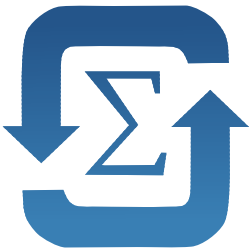
Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра прикладной математики

Лабораторная работа №2
по дисциплине «Уравнения математической физики»

Решение эллиптических краевых задач методом конечных разностей



Факультет: ПМИ
Группа: ПМ-63
Студенты: Шепрут И.И.
Вариант: 5
Преподаватель: Патрушев И.И.

Новосибирск
2019

1 Цель работы

Разработать программу решения нелинейной одномерной краевой задачи методом конечных элементов. Провести сравнение метода простой итерации и метода Ньютона для решения данной задачи.

2 Задание

1. Выполнить конечноэлементную аппроксимацию исходного уравнения в соответствии с заданием. Получить формулы для вычисления компонент матрицы A и вектора правой части b для метода простой итерации.
2. Реализовать программу решения нелинейной задачи методом простой итерации с учетом следующих требований:
   • язык программирования C++ или Фортран;
   • предусмотреть возможность задания неравномерных сеток по пространству и по времени, разрывность параметров уравнения по подобластям, учет краевых условий;
   • матрицу хранить в ленточном формате, для решения СЛАУ использовать метод LU - разложения;
   • предусмотреть возможность использования параметра релаксации.
3. Протестировать разработанную программу.
4. Провести исследования реализованных методов на различных зависимостях коэффициента от решения (или производной решения) в соответствии с заданием. На одних и тех же задачах сравнить по количеству итераци метод простой итерации. Исследовать скорость сходимости от параметра релаксации.

   Вариант 5: уравнение $-\operatorname{div}(\lambda(u)\operatorname{grad} u) + \sigma\frac{\partial u}{\partial t} = f$. Базисные функции - линейные.

3 Исследования

Далее под точностью решения будет подразумеваться L_2 норма между вектором q, полученным в ходе решения, на последнем моменте времени, и между реальным значением узлов, которые мы знаем, задавая функцию u. В исследованиях на порядок сходимости эта норма будет ещё делиться на число элементов, для нахождения среднего отклонения от идеального решения.

3.1 Точность для разных функций

Здесь показана точность решения и количество итераций в зависимости от функций $u(x,t)$ и $\lambda(u)$. Запускается со следующими параметрами:
• $sigma = 1$.
• $\varepsilon = 0.001$.
• $\text{iters}_{max} = 500$.
• Функция правой части высчитывается автоматически.
• Сетка по пространству равномерная: $(1, 1.1, \dots, 1.9, 2)$. Сетка по времени равномерная: $(0, 0.1, \dots, 0.9, 1)$.
• Начальное приближение: для функций u, линейных по t — $(1, 1, \dots)$.
• Для функций u, нелинейных по t начальное приближение в момент $t = 0$ — $(u(1,0), u(1.1,0), \dots, u(1.9,0), u(2,0))$, то есть истинное решение.

$u(x,t)$ \ $\lambda(u)$	1	u	u^2	u^2+1	u^3	u^4	e^u	$\sin u$
$3x+t$	10 0.01	20 $0.28 \cdot 10^{-7}$	48 $0.44 \cdot 10^{-4}$	46 $0.38 \cdot 10^{-4}$	51 $0.63 \cdot 10^{-3}$	62 $0.67 \cdot 10^{-3}$	86 $0.1 \cdot 10^{-2}$	2704 19
$2x^2+t$	10 0.01	40 $0.35 \cdot 10^{-3}$	53 $0.31 \cdot 10^{-2}$	50 $0.27 \cdot 10^{-2}$	72 $0.4 \cdot 10^{-2}$	5010 20	16 $2.4 \cdot 10^{5}$	5010 $2.8 \cdot 10^{2}$
x^3+t	10 $0.75 \cdot 10^{-2}$	39 $0.13 \cdot 10^{-2}$	64 $0.84 \cdot 10^{-2}$	58 $0.61 \cdot 10^{-2}$	106 0.01	5010 18	12 $3.8 \cdot 10^{5}$	5010 62
x^4+t	10 0.014	49 $0.46 \cdot 10^{-2}$	70 0.044	64 0.037	3074 0.061	5010 29	5010 *nan*	5010 $4.5 \cdot 10^{2}$
e^x+t	10 0.01	36 $0.18 \cdot 10^{-3}$	46 $0.99 \cdot 10^{-3}$	45 $0.87 \cdot 10^{-3}$	55 $0.29 \cdot 10^{-2}$	70 $0.71 \cdot 10^{-2}$	24 $1.2 \cdot 10^{5}$	5010 $1.1 \cdot 10^{2}$
$3x+t^2$	10 0.38	17 0.062	38 0.011	38 0.01	46 $0.19 \cdot 10^{-2}$	56 $0.3 \cdot 10^{-3}$	64 $0.38 \cdot 10^{-3}$	5010 63
$3x+t^3$	10 1.4	20 0.22	36 0.04	36 0.039	43 $0.74 \cdot 10^{-2}$	49 $0.6 \cdot 10^{-3}$	61 $0.38 \cdot 10^{-2}$	5010 $2.4 \cdot 10^{4}$
$3x+e^t$	10 0.65	20 0.081	40 0.011	40 0.011	40 $0.19 \cdot 10^{-2}$	50 $0.31 \cdot 10^{-3}$	74 $0.15 \cdot 10^{-2}$	5010 48
$3x+sin(t)$	10 0.17	11 0.029	38 $0.51 \cdot 10^{-2}$	38 $0.49 \cdot 10^{-2}$	45 $0.67 \cdot 10^{-3}$	56 $0.42 \cdot 10^{-3}$	68 $0.11 \cdot 10^{-2}$	5010 24
e^x+t^2	10 0.38	29 0.06	38 0.013	38 0.012	51 $0.27 \cdot 10^{-2}$	64 $0.32 \cdot 10^{-2}$	81 0.011	5010 $2.7 \cdot 10^{2}$
e^x+t^3	10 1.4	27 0.22	35 0.044	35 0.042	45 $0.86 \cdot 10^{-2}$	59 $0.26 \cdot 10^{-2}$	71 0.022	5010 $2.2 \cdot 10^{3}$
e^x+e^t	10 0.65	30 0.081	40 0.012	40 0.012	50 $0.3 \cdot 10^{-2}$	60 $0.24 \cdot 10^{-2}$	98 0.025	5010 $7.7 \cdot 10^{3}$
$e^x+sin(t)$	10 0.17	30 0.028	39 $0.52 \cdot 10^{-2}$	39 $0.49 \cdot 10^{-2}$	50 $0.28 \cdot 10^{-2}$	64 $0.95 \cdot 10^{-2}$	82 0.032	5010 $2.5 \cdot 10^{3}$

3.2 Зависимость точности от нелинейной сетки

3.2.1 Функции нелинейной сетки

В ходе выполнения лабораторной работы были обнаружены функции, позволяющие легко задавать неравномерную сетку, сгущающуюяся к одному из концов.

Если у нас задано начало — a и конец сетки — b, а количество элементов n, тогда сетку можно задать следующим образом:

$$x_i = a + m\left(\frac{i}{n}\right) \cdot (b-a), i = \overline{0,n}$$

,

где $m(x)$ — некоторая функция, задающая неравномерную сетку. При этом x обязан принадлежать области $[0,1]$, а функция m возвращать значения из той же области, и при этом быть строго монотонной на этом участке. Тогда гарантируется условие на сетке, что $x_j \leqslant x_i$ при $j \leqslant i$.

