Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *Робототехники и комплексной автоматизации*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

по дисциплине: «Модели и методы анализа проектных решений»

Студент	Харитонов Евгений Юрьевич
Группа	РК6-72Б
Тип задания	лабораторная работа
Тема лабораторной работы	метод конечных элементов

Студент ________________ **Харитонов Е.Ю.**
подпись, дата *фамилия, и.о.*

Преподаватель ________________ **Трудоношин В.А.**
подпись, дата *фамилия, и.о.*

Оценка ________________________________

Москва, 2020 г.

Оглавление

Задание

Вариант 39.

Методом конечных элементов решить уравнение:

$$2\frac{d^2u}{dx^2} - 7u + 3 = 0$$

при следующих граничных условиях:

$$\frac{du}{dx}(x = 2) = -5$$

$$u(7) = 10\,.$$

Количество конечных элементов для первого расчёта равно 20, для второго расчёта - 40. Сравнить результат с аналитическим решением, оценить максимальную погрешность.

Выполнение работы

Получение аналитического решения

Для получения аналитического решения был использован сервис https://www.wolframalpha.com/. Результат представлен на рис. 1.

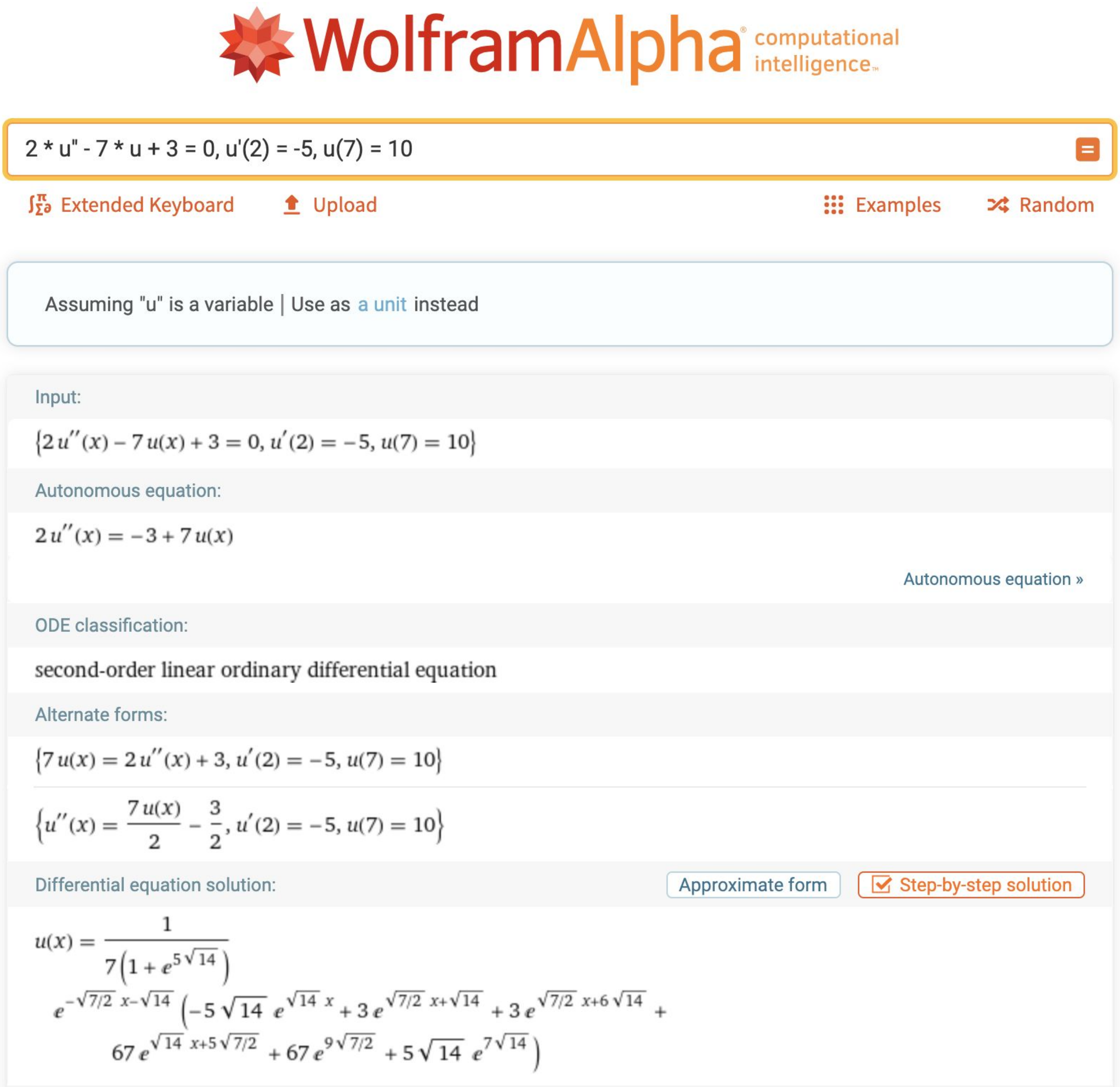

Рис. 1. Аналитическое решение неоднородного дифференциального уравнения с помощью сервиса wolframalpha.

