m*m -
1) * f);
        double k_density = -(m * m * density * fd) / ((m*m - 1) * f);

        return std::make_tuple(k_speed, k_pressure, k_density);
    };

    size_t n = (x1 - x0) / h + 1;
    if (calc_square_deriv(x0) < 0)
    {
        n = std::numeric_limits<size_t>::max();
    }

    std::vector<gas_params> params(0);

    double mach0 = calc_mach_number(speed0, pressure0, density0,
adiabatic_index);
    params.push_back( { x0, speed0, pressure0, density0, mach0 } );

    for (size_t i = 1; i < n; ++i)
    {
        double const &x = params[i-1].x;
        double const &speed = params[i-1].speed;
        double const &pressure = params[i-1].pressure;
        double const &density = params[i-1].density;
        double const &mach = params[i-1].mach_number;

        auto [s_k1, p_k1, d_k1] = calc_step(x, speed, pressure,
density);
        auto [s_k2, p_k2, d_k2] = calc_step(x + h/2, speed + h*s_k1/2,
pressure + h*p_k1/2, density + h*d_k1/2);
```

```
        auto [s_k3, p_k3, d_k3] = calc_step(x + h/2, speed + h*s_k2/2,
pressure + h*p_k2/2, density + h*d_k2/2);
        auto [s_k4, p_k4, d_k4] = calc_step(x + h, speed + h*s_k3,
pressure + h*p_k3, density + h*d_k3);
        double next_x        = x0 + i*h;
        double next_speed    = params[i-1].speed    + h/6 * (s_k1 +
2*s_k2 + 2*s_k3 + s_k4);
        double next_pressure = params[i-1].pressure + h/6 * (p_k1 +
2*p_k2 + 2*p_k3 + p_k4);
        double next_density  = params[i-1].density  + h/6 * (d_k1 +
2*d_k2 + 2*d_k3 + d_k4);
        double next_mach     = calc_mach_number(next_speed,
next_pressure, next_density, adiabatic_index);

        if ((calc_square_deriv(x0) < 0) && (
                (mach0 < 1 && next_mach > 1 -
std::numeric_limits<double>::epsilon()) ||
                (mach0 > 1 && next_mach < 1 +
std::numeric_limits<double>::epsilon()) ||
                (mach0 < 1 && next_mach < mach) ||
                (mach0 > 1 && next_mach > mach)))
        {
            return params;
        }

        params.push_back( { next_x, next_speed, next_pressure,
next_density, next_mach } );
    }

    return params;
}
```

Листинг 47. Чтение входных данных из файла и вызов основной функции

```
int main()
{
    std::freopen("input.txt", "r", stdin);
    // x0 x1 r0 a h
    // speed0
    // pressure0
    // density0
    // adiabatic index

    double x0, x1, r0, a, h;
    double speed0, pressure0, density0;
    double adiabatic_index;

    std::cin >> x0 >> x1 >> r0 >> a >> h >> speed0 >> pressure0 >>
density0 >> adiabatic_index;

    double mach0 = calc_mach_number(speed0, pressure0, density0,
adiabatic_index);
```

```
    std::cout << a << std::endl;

    std::function<double (double)> calc_radius = [r0, a, x0](double x)
    {
        return r0 + a * (x - x0);
    };

    std::function<double (double)> calc_square = [calc_radius](double
x) -> double
    {
        return std::numbers::pi * std::pow(calc_radius(x), 2);
    };

    std::function<double (double)> calc_square_deriv = [a,
calc_radius](double x) -> double
    {
        return 2 * std::numbers::pi * a * calc_radius(x);
    };

    auto params = calc_gas_params_with_runge_kutta4(x0, x1, h, speed0,
                              pressure0, density0, adiabatic_index,
                              calc_square, calc_square_deriv);
```

Листинг 48. Построение графиков

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

def update_scale(scale, mn, mx):
    scale = np.delete(scale, np.argmax(scale))
    scale = np.delete(scale, np.argmin(np.abs(scale - mn)))
    scale = np.delete(scale, np.argmin(np.abs(scale - mx)))
    scale = np.append(scale, mn)
    scale = np.append(scale, mx)
    return scale

x = 1
speed = []
pressure = []
density = []
mach_number = []
r0 = 0
a = 0
x0 = 0
x1 = 0
radius = []

with open('buf', 'r', encoding='utf-8') as buf_file:
    tmp = buf_file.readline().split(' ')
    n = int(buf_file.readline())
    x = buf_file.readline().split(' ')
    speed = buf_file.readline().split(' ')
```

```
    pressure = buf_file.readline().split(' ')
    density = buf_file.readline().split(' ')
    mach_number = buf_file.readline().split(' ')

    x.pop()
    speed.pop()
    pressure.pop()
    density.pop()
    mach_number.pop()

    r0 = float(tmp[0])
    a = float(tmp[1])
    x = list(map(float, x))
    speed = list(map(float, speed))
    pressure = list(map(float, pressure))
    density = list(map(float, density))
    mach_number = list(map(float, mach_number))
    x0 = x[0]
    x1 = x[n-1]
    radius = [r0 + a*(t - x0) for t in x]

fig, ax = plt.subplots(3, 2, figsize=(18, 12))

ax[0][0].plot(x, radius, label='Профиль канала')
ax[0][0].set_title('Профиль канала')
ax[0][0].set_xlabel('Продольная координата x, м', fontsize=14)
ax[0][0].set_ylabel('Радиус r, м', fontsize=14)
ax[0][0].set_yticks(update_scale(ax[0][0].get_yticks(), min(radius),
max(radius)))
ax[0][0].grid(color='black', linewidth=0.5)

ax[0][1].set_visible(False)

ax[1][0].plot(x, speed, label='Скорость')
ax[1][0].set_title('Скорость воздуха')
ax[1][0].set_xlabel('Продольная координата x, м', fontsize=14)
ax[1][0].set_ylabel('Скорость u, м/с', fontsize=14)
ax[1][0].set_yticks(update_scale(ax[1][0].get_yticks(), min(speed),
max(speed)))
ax[1][0].grid(color='black', linewidth=0.5)

ax[1][1].plot(x, mach_number, label='Число Маха')
ax[1][1].set_title('Число Маха')
ax[1][1].set_xlabel('Продольная координата x, м', fontsize=14)
ax[1][1].set_ylabel('Число Маха M', fontsize=14)
ax[1][1].set_yticks(update_scale(ax[1][1].get_yticks(),
min(mach_number), max(mach_number)))
ax[1][1].grid(color='black', linewidth=0.5)

ax[2][0].plot(x, pressure, label='Давление')
ax[2][0].set_title('Давление воздуха')
```

```
ax[2][0].set_xlabel('Продольная координата x, м', fontsize=14)
ax[2][0].set_ylabel('Давление P, Па', fontsize=14)
ax[2][0].set_yticks(update_scale(ax[2][0].get_yticks(), min(pressure),
max(pressure)))
ax[2][0].grid(color='black', linewidth=0.5)

ax[2][1].plot(x, density, label='Плотность')
ax[2][1].set_title('Плотность воздуха')
ax[2][1].set_xlabel('Продольная координата x, м', fontsize=14)
ax[2][1].set_ylabel('Плотность ρ, кг/м3', fontsize=14)
ax[2][1].set_yticks(update_scale(ax[2][1].get_yticks(), min(density),
max(density)))
ax[2][1].grid(color='black', linewidth=0.5)

plt.subplots_adjust(hspace=0.5, wspace=0.3, top=0.95, bottom=0.07,
left=0.08, right=0.6)
plt.show()
```