Пример: при $m(x) = x$, сетка становится равномерной.

Найденные функции зависят от некоторого параметра t:

$$m_{1,t}(x) = x^t \qquad m_{2,t}(x) = x^{\frac{1}{t}}$$
$$m_{3,t}(x) = \frac{t^x-1}{t-1} \qquad m_{4,t}(x) = \frac{\frac{1}{t^x}}{\frac{1}{t}-1}$$

Что интересно, эти функции вырождаются в x при $t = 1$, а при $t = 0$, они вырождаются в сетку, полностью находящуюся на одном из концов: 1, 3 фукнции стремятся к концу b; а функции 2, 4 стремятся к концу a. 1 и 2 функции симметричны друг другу, как 3 и 4.

Таким образом, можно исследовать различные неравномерные сетки на итоговую точность и число итераций, изменяя параметр от 0 до 1.

3.2.2 Описание исследований

Параметры остаются прежними, с небольшими изменениями:

- $\text{iters}_{max} = 100$.
- Сетка по пространству неравномерная, если исследование происходит по сетке пространству, и равномерная, если исследование происходит по сетке времени.

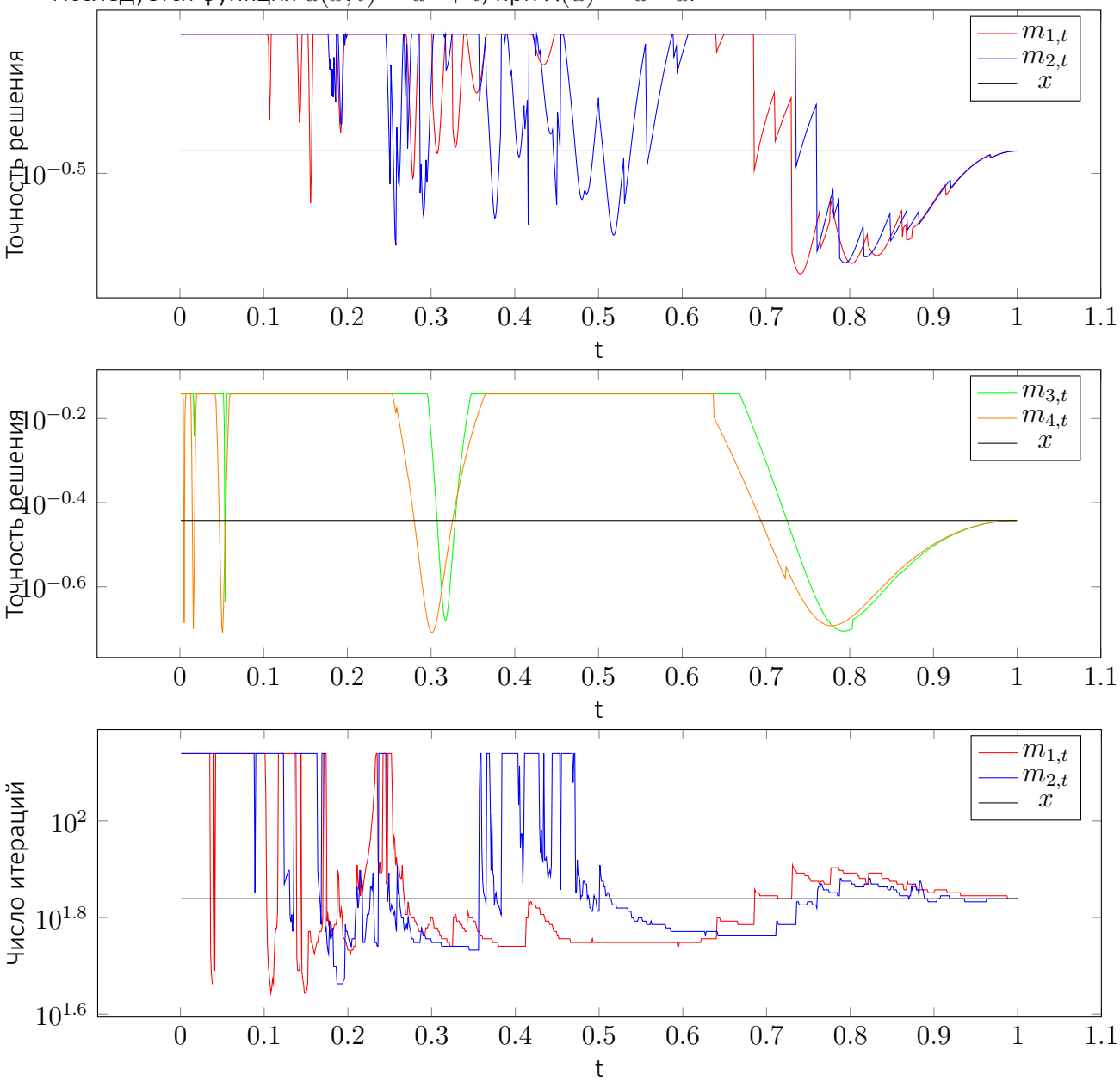
Исследуется скорость и качество сходимости в зависимости от параметра неравномерной сетки. Так же некоторые графики разделены для того, чтобы можно было что-то различить на них.

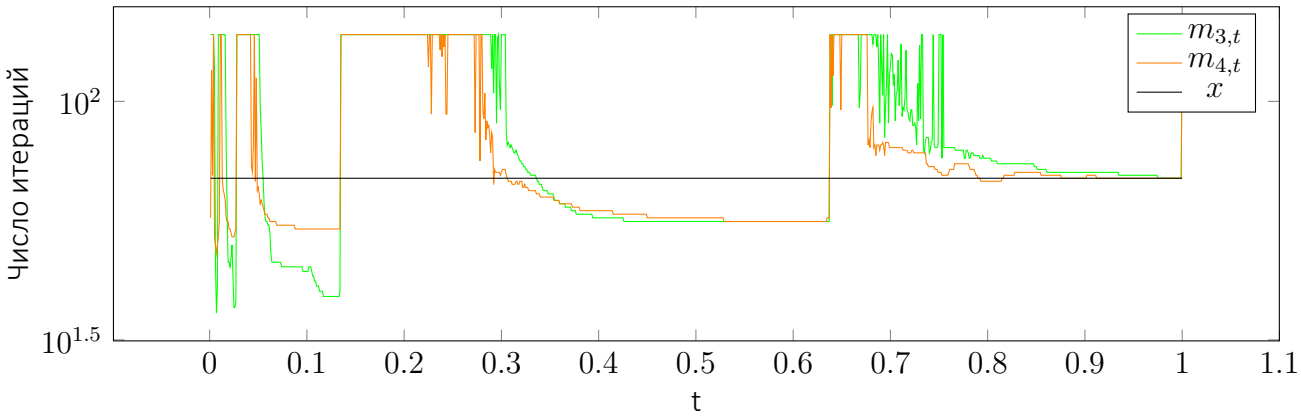
3.2.3 Сетка по пространству

В данных исследованиях неравномерность применяется к сетке по пространству.