Решение методом конечных элементов

1. Решение с помощью линейной функции формы

В случае линейной функции формы аппроксимация решения производится базисными функциями вида

$$y = a_0 + a_1 x$$

Возьмем в качестве длины конечного элемента L и получим координаты его граничных узлов

$$x = 0 \quad Y_i = a_0,$$

$$x = L \quad Y_j = a_0 + a_1 L,$$

а отсюда

$$a_0 = Y_i,$$

$$a_1 = \frac{Y_j - Y_i}{L}.$$

Подставив найденные коэффициенты в изначальную функцию получаем

$$y = \left(1 - \frac{x}{L}\right) Y_i + \frac{x}{L} Y_j = \begin{bmatrix} 1-\frac{x}{L} & \frac{x}{L} \end{bmatrix} \cdot \begin{bmatrix} Y_i \\ Y_j \end{bmatrix} = \mathbf{N}_e \mathbf{Y},$$

где $\mathbf{N}_e$ вектор функций формы. Далее необходимо применить метод Галеркина для получения вектора нагрузок и матрицы жесткости.

$$\int_0^L N_e^T \left(2\frac{\mathrm{d}^2U}{\mathrm{d}x^2} - 7U + 3\right) = 0,$$

$$\begin{bmatrix} -\left.\frac{\mathrm{d}U}{\mathrm{d}x}\right|_i \\ \left.\frac{\mathrm{d}U}{\mathrm{d}x}\right|_j \end{bmatrix} - \begin{bmatrix} \frac{1}{L} & -\frac{1}{L} \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix} \begin{bmatrix} U_i \\ U_j \end{bmatrix} - \frac{7}{2} \begin{bmatrix} \frac{L}{3} & \frac{L}{6} \\ \frac{L}{6} & \frac{L}{3} \end{bmatrix} \begin{bmatrix} U_i \\ U_j \end{bmatrix} + \frac{3}{2} \begin{bmatrix} \frac{L}{2} \\ \frac{L}{2} \end{bmatrix} = 0.$$

После выполнения ряда преобразований была получена СЛАУ $Au = b$, где A - матрица жесткости, b - вектор нагрузок, а вектор неизвестных u - перемещения в узлах:

$$A = \begin{bmatrix} \frac{1}{L} + \frac{7L}{6} & -\frac{1}{L} + \frac{7L}{12} \\ -\frac{1}{L} + \frac{7L}{12} & \frac{1}{L} + \frac{7L}{6} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$b = \begin{bmatrix} -\frac{\mathrm{d}U}{\mathrm{d}x}\Big|_i + \frac{3L}{4} \\ \frac{\mathrm{d}U}{\mathrm{d}x}\Big|_j + \frac{3L}{4} \end{bmatrix}.$$

Для дальнейшего получения глобальной матрицы жесткости и глобального вектора нагрузок выполним простую замену и проведем ансамблирование:

$$\begin{bmatrix} a & b & 0 & 0 & 0 \\ c & a+d & b & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & c & a+d & b \\ 0 & 0 & 0 & c & d \end{bmatrix} \begin{bmatrix} U_0 \\ U_1 \\ \vdots \\ U_{n-1} \\ U_n \end{bmatrix} = \begin{bmatrix} \frac{3L}{4} - \frac{\mathrm{d}U}{\mathrm{d}x}\Big|_0 \\ \frac{3L}{2} \\ \vdots \\ \frac{3L}{2} \\ \frac{3L}{4} - \frac{\mathrm{d}U}{\mathrm{d}x}\Big|_n \end{bmatrix}.$$

Чтобы решить данную СЛАУ, необходимо учесть граничные условия:

$$\frac{du}{dx}(x = 2) = -5,$$

$$u(7) = 10,$$

$$\begin{bmatrix} a & b & 0 & 0 & 0 \\ c & a+d & b & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & c & a+d & 0 \\ 0 & 0 & 0 & c & 1 \end{bmatrix} \begin{bmatrix} U_0 \\ U_1 \\ \vdots \\ U_{n-1} \\ \frac{\mathrm{d}U}{\mathrm{d}x}\Big|_n \end{bmatrix} = \begin{bmatrix} \frac{3L}{4} + 5 \\ \frac{3L}{2} \\ \vdots \\ \frac{3L}{2} - 10b \\ \frac{3L}{4} - 10d \end{bmatrix}.$$

Поскольку глобальная матрица жесткости трехдиагональная, то при программной реализации решения СЛАУ разумно использовать метод прогонки. Также заметим, что строки 1...n-2 одинаковые, поэтому при программной реализации имеет смысл не хранить в памяти всю глобальную матрицу, а хранить только матрицу размером 4 на 4 (в ней первая строка будет равна строкам 1...n-2 в полной глобальной матрице жесткости). Алгоритм прогонки был реализован с учетом данного способа хранения матрицы.