Программа считывает следующие входные данные из файла input.txt:

1) Геометрические параметры сопла: поперечная координата, соответствующая началу сопла x0, поперечная координата, соответствующая концу сопла x1 (используется, когда скорость может бесконечно убывать или возростать), радиус сопла в его начале r0, коэффициент изменения радиуса сопла a;
2) Шаг расчёта дифференциальных уравнений h;
3) Параметры потока в начале сопла: скорость speed0, давление pressure0, плотность density0
4) Показатель адиабаты adiabatic_index.

Затем с помощью метода Рунге-Кутты четвёртого порядка на интервале $[x_0; x^*]$ численно решается приведённая выше система уравнений и по полученным данным строятся графики. Здесь в случае сужающегося канала, x^* равно точке, в которой число Маха газа достигает единицы, и $x^* = x_1$ в случае, когда сопло расширяется.

Вывод

В результате выполнения курсового проекта была достигнута поставленная цель — разработка и применение численных методов для решения разнообразных задач математического моделирования. Проведенное исследование позволило обеспечить освоение практических навыков и углубить понимание особенностей численных подходов.

В ходе работы были успешно решены следующие задачи:

1. Изучение основных численных методов: были исследованы численные методы для решения линейных и нелинейных уравнений, такие как метод Гаусса, метод прогонки и итерационные методы, что позволило обеспечить базу для последующих разработок и применений.
2. Алгоритмы для численных методов: Разработаны алгоритмы для реализации методов Гаусса, прогонки, метода Ньютона, метода секущих и других, которые обеспечили возможность решения систем уравнений.
3. Применение для систем уравнений: Разработанные алгоритмы были успешно применены для решения систем линейных и нелинейных уравнений, подтверждая их эффективность и пригодность для практических задач.
4. Построение интерполяционных функций: Построены интерполяционные многочлены и сплайны, а также аппроксимированы функции методом наименьших квадратов, что расширило применение численных методов в задачах приближенного анализа.
5. Численные методы для производных и интегралов: было выполнено определение значений производных и интегралов с использованием численных методов, что продемонстрировало возможности вычислительной аппроксимации.
6. Решение задач Коши: Реализованы методы Эйлера, Рунге-Кутты и Адамса для обыкновенных дифференциальных уравнений второго порядка, что подтвердило гибкость этих методов в численных вычислениях.
7. Анализ точности: Проведен анализ точности численных методов, исследовано влияние параметров вычислений на их результаты, что позволило оценить практическую применимость и стабильность методов.
8. Разработан вычислительный алгоритм и программа расчёта параметров стационарного потока идеального газа в канале переменного сечения.

Реализация всех задач показала эффективность применения языка программирования C++ для создания высокопроизводительных и гибких программных решений, предоставляя удобные средства для анализа и визуализации результатов.

Таким образом, выполненные исследования и разработки значимо расширили понимание и практическое применение численных методов в решении задач различного класса, подчеркивая их роль в современном математическом моделировании и вычислительной математике.

Использованные источники

1) Термодинамические свойства индивидуальных веществ. Справочное издание: В 4-х т./ Л.В. Гурвич, И.В. Вейц, В.А. Медведев и др. – 3-е изд., перераб. и расширен. – Т. 1. Кн. 2. – М.: Наука, 1978. – 328 с.

2) Численные методы: учебник и практикум для среднего профессионального образования / У. Г. Пирумов и др. – 5-е изд., перераб. и доп. – М.: Издательство Юрайт, 2015. – 421 с.

3) Элементарные модели и вычислительные алгоритмы физической газовой динамики. Термодинамика и химическая кинетика / В.Ю. Гидаспов, Н.С. Северина. — М.: Факториал, 2014. — 84 с.