3.2.3.1 u = x4 + t

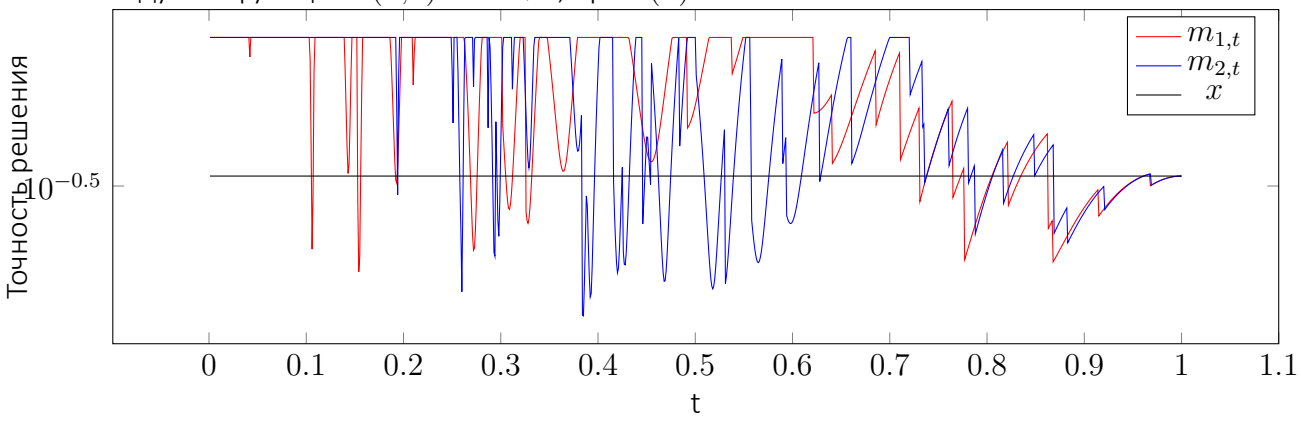
Исследуется функция $u(x,t) = x^4 + t$, при $\lambda(u) = u * u$.

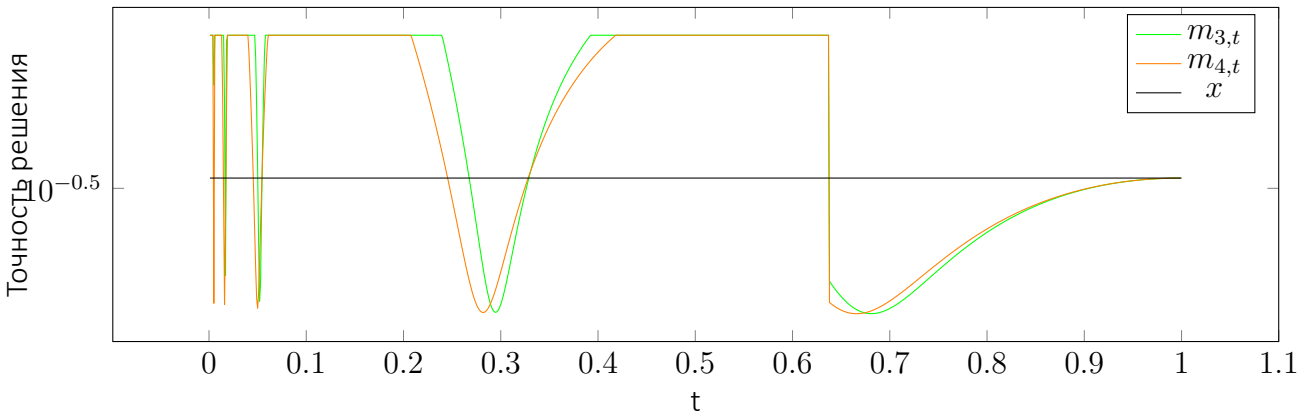


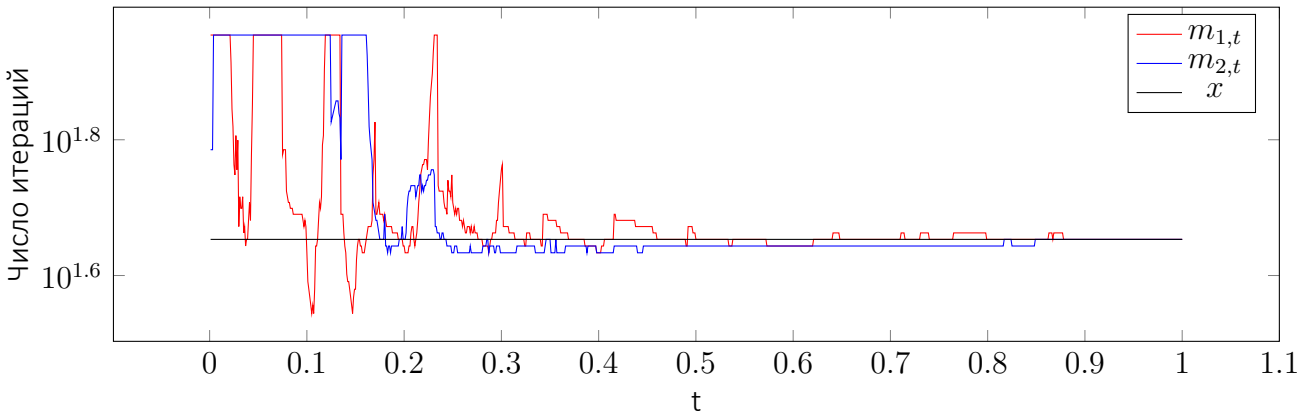


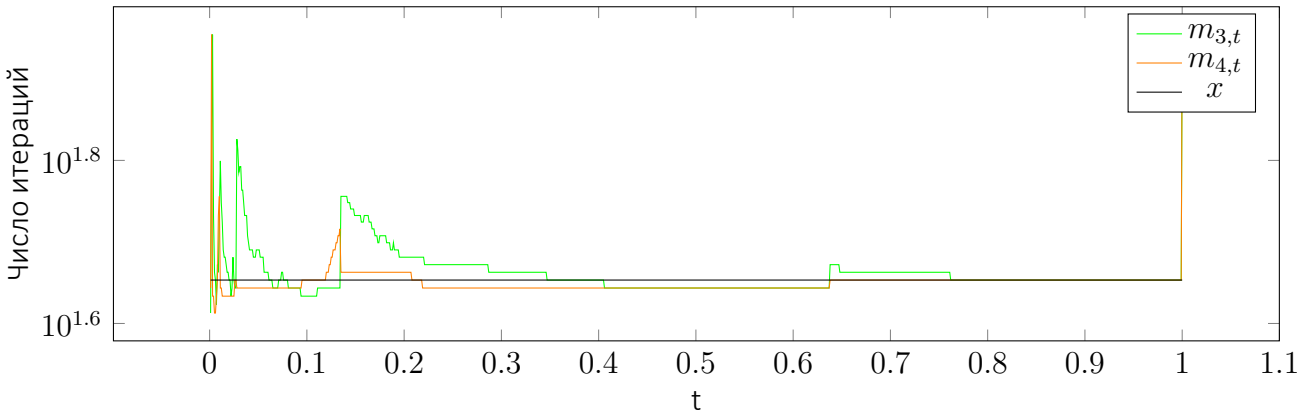
3.2.3.2 u = exp(x) + t

Исследуется функция $u(x,t) = e^x + t$, при $\lambda(u) = u * u$.