2. Решение с помощью кубической функции формы

Выполнив аналогичные со случаем линейной функции формы действия и преобразования над полиномом 3-й степени, получим вектор функции формы и применим метод Галеркина для получения матрицы жесткости и вектора нагрузок

$$\begin{bmatrix} \frac{37}{10L}+\frac{4L}{15} & -\frac{189}{40L}+\frac{33L}{160} & \frac{27}{20L}-\frac{3L}{40} & -\frac{13}{40L}+\frac{19L}{480} \\ -\frac{189}{40L}+\frac{33L}{160} & \frac{54}{5L}+\frac{27}{20L} & -\frac{279}{40L}-\frac{27L}{160} & \frac{27}{20L}-\frac{3L}{40} \\ \frac{27}{20L}-\frac{3L}{40} & -\frac{279}{40L}-\frac{27L}{160} & \frac{54}{5L}+\frac{27}{20L} & -\frac{189}{40L}+\frac{33L}{160} \\ -\frac{13}{40L}+\frac{19L}{480} & \frac{27}{20L}-\frac{3L}{40} & -\frac{189}{40L}+\frac{33L}{160} & \frac{37}{10L}+\frac{4L}{15} \end{bmatrix} \begin{bmatrix} U_i \\ U_j \\ U_k \\ U_l \end{bmatrix} = \begin{bmatrix} \frac{3L}{16}-\left.\frac{\mathrm{d}U}{\mathrm{d}x}\right|_i \\ \frac{9L}{16} \\ \frac{9L}{16} \\ \frac{3L}{16}+\left.\frac{\mathrm{d}U}{\mathrm{d}x}\right|_l \end{bmatrix}$$

Произведя замену переменных, запишем матрицы в общем виде и обнулим элементы, стоящие выше и ниже главной диагонали (исключая крайние столбцы):

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} U_i \\ U_j \\ U_k \\ U_l \end{bmatrix} = \begin{bmatrix} -\left.\frac{\mathrm{d}U}{\mathrm{d}x}\right|_i + b_0 \\ b_1 \\ b_2 \\ b_3 + \left.\frac{\mathrm{d}U}{\mathrm{d}x}\right|_l \end{bmatrix},$$

$$\begin{bmatrix} \bar{a_{00}} & 0 & 0 & \bar{a_{03}} \\ \bar{a_{10}} & \bar{a_{11}} & 0 & \bar{a_{13}} \\ \bar{a_{20}} & 0 & \bar{a_{22}} & \bar{a_{23}} \\ \bar{a_{30}} & 0 & 0 & \bar{a_{33}} \end{bmatrix} \begin{bmatrix} U_i \\ U_j \\ U_k \\ U_l \end{bmatrix} = \begin{bmatrix} \bar{b_0} \\ \bar{b_1} \\ \bar{b_2} \\ \bar{b_3} \end{bmatrix}.$$

Заметим, что в таком случае нет зависимости внешних узлов от внутренних, а значит внутренние можно исключить. В результате получим систему

$$
\begin{bmatrix} \bar{a_{00}} & \bar{a_{03}} \\ \bar{a_{30}} & \bar{a_{33}} \end{bmatrix} \begin{bmatrix} U_i \\ U_l \end{bmatrix} = \begin{bmatrix} \bar{b_0} \\ \bar{b_3} \end{bmatrix}.
$$

Дальнейший процесс решения аналогичен случаю линейной функции формы.

Анализ результатов

Описанные выше алгоритмы были реализованы на языке C++. Тестовые запуски произведены для линейной и кубической функций формы для случаев 20 и 40 элементов. В качестве точного решения для оценки точности метода конечных элементов было использовано решение, полученное аналитически. На рисунках 2-5 представлены графики, визуализирующие результаты.

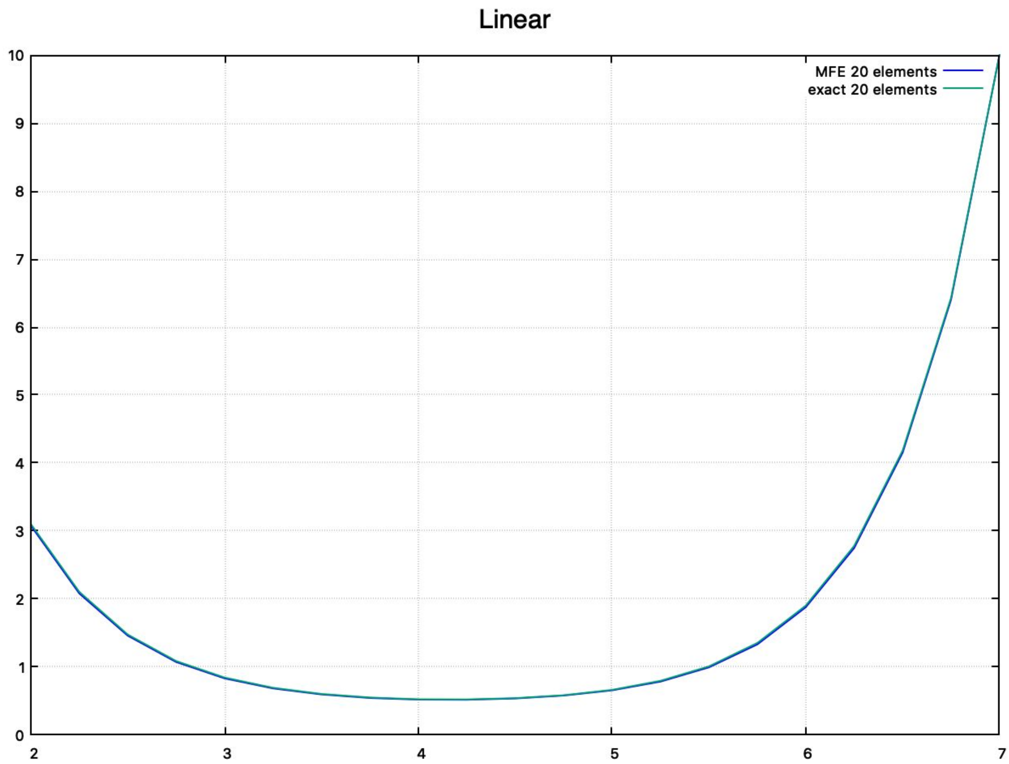

Рис. 2. Решение МКЭ линейной функции формы с 20 конечными элементами (синий график) и точное решение (зеленый график)

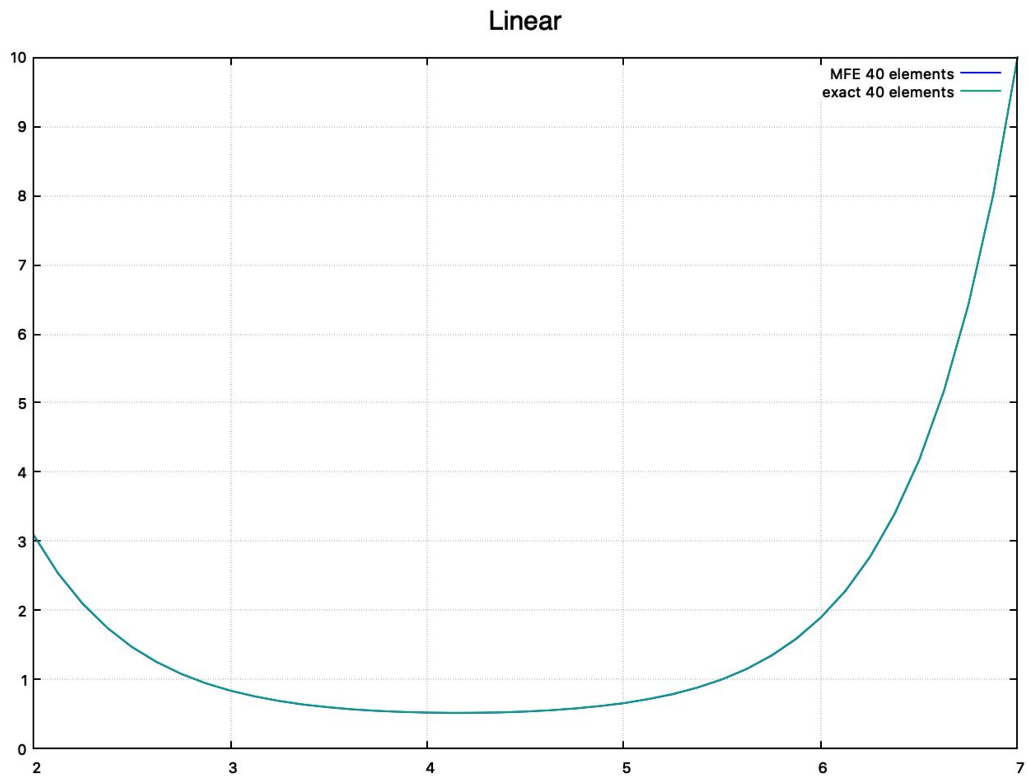

Рис. 3. Решение МКЭ линейной функции формы с 40 конечными элементами (синий график) и точное решение (зеленый график)