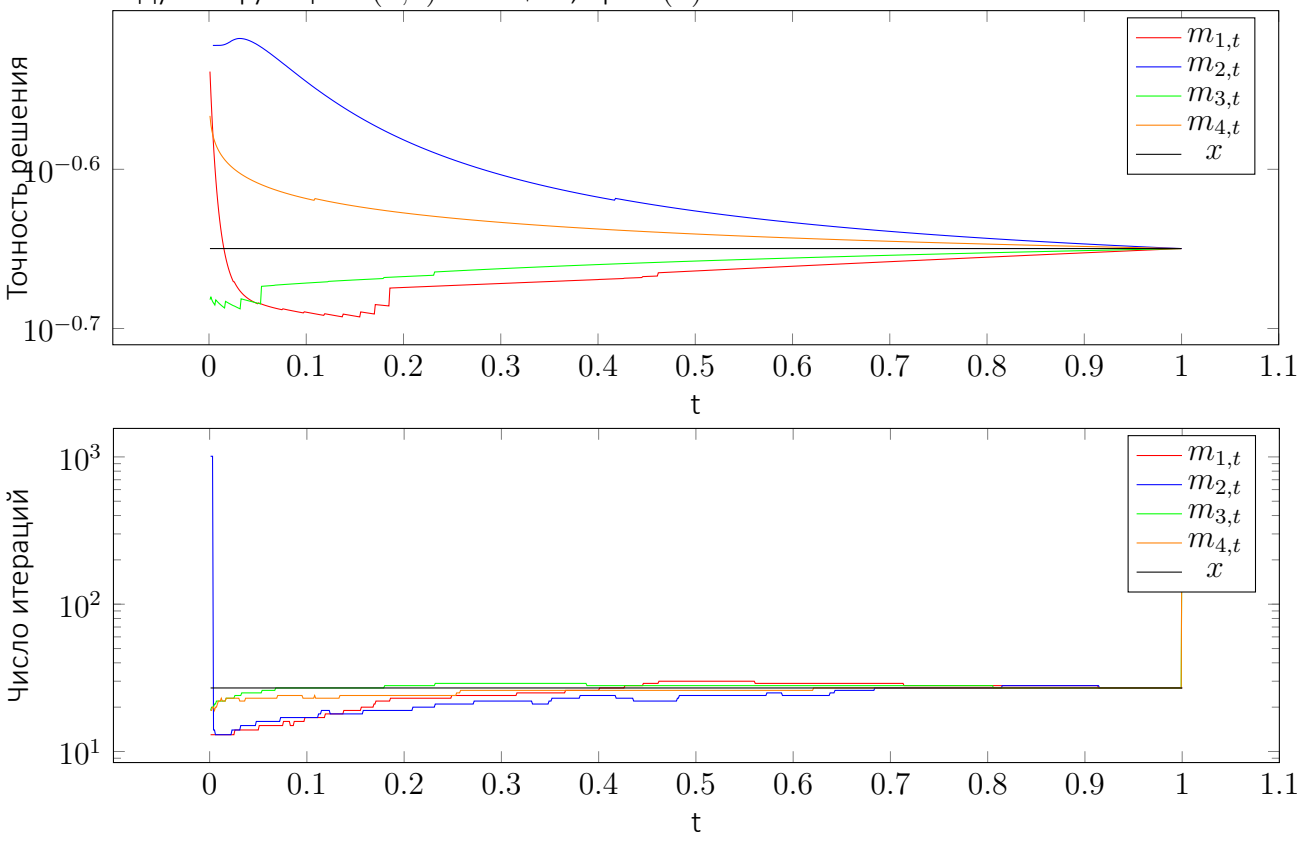


3.2.4 Сетка по времени

В данных исследованиях неравномерность применяется к сетке по времени.

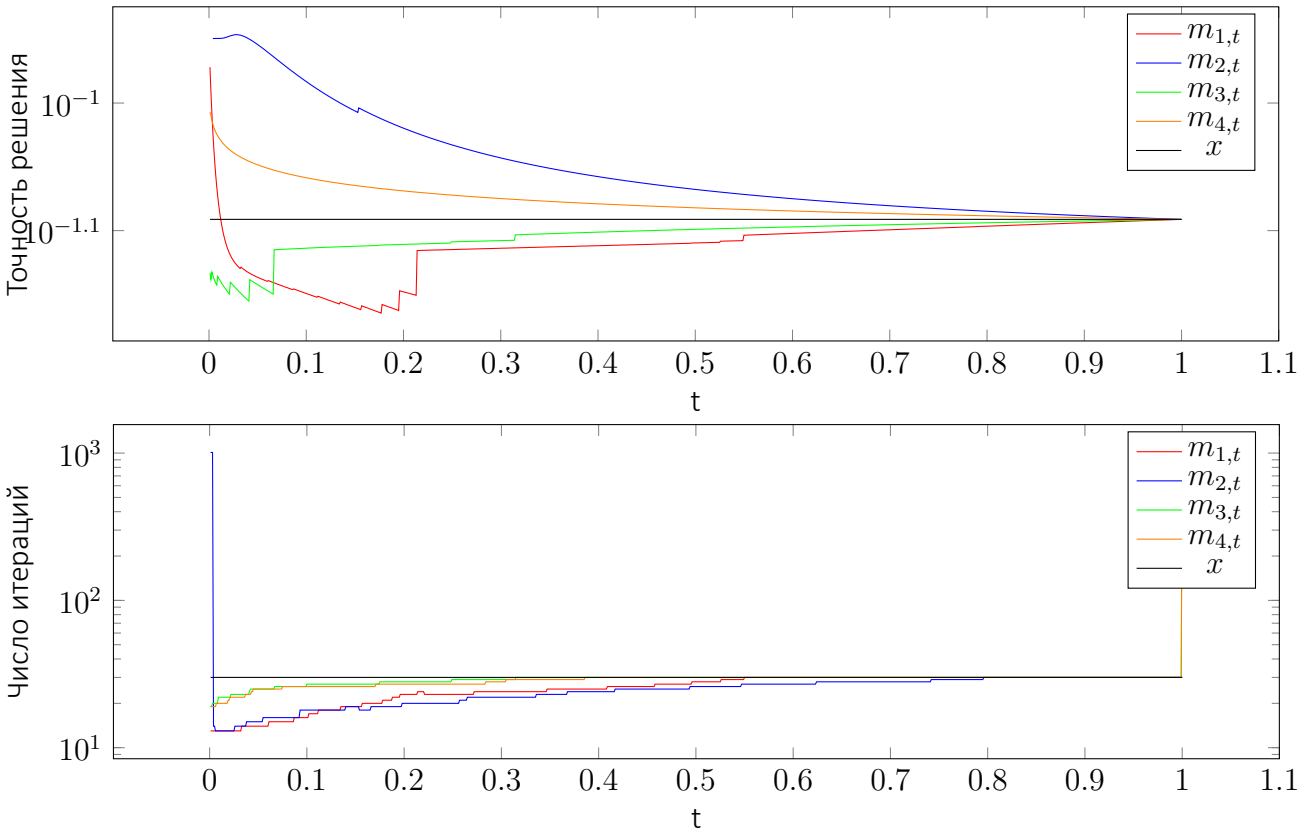
3.2.4.1 u = exp(x) + t3

Исследуется функция $u(x,t) = e^x + t^3$, при $\lambda(u) = u$.



3.2.4.2 u = exp(x) + exp(t)

Исследуется функция $u(x,t) = e^x + e^t$, при $\lambda(u) = u$.

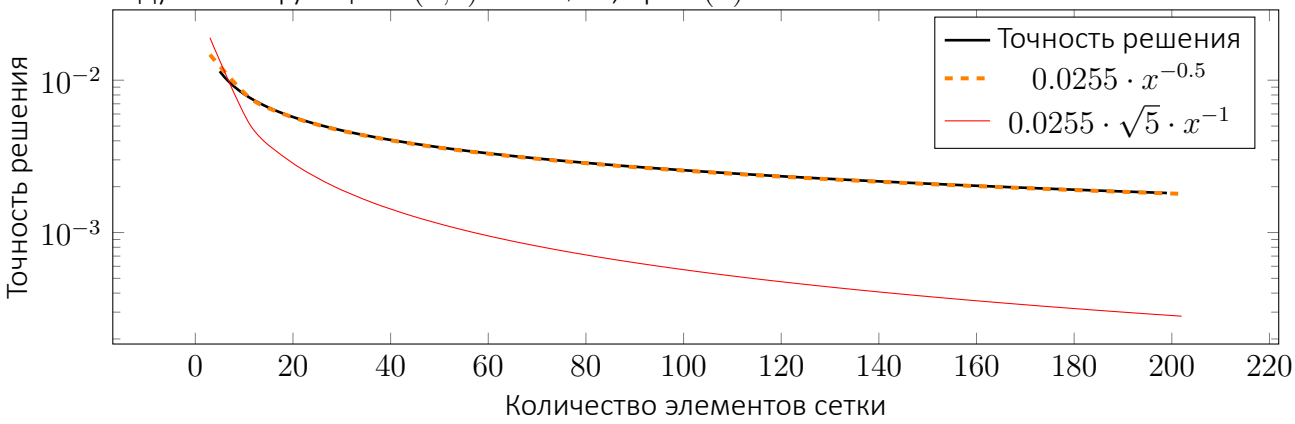


3.3 Точность в зависимости от размера сетки

В данном пункте определяется *порядок сходимости*. Он равен тому числу, которое стоит в степени функции x, которая хорошо аппроксимирует данную зависимость.

3.3.1 Сетка по пространству

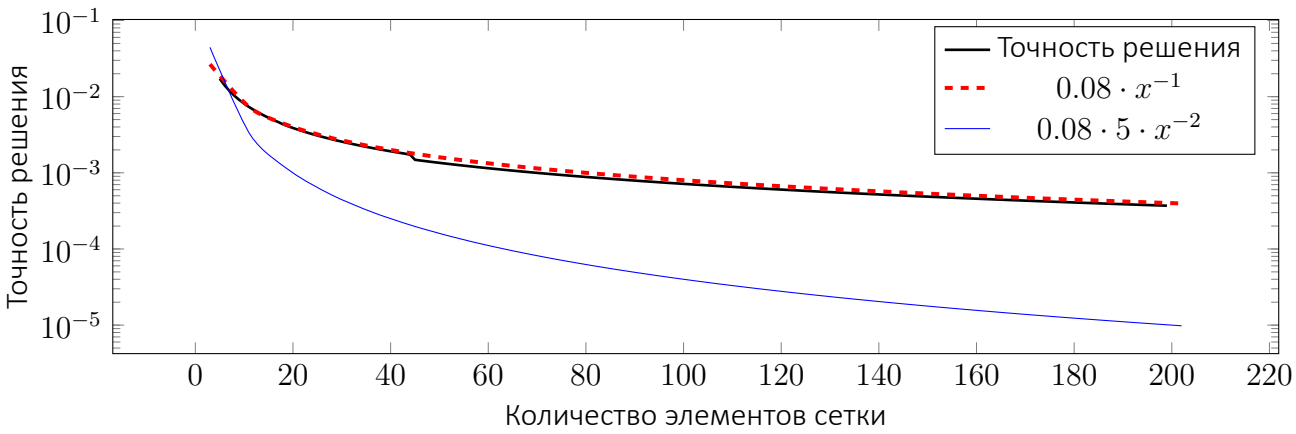
Исследуется на функции $u(x,t) = e^x + e^t$, при $\lambda(u) = u$.



Вывод: порядок сходимости по пространственной сетке равен 0.5.

3.3.2 Сетка по времени

Исследуется на функции $u(x,t) = e^x + e^t$, при $\lambda(u) = u$.



Вывод: порядок сходимости по временной сетке равен 1.

4 Выводы

● Порядок аппроксимации метода конечных элементов с линейными элементами с первыми краевыми условиями равен ? при нелинейности $\lambda = 1$, и ? при нелинейности $\lambda = u$.
● TODO...
● TODO...
● TODO...
● TODO...