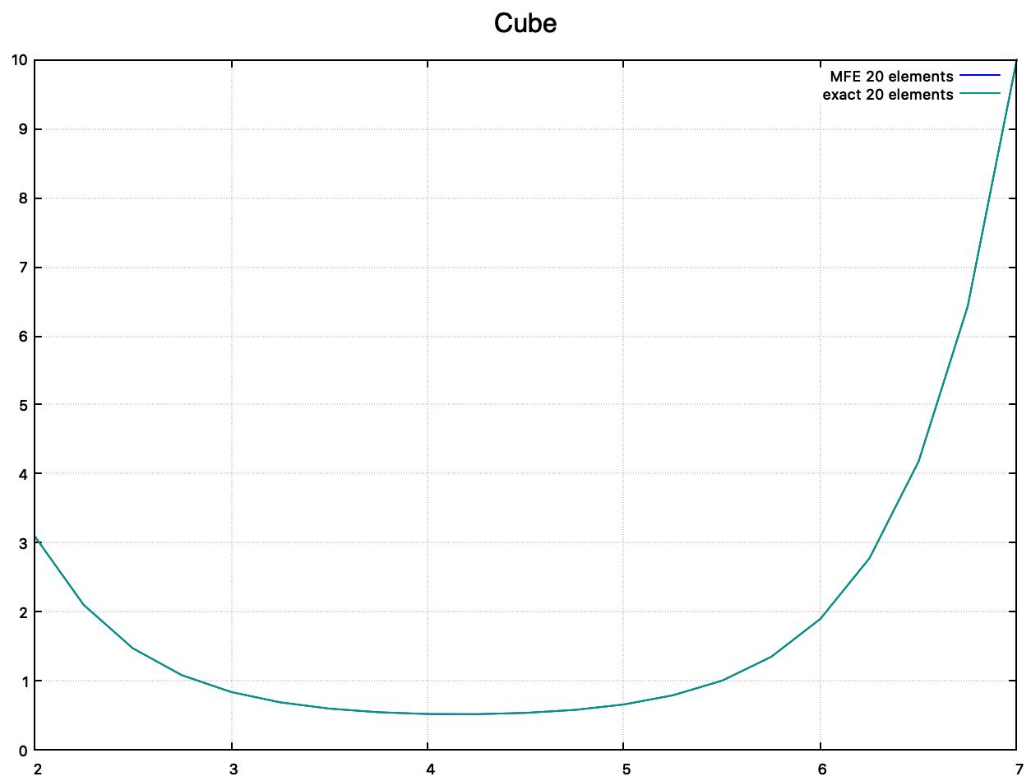

Рис. 4. Решение МКЭ кубической функции формы с 20 конечными элементами (синий график) и точное решение (зеленый график)

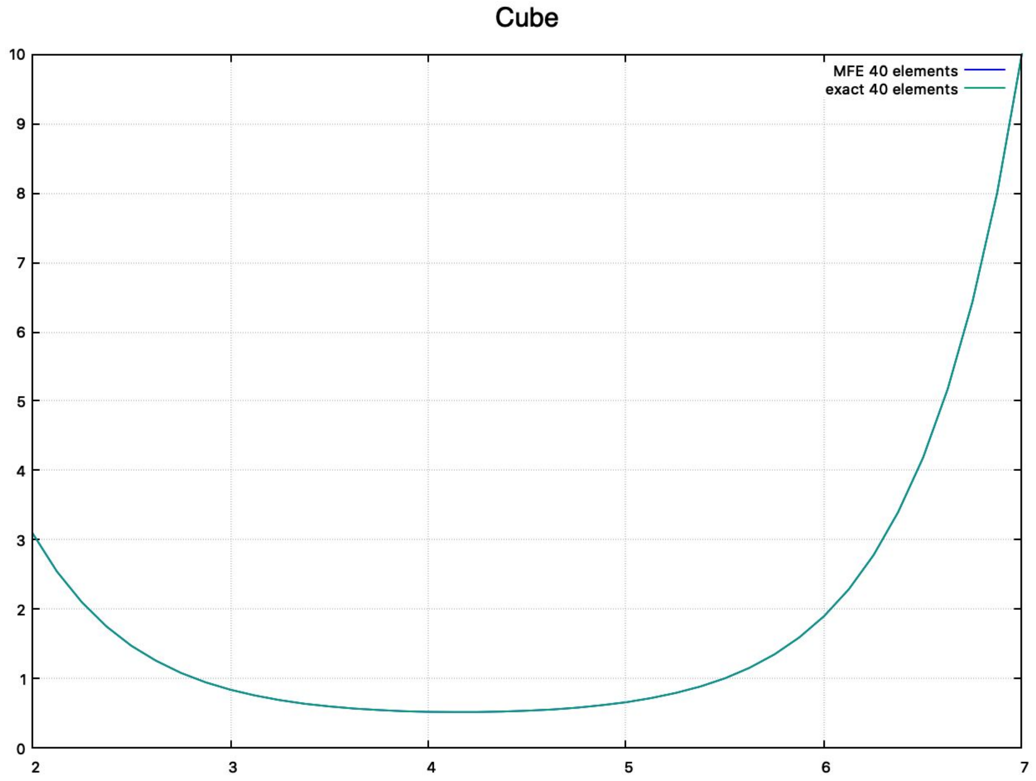

Рис. 5. Решение МКЭ кубической функции формы с 40 конечными элементами (синий график) и точное решение (зеленый график)

Заметим, что визуально между точным решением и решением МКЭ нет никакой разницы, поэтому была найдена максимальная абсолютная погрешность для каждого случая. Результаты представлены в таблице 1.

Таблица 1. Погрешность в зависимости от функции формы и количества конечных элементов

Функция формы	Количество элементов	Максимальная абсолютная погрешность
Линейная	20	3.27294e-02
	40	8.05715e-03
Кубическая	20	1.8406e-07
	40	2.85978e-09

Оценим, сколько нужно использовать линейных конечных элементов для достижения погрешности, аналогичной погрешности при использовании кубической функции формы. При использовании 8500 линейных элементов погрешность в узлах составила 1.78265e-07, что близко к погрешности при использовании 20 кубических элементов. Аналог 40 кубических элементов - 80000 элементов (погрешность 2.87777e-09). Однако, было замечено, что начиная с 300000 линейных элементов погрешность при увеличении количества элементов начинает расти. Возможно, это связано с накоплением погрешности методом прогонки.

Заключение

В ходе решения дифференциального уравнения методом конечных элементов было установлено, что использование кубической функции формы конечного элемента позволяет достичь значительно более точных результатов по сравнению с линейными элементами.

Приложение

Код программы

```
#include <vector>
#include <cmath>
#include <iostream>

// Настройки
bool is_cube = false;
size_t elements_count = 80000;

// задание ГУ
double du_0 = -5;
double u_n = 10;
double start_x = 2;
double end_x = 7;

// Вспомогательное
```

```
double element_len = (end_x - start_x) / elements_count;

double A_local_linear[2][2] = {{1 / element_len + 7 * element_len / 6,      -1 / element_len + 7 * element_len / 
12},
                              {-1 / element_len + 7 * element_len / 12,    1 / element_len + 7 * element_len / 6}};
double b_local_linear[2] =     {3 * element_len / 4,                        3 * element_len / 4};

double A_local_cube[4][4] =
        {{37 / (10 * element_len) + 4 * element_len / 15, -189 / (40 * element_len) + 33 * element_len / 160, 27 / 
(20 * element_len) - 3 * element_len / 40, -13 / (40 * element_len) + 19 * element_len / 480},
         {-189 / (40 * element_len) + 33 * element_len / 160, 54 / (5 * element_len) + 27 * element_len / 20, -297 / 
(40 * element_len) - 27 * element_len / 160, 27 / (20 * element_len) - 3 * element_len / 40},
         {27 / (20 * element_len) - 3 * element_len / 40, -297 / (40 * element_len) + -27 * element_len / 160, 54 / 
(5 * element_len) + 27 * element_len / 20, -189 / (40 * element_len) + 33 * element_len / 160},
         {-13 / (40 * element_len) + 19 * element_len / 480, 27 / (20 * element_len) - 3 * element_len / 40, -189 / 
(40 * element_len) + 33 * element_len / 160, 37 / (10 * element_len) + 4 * element_len / 15}};
double b_local_cube[4] = {3 * element_len / 16, 9 * element_len / 16, 9 * element_len / 16, 3 * element_len / 
16};

// Точное решение
std::vector<double> solution() {
    std::vector<double> result(elements_count + 1);
    for (size_t i = 0; i < elements_count + 1; i++) {
        double x = start_x + i * element_len;
        result[i] = 3.0/7 + (67*exp(7*sqrt(14)/2) + 5*sqrt(14)*exp(6*sqrt(14)))*exp(-x*sqrt(14)/2)/(7*(1 + 
exp(5*sqrt(14)))) + (-5*sqrt(14) + 67*exp(5*sqrt(14)/2))*exp(-sqrt(14))*exp(x*sqrt(14)/2)/(7*(1 + 
exp(5*sqrt(14))));
    }
    return result;
}

template <typename T>
class Matrix {
public:
    Matrix(size_t rows, size_t cols, T value);
    Matrix(const Matrix<T> &matrix);
    std::vector<T> tridiagonal_matrix_algorithm(std::vector<T> &b);
    std::vector<T> fast_tridiagonal_matrix_algorithm(std::vector<T> &b);
    T& get_elem(size_t r, size_t c);
private:
    std::vector<std::vector<T>> data;
    size_t rows;
    size_t cols;
};

template <typename T>
Matrix<T>::Matrix(size_t rows, size_t cols, T value) : rows(rows), cols(cols) {
    data.resize(rows);
    for (size_t i = 0; i < rows; i++) {
        data[i].resize(cols, value);
    }
}
```

```
template <typename T>
Matrix<T>::Matrix(const Matrix<T> &matrix) : rows(matrix.rows), cols(matrix.cols), data(matrix.data) {
}

template <typename T>
T& Matrix<T>::get_elem(size_t r, size_t c) {
    return data[r][c];
}

//метод прогонки
template <typename T>
std::vector<T> Matrix<T>::tridiagonal_matrix_algorithm(std::vector<T> &b) {

    size_t n = this->rows;
    std::vector<T> alpha(n);
    std::vector<T> beta(n);

    alpha[0] = -this->data[0][1] / this->data[0][0];