5 Код программы

FILE **main.cpp**

```cpp
#include <iostream>
#include <algorithm>
#include <functional>
#include <iterator>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <Eigen/Dense>

using namespace std;

typedef Eigen::MatrixXd Matrix;
typedef Eigen::VectorXd Vector;

typedef function<double(double)> Function1D;
typedef function<double(double, double)> Function2D;
typedef function<double(double, double, double)> Function3D;

//---------------------------------------------------------------------------
double integral_gauss3(const vector<double>& X, Function1D f) {
    const double x1 = -sqrt(3.0/5.0);
    const double x2 = 0;
    const double x3 = -x1;
    const double q1 = 5.0/9.0;
    const double q2 = 8.0/9.0;
    const double q3 = q1;
    double sum = 0;
    double xk = 0;
    double h = X[1] - X[0];
    double h2 = h/2.0;

    for (int i = 0; i < X.size()-1; ++i) {
        xk = (X[i]+X[i+1])/2.0;
        sum += q1 * f(xk + x1 * h2);
        sum += q2 * f(xk + x2 * h2);
        sum += q3 * f(xk + x3 * h2);
    }
```

```
    sum *= h;
    sum /= 2.0;
    return sum;
}

//---------------------------------------------------------------------------
void make_grid(vector<double>& X, double a, double b, long n, Function1D move = [] (double x) ->
↪  double {return x;}) {
    X.clear();
    double size = b-a;
    for (double i = 0; i <= n; ++i)
        X.push_back(a + move(i/double(n)) * size);
}

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
double basicFunction(double left, double middle, double right, double x) {
    // Базовая линейная функция
    if (x > left && x < right) {
        if (x < middle) {
            return (x-left)/(middle-left);
        } else {
            return 1-(x-middle)/(right-middle);
        }
    }

    return 0;
}

//---------------------------------------------------------------------------
double basicFunctionGrad(double left, double middle, double right, double x) {
    // Базовая линейная функция
    if (x > left && x < right) {
        if (x < middle) {
            return 1.0/(middle-left);
        } else {
            return -1.0/(right-middle);
        }
    }

    return 0;
}

//---------------------------------------------------------------------------
Function1D calcFirstDerivative(const Function1D& f) {
    return [f](double x) -> double {
        const double h = 0.001;
        return (-f(x + 2 * h) + 8 * f(x + h) - 8 * f(x - h) + f(x - 2 * h)) / (12 * h);
    };
}

//---------------------------------------------------------------------------
Function2D calcRightPart(
    const Function1D& lambda,
    const Function2D& u,
    const Function1D& sigma
) {
    // f = -div(lambda(u) * grad u) + sigma * du/dt
    return [=](double x, double t) -> double {
        using namespace placeholders;
        auto ut = calcFirstDerivative(bind(u, x, _1));
        auto ux = calcFirstDerivative(bind(u, _1, t));
        auto lambda_grad = [lambda, ux, u](double x, double t) -> double {
            return lambda(u(x, t)) * ux(x);
        };
        auto div = calcFirstDerivative(bind(lambda_grad, _1, t));
        return -div(x) + sigma(x) * ut(x);
    };
}

//---------------------------------------------------------------------------
double norm(const vector<double>& a, const vector<double>& b) {
    assert(a.size() == b.size());
    double sum = 0;
    for (int i = 0; i < a.size(); i++)
        sum += (a[i] - b[i])*(a[i] - b[i]);
    return sqrt(sum);
}

//---------------------------------------------------------------------------
ostream& operator<<(ostream& out, const vector<double>& mas) {
    for (auto& i : mas)
        out << i << " ";
    return out;
}
```

```
//--------------------------------------------------------------------------------
string write_for_latex_double(double v, int precision) {
    int power = log(std::fabs(v)) / log(10.0);
    double value = v / pow(10.0, power);

    if (v != v) return "nan";

    if (v == 0) {
        power = 0;
        value = 0;
    }

    stringstream sout;
    sout.precision(precision);
    if (power == -1 || power == 0 || power == 1) {
        sout << v;
    } else {
        sout << value << "\\cdot 10^{" << power << "}";
    }

    return sout.str();
}

//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------

//--------------------------------------------------------------------------------
class lin_approx_t
{
public:
    vector<double> q; /// Вектор весов
    vector<double> x; /// Массив положений каждого элемента

    int size(void) const {
        return x.size();
    }

    double left(int i) const {
        if (i == 0)
            return x[0]-0.00001;
        else
            return x[i-1];
    }

    double middle(int i) const {
        return x[i];
    }

    double right(int i) const {
        if (i == x.size() - 1)
            return x.back()+0.00001;
        else
            return x[i+1];
    }

    double value(double pos) const {
        int start = distance(x.begin(), lower_bound(x.begin(), x.end(), pos))-1;
        if (start == -1) {
            if (fabs(pos-x[0]) < 0.00001)
                return q[0];
            else
                return 0;
        }
        if (start == x.size()) {
            if (fabs(pos-x.back()) < 0.00001)
                return q.back();
            else
                return 0;
        }
        double sum = 0;
        for (int i = start; i < min<int>(start+2, x.size()); ++i)
        //for (int i = 0; i < x.size(); ++i)
            sum += q[i] * basic(pos, i);
        return sum;
    } /// Получить значение функции аппроксимации в некоторой точке

    double basic(double pos, int i) const {
        return basicFunction(left(i), middle(i), right(i), pos);
    } /// Получить значение базовой функции под номером i в точке pos

    double basic_grad(double pos, int i) const {
        return basicFunctionGrad(left(i), middle(i), right(i), pos);
    } /// Получить значение производной базовой функции под номером i в точке pos
};

//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
```

```cpp
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
double calc_a1_integral(const Function1D& lambda, const lin_approx_t& u, int i, int j) {
    if (abs(i - j) > 1)
        return 0;
    auto f = [&] (double x) -> double {
        return lambda(u.value(x)) * u.basic_grad(x, i) * u.basic_grad(x, j);
    };
    vector<double> X;
    if (i != j)
        make_grid(X, min(u.middle(i), u.middle(j)), min(u.right(i), u.right(j)), 10);
    else
        make_grid(X, u.left(i), u.right(i), 10);
    return integral_gauss3(X, f);
}

//------------------------------------------------------------------------------
double calc_a2_integral(const Function1D& sigma, const lin_approx_t& u, int i, int j) {
    if (abs(i - j) > 1)
        return 0;
    auto f = [&] (double x) -> double {
        return sigma(x) * u.basic(x, i) * u.basic(x, j);
    };
    vector<double> X;
    if (i != j)
        make_grid(X, min(u.middle(i), u.middle(j)), min(u.right(i), u.right(j)), 10);
    else
        make_grid(X, u.left(i), u.right(i), 10);
    return integral_gauss3(X, f);
}

//------------------------------------------------------------------------------
double calc_b1_integral(const Function1D& f, const lin_approx_t& u, int i) {
    auto fun = [&] (double x) -> double {
        return f(x) * u.basic(x, i);
    };
    vector<double> X;
    make_grid(X, u.left(i), u.right(i), 10);
    return integral_gauss3(X, fun);
}

//------------------------------------------------------------------------------
double calc_b2_integral(const Function1D& sigma, const lin_approx_t& u, const lin_approx_t&
↪  u_last_time, int i) {
    auto fun = [&] (double x) -> double {
        return sigma(x) * u_last_time.value(x) * u.basic(x, i);
    };
    vector<double> X;
    make_grid(X, u.left(i), u.right(i), 10);
    return integral_gauss3(X, fun);
}

//------------------------------------------------------------------------------
lin_approx_t approximate_function(Function1D lambda, const lin_approx_t& u) {
    lin_approx_t result = u;
    for (int i = 0; i < result.q.size(); ++i)
        result.q[i] = lambda(u.x[i]);
    return result;
}

//------------------------------------------------------------------------------
Matrix calcA(const lin_approx_t& u_last, double dt, Function1D lambda, Function1D sigma) {
    Matrix result(u_last.size(), u_last.size());
    for (int i = 0; i < u_last.size()-1; ++i) {
        for (int j = 0; j < u_last.size(); ++j) {
            result(i, j) = calc_a1_integral(lambda, u_last, i, j) + calc_a2_integral(sigma, u_last, i,
            ↪  j) / dt;
        }
    }

    return result;
} /// Рассчитать матрицу A(q)