    beta[0] = b[0] / this->data[0][0];


    for (size_t i = 1; i < n - 1; i++) {
        double y = this->data[i][i] + this->data[i][i - 1] * alpha[i - 1];
        alpha[i] = -this->data[i][i + 1] / y;
        beta[i] = (b[i] - this->data[i][i - 1] * beta[i - 1]) / y;
    }
    double y = this->data[n - 1][n - 1] + this->data[n - 1][n - 2] * alpha[n - 2];
    beta[n - 1] = (b[n - 1] - this->data[n - 1][n - 2] * beta[n - 2]) / y;

    std::vector<T> x(n);
    x[n - 1] = beta[n - 1];
    for (ssize_t i = n - 2; i >= 0; i--) {
        x[i] = alpha[i] * x[i + 1] + beta[i];
    }

    return x;
}

// ускоренный метод прогонки
template <typename T>
std::vector<T> Matrix<T>::fast_tridiagonal_matrix_algorithm(std::vector<T> &b) {

    size_t n = 4;
    std::vector<T> alpha(elements_count + 1);
    std::vector<T> beta(elements_count + 1);
///
    alpha[0] = -this->data[0][1] / this->data[0][0];
    beta[0] = b[0] / this->data[0][0];

    for (size_t i = 1; i < elements_count - 1; i++) {
        double y = this->data[1][1] + this->data[1][0] * alpha[i - 1];
```

```
        alpha[i] = -this->data[1][2] / y;
        beta[i] = (b[1] - this->data[1][0] * beta[i - 1]) / y;
    }

    double y = this->data[n - 2][n - 2] + this->data[n - 2][n - 3] * alpha[elements_count - 2];
    alpha[elements_count - 1] = -this->data[n - 2][n - 1] / y;
    beta[elements_count - 1] = (b[n - 2] - this->data[n - 2][n - 3] * beta[elements_count + 1 - 3]) / y;

    y = this->data[n - 1][n - 1] + this->data[n - 1][n - 2] * alpha[elements_count - 1];
    beta[elements_count] = (b[n - 1] - this->data[n - 1][n - 2] * beta[elements_count - 1]) / y;
///
    std::vector<T> x(elements_count + 1);
    x[elements_count] = beta[elements_count];
    for (ssize_t i = elements_count - 1; i >= 0; i--) {
        x[i] = alpha[i] * x[i + 1] + beta[i];
    }

    return x;
}


std::vector<double> linear() {

    // Получение матрицы A ансамблированием и учет граничных условий
    Matrix<double> A(elements_count + 1, elements_count + 1, 0);
    for (size_t i = 0; i < elements_count; i++) {
        for (size_t j = 0; j < 2; j++) {
            for (size_t k = 0; k < 2; k++) {
                A.get_elem(i + j, i + k) += A_local_linear[j][k];
            }
        }
    }
    A.get_elem(elements_count, elements_count) = 1;
    A.get_elem(elements_count - 1, elements_count) = 0;

    // Получение вектора b и учет граничных условий
    std::vector<double> b(elements_count + 1);
    for (size_t i = 1; i < elements_count - 1; i++) {
        b[i] = b_local_linear[0] + b_local_linear[1];
    }
    b[0] = b_local_linear[0] - du_0;
    b[elements_count - 1] = b_local_linear[0] + b_local_linear[1] - A_local_linear[0][1] * u_n;
    b[elements_count] = b_local_linear[1] - A_local_linear[1][1] * u_n;

    // Решение СЛАУ Ax=b
    std::vector<double> x = A.tridiagonal_matrix_algorithm(b);
    x[elements_count] = u_n;
    return x;
}

std::vector<double> fast_linear() {
```

```
    // Получение матрицы А ансамблированием и учет граничных условий
    Matrix<double> A(4, 4, 0);
    for (size_t i = 0; i < 3; i++) {
        for (size_t j = 0; j < 2; j++) {
            for (size_t k = 0; k < 2; k++) {
                A.get_elem(i + j, i + k) += A_local_linear[j][k];
            }
        }
    }
    A.get_elem(3, 3) = 1;
    A.get_elem(2, 3) = 0;

    // Получение вектора b и учет граничных условий
    std::vector<double> b(4);
    b[0] = b_local_linear[0] - du_0;
    b[1] = b_local_linear[0] + b_local_linear[1];
    b[2] = b_local_linear[0] + b_local_linear[1] - A_local_linear[0][1] * u_n;
    b[3] = b_local_linear[1] - A_local_linear[1][1] * u_n;

    // Решение СЛАУ Ax=b
    std::vector<double> x = A.fast_tridiagonal_matrix_algorithm(b);
    x[elements_count] = u_n;
    return x;
}

std::vector<double> cube() {

    // Приведение с момощью метода Гаусса локальной матрицы к виду,
    // необходимому для исключения внутренних элементов
    for (size_t i = 1; i < 3; i++) {
        for (size_t j = 0; j < 4; j++) {
            if (i == j | fabs(A_local_cube[j][i]) < 1e-16) {
                continue;
            }
            double piv = A_local_cube[j][i] / A_local_cube[i][i];
            b_local_cube[j] -= piv * b_local_cube[i];
            for (size_t k = 0; k < 4; k++) {