//------------------------------------------------------------------------------
Vector calcB(const lin_approx_t& u_last, double dt, Function1D f, Function1D sigma, const
↪  lin_approx_t& u_last_time) {
    Vector result(u_last.size());
    for (int i = 0; i < u_last.size(); ++i) {
        result(i) = calc_b1_integral(f, u_last, i) + calc_b2_integral(sigma, u_last, u_last_time, i) /
        ↪  dt;
    }

    return result;
} /// Рассчитать вектор правой части b(q)

//------------------------------------------------------------------------------
```

```
void write_first_boundary_conditions(Matrix& A, Vector& b, const lin_approx_t& u, Function2D u_true,
    double time) {
    A(0, 0) = 1;
    for (int i = 1; i < A.cols(); ++i) A(0, i) = 0;
    b(0) = u_true(u.x.front(), time);

    A(A.rows()-1, A.cols()-1) = 1;
    for (int i = 0; i < A.cols() - 1; ++i) A(A.rows()-1, i) = 0;
    b(b.rows()-1) = u_true(u.x.back(), time);
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
struct Result
{
    enum ExitType {
        EXIT_RESIDUAL,
        EXIT_STEP,
        EXIT_ITERATIONS,
        EXIT_ERROR
    } exitType;

    lin_approx_t answer;
    int iterations;
    double residual;
};

//-----------------------------------------------------------------------------
ostream& operator<<(ostream& out, const Result::ExitType& exit) {
    switch (exit) {
        case Result::EXIT_RESIDUAL: out << "residual"; break;
        case Result::EXIT_STEP: out << "step"; break;
        case Result::EXIT_ITERATIONS: out << "iterations"; break;
        case Result::EXIT_ERROR: out << "error"; break;
    }
    return out;
}

//-----------------------------------------------------------------------------
Result solveFixedPointIteration(
    Function2D f,
    Function2D u_true,
    Function1D lambda,
    Function1D sigma,
    const lin_approx_t& u_last_time,
    double dt,
    double time,
    double eps,
    int maxiter
) {
    lin_approx_t u = u_last_time;

    Vector last_x(u.q.size());
    for (int i = 0; i < u.q.size(); i++)
        last_x(i) = u.q[i];

    Result result;
    result.iterations = 0;
    while (true) {
        auto A = calcA(u, dt, lambda, sigma);
        auto b = calcB(u, dt, bind(f, placeholders::_1, time), sigma, u_last_time);
        write_first_boundary_conditions(A, b, u, u_true, time);

        Eigen::JacobiSVD<Matrix> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);
        Vector x = svd.solve(b);
        for (int i = 0; i < u.size(); ++i)
            u.q[i] = x(i);

        result.iterations++;

        if (result.iterations > maxiter) {
            result.exitType = Result::EXIT_ITERATIONS;
            break;
        }

        {
            auto A1 = calcA(u, dt, lambda, sigma);
            auto b1 = calcB(u, dt, bind(f, placeholders::_1, time), sigma, u_last_time);
            write_first_boundary_conditions(A1, b1, u, u_true, time);

            double au = (A1 * x - b1).norm();
            double bu = b1.norm();
            result.residual = au/bu;
            if (au/bu < eps) {
                result.exitType = Result::EXIT_RESIDUAL;
                break;
```

```
            }
        }

        if ((x-last_x).norm() < eps) {
            result.exitType = Result::EXIT_STEP;
            break;
        }

        last_x = x;
    }

    result.answer = u;

    return result;
} /// Решить уравнение методом простой итерации

lin_approx_t calcTrulyApprox(const vector<double>& grid, const Function2D& u_true, double time) {
    lin_approx_t u_truly_approx;
    u_truly_approx.x = grid;
    u_truly_approx.q = grid;
    for (int i = 0; i < u_truly_approx.x.size(); i++)
        u_truly_approx.q[i] = u_true(u_truly_approx.x[i], time);
    return u_truly_approx;
}

//-----------------------------------------------------------------------------
void write_iterations_information(double t0, const Result& result, const Function2D& u_true) {
    cout << "time: " << t0 << endl;
    cout << "residual: " << result.residual << endl;
    cout << "iterations: " << result.iterations << endl;
    cout << "answer:   " << result.answer.q << endl;

    lin_approx_t u_truly_approx = calcTrulyApprox(result.answer.x, u_true, t0);
    cout << "shold be: " << u_truly_approx.q << endl;

    cout << "norm: " << norm(u_truly_approx.q, result.answer.q) << endl;

    cout << endl << "----------------------------" << endl;
}

//-----------------------------------------------------------------------------
vector<Result> solveByTime(
    Function2D f,
    Function2D u_true,
    Function1D lambda,
    Function1D sigma,
    const lin_approx_t& u_start,
    const vector<double>& time_grid,
    double eps,
    double maxiter
) {
    vector<Result> res;
    lin_approx_t u = u_start;

    for (int i = 1; i < time_grid.size(); ++i) {
        double t0 = time_grid[i];
        double t1 = time_grid[i-1];
        auto result = solveFixedPointIteration(f, u_true, lambda, sigma, u, t0-t1, t0, eps, maxiter);
        res.push_back(result);

        // Выводим мета-информацию
        //write_iterations_information(t0, result, u_true);

        u = result.answer;
    }

    return res;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
template<class A, class B, class C>
struct triple {
    A first;
    B second;
    C third;
};

//-----------------------------------------------------------------------------
vector<triple<Function2D, string, bool>> us;
vector<pair<Function1D, string>> lambdas;
vector<Function2D> moves;
auto sigma = [] (double x) -> double { return 1; };

double a = 1, b = 2, n = 10; // Характеристики сетки по пространству
double at = 0, bt = 1, nt = 10; // Характеристики сетки по времени
```

```
double na = 0, nb = 1, nn = 1000; // Характеристики сетки по параметру t, в зависимости от которого меняются неравномерные сетки
int sa = 5, sb = 200; // Характеристики сетки по размеру

//-----------------------------------------------------------------------------
void init() {
    us.push_back({[] (double x, double t) -> double { return 3*x + t; }, "$3x+t$", false});
    us.push_back({[] (double x, double t) -> double { return 2*x*x + t; }, "$2x^2+t$", false});
    us.push_back({[] (double x, double t) -> double { return x*x*x + t; }, "$x^3+t$", false});
    us.push_back({[] (double x, double t) -> double { return x*x*x*x + t; }, "$x^4+t$", false});
    us.push_back({[] (double x, double t) -> double { return exp(x) + t; }, "$e^x+t$", false});
    us.push_back({[] (double x, double t) -> double { return 3*x + t*t; }, "$3x+t^2$", true});
    us.push_back({[] (double x, double t) -> double { return 3*x + t*t*t; }, "$3x+t^3$", true});
    us.push_back({[] (double x, double t) -> double { return 3*x + exp(t); }, "$3x+e^t$", true});
    us.push_back({[] (double x, double t) -> double { return 3*x + sin(t); }, "$3x+sin(t)$", true});
    us.push_back({[] (double x, double t) -> double { return exp(x) + t*t; }, "$e^x+t^2$", true});
    us.push_back({[] (double x, double t) -> double { return exp(x) + t*t*t; }, "$e^x+t^3$", true});
    us.push_back({[] (double x, double t) -> double { return exp(x) + exp(t); }, "$e^x+e^t$", true});
    us.push_back({[] (double x, double t) -> double { return exp(x) + sin(t); }, "$e^x+sin(t)$", true}
);

    lambdas.push_back({[] (double u) -> double { return 1; }, "$1$"});
    lambdas.push_back({[] (double u) -> double { return u; }, "$u$"});
    lambdas.push_back({[] (double u) -> double { return u*u; }, "$u^2$"});
    lambdas.push_back({[] (double u) -> double { return u*u + 1; }, "$u^2+1$"});
    lambdas.push_back({[] (double u) -> double { return u*u*u; }, "$u^3$"});
    lambdas.push_back({[] (double u) -> double { return u*u*u*u; }, "$u^4$"});
    lambdas.push_back({[] (double u) -> double { return exp(u); }, "$e^u$"});
    lambdas.push_back({[] (double u) -> double { return sin(u); }, "sinu"});

    moves.push_back([] (double x, double t) -> double { return pow(x, t); });
    moves.push_back([] (double x, double t) -> double { return pow(x, 1.0/t); });
    moves.push_back([] (double x, double t) -> double { return (pow(t, x)-1.0)/(t-1.0); });
    moves.push_back([] (double x, double t) -> double { return (pow(1.0/t, x)-1.0)/(1.0/t-1.0); });
}

//-----------------------------------------------------------------------------
void writeFirstInvestigation() {
    ofstream fout("first.txt");
    fout << "a\t";
    for (auto& i : lambdas) fout << i.second << ((i.second != lambdas.back().second) ? "\t" : "");
    fout << endl;

    vector<double> time;
    make_grid(time, at, bt, nt);
    for (auto& i : us) {
        fout << i.second << "\t";
        auto u_true = i.first;
        auto sigma = [] (double x) -> double { return 1; };

        lin_approx_t u;
        make_grid(u.x, a, b, n);
        for (int j = 0; j < u.x.size(); j++)
            if (i.third)
                u.q.push_back(u_true(u.x[j], at));
            else
                u.q.push_back(1);

        lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);

        for (auto& j : lambdas) {
            auto lambda = j.first;

            auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u, time, 0.001, 500);

            int itersum = 0;
            for (auto& k : result) itersum += k.iterations;

            double residual = norm(u_truly_approx.q, result.back().answer.q);

            fout << "\\scalebox{.55}{\\tcell{$" << itersum << "$\\\\$" << write_for_latex_double(residual, 2) << "$}}" << ((j.second != lambdas.back().second) ? "\t" : "");
        }
        fout << endl;
    }

    fout.close();
}

//-----------------------------------------------------------------------------
void writeGridInvestigation(
    const Function2D& u_true, string su,
    const string& file,
    const Function1D& lambda, string slambda) {
    // Делаем равномерную сетку по времени
    vector<double> time;
```

```
    make_grid(time, at, bt, nt);

    pair<int, double> resu = {-1, 0};

    auto test_grid = [&resu, time, u_true, lambda] (const vector<double>& grid) -> pair<int, double> {
        lin_approx_t u; u.x = grid;
        for (int j = 0; j < u.x.size(); j++) u.q.push_back(1);
        lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);

        auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
        ↪  time, 0.001, 100);

        int itersum = 0; for (auto& k : result) itersum += k.iterations;
        double residual = norm(u_truly_approx.q, result.back().answer.q);
        if (residual > resu.second * 2 && resu.first != -1) residual = resu.second * 2;
        if (itersum > resu.first * 2 && resu.first != -1) itersum = resu.first * 2;
        return {itersum, residual};
    };

    ofstream fout(file);
    fout << "u = " << su << ", lambda = " << slambda << endl;

    // Получаем результаты для равномерной сетки
    vector<double> x_uniform;
    make_grid(x_uniform, a, b, n);
    resu = test_grid(x_uniform);

    fout << "t\tru\tiu";
    int counter = 0;
    for (auto& i : moves) {
        counter++;
        fout << "\tr" << counter << "\ti" << counter;
    }
    fout << endl;
    for (int i = 1; i <= nn; i++) {
        double t = na + (nb-na) * i/nn;
        cout << "\r" << 100 * t << "      ";
        fout << t << "\t" << resu.second << "\t" << resu.first;

        vector<double> grid;
        for (auto& move : moves) {
            make_grid(grid, a, b, n, bind(move, placeholders::_1, t));
            auto res = test_grid(grid);
            fout << "\t" << res.second << "\t" << res.first;
        }

        fout << endl;
    }
    cout << endl;

    fout.close();
}
//---------------------------------------------------------------------------
void writeGridInvestigationTime(
    const Function2D& u_true, string su,
    const string& file,
    const Function1D& lambda, string slambda) {
    pair<int, double> resu = {-1, 0};

    auto test_grid = [u_true, lambda, resu] (const vector<double>& grid) -> pair<int, double> {
        lin_approx_t u; make_grid(u.x, a, b, n);
        for (int j = 0; j < u.x.size(); j++) u.q.push_back(u_true(u.x[j], at));
        lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);

        auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
        ↪  grid, 0.001, 100);

        int itersum = 0; for (auto& k : result) itersum += k.iterations;
        double residual = norm(u_truly_approx.q, result.back().answer.q);
        if (residual > resu.second * 2 && resu.first != -1) residual = resu.second * 2;
        if (itersum > resu.first * 2 && resu.first != -1) itersum = resu.first * 2;
        return {itersum, residual};
    };

    ofstream fout(file);
    fout << "u = " << su << ", lambda = " << slambda << endl;

    // Получаем результаты для равномерной сетки
    vector<double> uniform;
    make_grid(uniform, at, bt, nt);
    resu = test_grid(uniform);

    fout << "t\tru\tiu";
    int counter = 0;
    for (auto& i : moves) {
        counter++;
        fout << "\tr" << counter << "\ti" << counter;
    }
```

```
    fout << endl;
    for (int i = 1; i <= nn; i++) {
        double t = na + (nb-na) * i/nn;
        cout << "\r" << 100 * t << "      ";
        fout << t << "\t" << resu.second << "\t" << resu.first;

        vector<double> grid;
        for (auto& move : moves) {
            make_grid(grid, at, bt, nt, bind(move, placeholders::_1, t));
            auto res = test_grid(grid);
            fout << "\t" << res.second << "\t" << res.first;
        }

        fout << endl;
    }

    cout << endl;

    fout.close();
}

//-----------------------------------------------------------------------------
void writeGridInvestigationSize(
    const Function2D& u_true, string su,
    const string& file,
    const Function1D& lambda, string slambda) {
    ofstream fout(file);
    fout << "u = " << su << ", lambda = " << slambda << endl;

    // Делаем равномерную сетку по времени
    vector<double> time;
    make_grid(time, at, bt, nt);

    fout << "size\titerations\tresidual" << endl;
    for (int size = sa; size < sb; ++size) {
        lin_approx_t u;
        make_grid(u.x, a, b, size);
        for (int j = 0; j < u.x.size(); j++) u.q.push_back(u_true(u.x[j], at));
        lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);

        auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
        ↪  time, 0.001, 100);

        int itersum = 0; for (auto& k : result) itersum += k.iterations;
        double residual = norm(u_truly_approx.q, result.back().answer.q);

        residual /= size; // Важно! Так как если не разделить, то расстояние между каждым узлом будет
        ↪  в итоге уменьшаться, но раз мы их делаем всё больше и больше, то в сумме оно будет
        ↪  увеличиваться, и в итоге точность будет расти.

        fout << size << "\t" << itersum << "\t" << residual << endl;
    }

    fout.close();
}

//-----------------------------------------------------------------------------
void writeGridInvestigationSizeTime(
    const Function2D& u_true, string su,
    const string& file,
    const Function1D& lambda, string slambda) {
    ofstream fout(file);
    fout << "u = " << su << ", lambda = " << slambda << endl;

    lin_approx_t u;
    make_grid(u.x, a, b, n);
    for (int j = 0; j < u.x.size(); j++) u.q.push_back(u_true(u.x[j], at));
    lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);

    fout << "size\titerations\tresidual" << endl;
    for (int size = sa; size < sb; ++size) {
        vector<double> time;
        make_grid(time, at, bt, size);

        auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
        ↪  time, 0.001, 100);

        int itersum = 0; for (auto& k : result) itersum += k.iterations;
        double residual = norm(u_truly_approx.q, result.back().answer.q);

        residual /= size; // Важно! Так как если не разделить, то расстояние между каждым узлом будет
        ↪  в итоге уменьшаться, но раз мы их делаем всё больше и больше, то в сумме оно будет
        ↪  увеличиваться, и в итоге точность будет расти.

        fout << size << "\t" << itersum << "\t" << residual << endl;
    }

    fout.close();
```

```
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------

int main() {
    init();

    writeFirstInvestigation();

    writeGridInvestigation(
        [] (double x, double t) -> double { return x*x*x*x + t; }, "$x^4+t$",
        "x4_space.txt",
        [] (double u) -> double { return u*u; }, "$u^2$"
    );

    writeGridInvestigation(
        [] (double x, double t) -> double { return exp(x) + t; }, "$exp(x)+t$",
        "expx_space.txt",
        [] (double u) -> double { return u*u; }, "$u^2$"
    );

    writeGridInvestigationTime(
        [] (double x, double t) -> double { return exp(x) + t*t*t; }, "$e^x+t^3$",
        "t3_time.txt",
        [] (double u) -> double { return u; }, "$u$"
    );

    writeGridInvestigationTime(
        [] (double x, double t) -> double { return exp(x) + exp(t); }, "$e^x+e^t$",
        "expt_time.txt",
        [] (double u) -> double { return u; }, "$u$"
    );

    writeGridInvestigationSize(
        [] (double x, double t) -> double { return exp(x) + exp(t); }, "$e^x+e^t$",
        "expx_expt_size_space.txt",
        [] (double u) -> double { return u; }, "$u$"
    );

    writeGridInvestigationSizeTime(
        [] (double x, double t) -> double { return exp(x) + exp(t); }, "$e^x+e^t$",
        "expx_expt_size_time.txt",
        [] (double u) -> double { return u; }, "$u$"
    );
}
```