                A_local_cube[j][k] -= piv * A_local_cube[i][k];
            }
        }
    }

    // Получение матрицы А ансамблированием и учет граничных условий
    Matrix<double> matrix(elements_count + 1, elements_count + 1, 0);
    for (size_t i = 0; i < elements_count; i++) {
        matrix.get_elem(i, i) += A_local_cube[0][0];
        matrix.get_elem(i + 1, i) += A_local_cube[3][0];
        matrix.get_elem(i, i + 1) += A_local_cube[0][3];
        matrix.get_elem(i + 1, i + 1) += A_local_cube[3][3];
    }
    matrix.get_elem(elements_count, elements_count) = 1;
    matrix.get_elem(elements_count - 1, elements_count) = 0;
```

```
    // Получение вектора b и учет граничных условий
    std::vector<double> b(elements_count + 1);
    for (size_t i = 1; i < elements_count - 1; i++) {
        b[i] = b_local_cube[0] + b_local_cube[3];
    }
    b[0] = b_local_cube[0] - du_0;
    b[elements_count] = b_local_cube[3] - A_local_cube[3][3] * u_n;
    b[elements_count - 1] = b_local_cube[0] + b_local_cube[3] - A_local_cube[0][3] * u_n;

    // Решение СЛАУ Ax=b
    std::vector<double> x = matrix.tridiagonal_matrix_algorithm(b);
    x[elements_count] = u_n;
    return x;
}

std::vector<double> fast_cube() {

    // Приведение с момощью метода Гаусса локальной матрицы к виду,
    // необходимому для исключения внутренних элементов
    for (size_t i = 1; i < 3; i++) {
        for (size_t j = 0; j < 4; j++) {
            if (i == j | fabs(A_local_cube[j][i]) < 1e-16) {
                continue;
            }
            double piv = A_local_cube[j][i] / A_local_cube[i][i];
            b_local_cube[j] -= piv * b_local_cube[i];
            for (size_t k = 0; k < 4; k++) {
                A_local_cube[j][k] -= piv * A_local_cube[i][k];
            }
        }
    }

    // Получение матрицы A ансамблированием и учет граничных условий
    Matrix<double> matrix(4, 4, 0);
    for (size_t i = 0; i < 3; i++) {
        matrix.get_elem(i, i) += A_local_cube[0][0];
        matrix.get_elem(i + 1, i) += A_local_cube[3][0];
        matrix.get_elem(i, i + 1) += A_local_cube[0][3];
        matrix.get_elem(i + 1, i + 1) += A_local_cube[3][3];
    }
    matrix.get_elem(3, 3) = 1;
    matrix.get_elem(2, 3) = 0;

    // Получение вектора b и учет граничных условий
    std::vector<double> b(4);
    b[0] = b_local_cube[0] - du_0;
    b[1] = b_local_cube[0] + b_local_cube[3];
    b[2] = b_local_cube[0] + b_local_cube[3] - A_local_cube[0][3] * u_n;
    b[3] = b_local_cube[3] - A_local_cube[3][3] * u_n;

    // Решение СЛАУ Ax=b
```

```
    std::vector<double> x = matrix.fast_tridiagonal_matrix_algorithm(b);
    x[elements_count] = u_n;
    return x;
}

void print_graph(std::vector<double> &res_y, std::string graph_mame) {
    std::vector<double> x(elements_count + 1);
    for (size_t i = 0; i < res_y.size(); i++) {
        x[i] = start_x + i * element_len;
    }
    std::vector<double> y = solution();

    FILE* gnuplot = popen("gnuplot -persist", "w");

    fprintf(gnuplot, "$mfe_res << EOD\n");
    for (size_t i = 0; i < elements_count + 1; i++) {
        fprintf(gnuplot, "%lf %lf\n", x[i], res_y[i]);
    }
    fprintf(gnuplot, "EOD\n");

    fprintf(gnuplot, "$exact << EOD\n");
    for (size_t i = 0; i < elements_count + 1; i++) {
        fprintf(gnuplot, "%lf %lf\n", x[i], y[i]);
    }
    fprintf(gnuplot, "EOD\n");

    fprintf(gnuplot, "set grid\n set title '%s' font \"Helvetica,16\" lt 3 lw 5\n", graph_mame.c_str());
    fprintf(gnuplot, "plot '$mfe_res' using 1:2 with lines title 'MFE %zu elements' lc rgb \"blue\" lw 1, '$exact'
using 1:2 with lines title 'exact %zu elements',\n", elements_count, elements_count);
    fflush(gnuplot);
}

double max_error(std::vector<double> y) {
    std::vector<double> exact_y = solution();
    double max = 0;
    for (size_t i = 0; i < y.size(); i++) {
        double error = fabs(y[i] - exact_y[i]);
        if (error > max) {
            max = error;
        }
    }
    return max;
}

int main() {
    std::vector<double> res;
    if (!is_cube) {
        res = fast_linear();
        //print_graph(res, std::string("Linear"));
    } else {
        res = fast_cube();
        //print_graph(res, std::string("Cube"));
```

```
    }

    std::cout << "Max error between points " << max_error(res) << std::endl;
    return 0;
}
```