

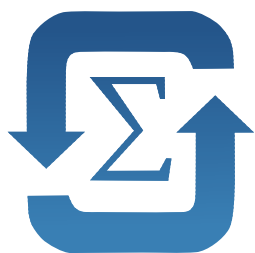
Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



Кафедра прикладной математики

Лабораторная работа №2
по дисциплине «Уравнения математической физики»

Решение нелинейных начально-краевых задач



Факультет:	ПМИ
Группа:	ПМ-63
Студент:	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 итераций 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.

Хотя, наверное, можно было бы считать общую площадь между истинной функцией и полученной аппроксимацией при помощи интеграла разности двух этих функций. Тогда появляется вопрос: как нам считать точность решения при неравномерной сетке? Очевидно, она может плохо аппроксимировать те части уравнения, на которых она сильно разрежена. Хотя, насколько сильно нас интересуют эти части? И насколько сильно точность по всей функции будет отличаться от точности по элементам сетки. Это всё интересные вопросы для потенциальных исследований, выходящие за рамки данной работы.

Далее под числом итераций будет подразумеваться сумма числа итераций по нелинейности по всем временным точкам. Например, если у нас имеется сетка по времени из 10 элементов, и на каждом шаге было совершено 10 итераций методом простой итерации, то итоговое число итераций будет равно 100.

Так же далее для всех исследований будут использоваться следующие параметры, изменение этих параметров будет оговариваться отдельно.

- $\sigma = 1$.
- $\varepsilon = 0.001$.
- $\text{iters}_{\max} = 500$.

- 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 в этот момент времени.

3.1 Точность для разных функций

Здесь показана точность решения и количество итераций в зависимости от функций $u(x, t)$ и $\lambda(u)$. $\varepsilon = 10^{-7}$.

$u(x, t) \backslash \lambda(u)$	1	u	u^2	$u^2 + 1$	u^3	u^4	e^u	$\sin u$
$3x + t$	10 0.01	47 $0.2 \cdot 10^{-6}$	101 $0.62 \cdot 10^{-8}$	96 $0.45 \cdot 10^{-8}$	123 $0.36 \cdot 10^{-7}$	152 $0.5 \cdot 10^{-7}$	187 $0.22 \cdot 10^{-6}$	3492 26
$2x^2 + t$	10 0.01	91 $0.24 \cdot 10^{-6}$	142 $0.28 \cdot 10^{-6}$	136 $0.18 \cdot 10^{-6}$	207 $0.34 \cdot 10^{-6}$	5010 20	91 $2.4 \cdot 10^5$	5010 $2.8 \cdot 10^2$
$x^3 + t$	10 $0.75 \cdot 10^{-2}$	106 $0.2 \cdot 10^{-5}$	190 $0.57 \cdot 10^{-6}$	171 $0.31 \cdot 10^{-6}$	313 $0.29 \cdot 10^{-5}$	5010 18	132 $3.8 \cdot 10^5$	5010 62
$x^4 + t$	10 0.014	135 $0.81 \cdot 10^{-6}$	264 $0.29 \cdot 10^{-5}$	231 $0.5 \cdot 10^{-5}$	3207 $0.38 \cdot 10^{-4}$	5010 29	5010 <i>nan</i>	5010 $4.5 \cdot 10^2$
$e^x + t$	10 0.01	79 $0.21 \cdot 10^{-6}$	113 $0.12 \cdot 10^{-6}$	110 $0.84 \cdot 10^{-7}$	151 $0.13 \cdot 10^{-6}$	194 $0.79 \cdot 10^{-6}$	106 $1.2 \cdot 10^5$	5010 $1.1 \cdot 10^2$
$3x + t^2$	10 0.38	51 0.06	94 0.011	90 0.01	119 $0.2 \cdot 10^{-2}$	150 $0.39 \cdot 10^{-3}$	170 $0.16 \cdot 10^{-2}$	5010 63
$3x + t^3$	10 1.4	60 0.22	94 0.04	88 0.039	117 $0.76 \cdot 10^{-2}$	146 $0.14 \cdot 10^{-2}$	165 $0.58 \cdot 10^{-2}$	5010 $2.4 \cdot 10^4$
$3x + e^t$	10 0.65	51 0.082	86 0.011	85 0.011	105 $0.15 \cdot 10^{-2}$	128 $0.22 \cdot 10^{-3}$	180 $0.5 \cdot 10^{-3}$	5010 48
$3x + \sin(t)$	10 0.17	51 0.029	93 $0.53 \cdot 10^{-2}$	91 $0.51 \cdot 10^{-2}$	117 $0.1 \cdot 10^{-2}$	147 $0.2 \cdot 10^{-3}$	170 $0.85 \cdot 10^{-3}$	5010 24
$e^x + t^2$	10 0.38	71 0.06	109 0.011	108 0.011	147 $0.21 \cdot 10^{-2}$	193 $0.44 \cdot 10^{-3}$	229 $0.19 \cdot 10^{-2}$	5010 $2.7 \cdot 10^2$
$e^x + t^3$	10 1.4	70 0.22	106 0.041	105 0.039	144 $0.8 \cdot 10^{-2}$	186 $0.16 \cdot 10^{-2}$	219 $0.69 \cdot 10^{-2}$	5010 $2.2 \cdot 10^3$
$e^x + e^t$	10 0.65	70 0.081	99 0.011	98 0.011	127 $0.16 \cdot 10^{-2}$	160 $0.23 \cdot 10^{-3}$	247 $0.59 \cdot 10^{-3}$	5010 $7.7 \cdot 10^3$
$e^x + \sin(t)$	10 0.17	73 0.028	107 $0.54 \cdot 10^{-2}$	105 $0.52 \cdot 10^{-2}$	146 $0.11 \cdot 10^{-2}$	189 $0.23 \cdot 10^{-3}$	229 $0.1 \cdot 10^{-2}$	5010 $2.5 \cdot 10^3$

Вывод: порядок аппроксимации сильно зависит от нелинейности. Например, при отсутствии нелинейности порядок аппроксимации нулевой, а при нелинейности вплоть до 3 степени, порядок аппроксимации равен ∞ , так как довольно точно аппроксимируется функция e^x . При $\lambda(u) = u^4$ порядок аппроксимации вообще равен 1.

Вывод: порядок аппроксимации по времени равен 1.

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, сгущающуюся 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 монотонной на этом участке. Тогда гарантируется условие монотонности сетки, то есть что при $j \leq i \Rightarrow x_j \leq x_i$.

Пример: при $m(x) = x$, сетка становится равномерной.
Найденные функции зависят от некоторого параметра t :

$$\begin{aligned} m_{1,t}(x) &= x^t & m_{2,t}(x) &= x^{\frac{1}{t}} \\ m_{3,t}(x) &= \frac{t^x - 1}{t - 1} & m_{4,t}(x) &= \frac{\frac{1}{t^x} - 1}{\frac{1}{t} - 1} \end{aligned}$$

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 функции 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.

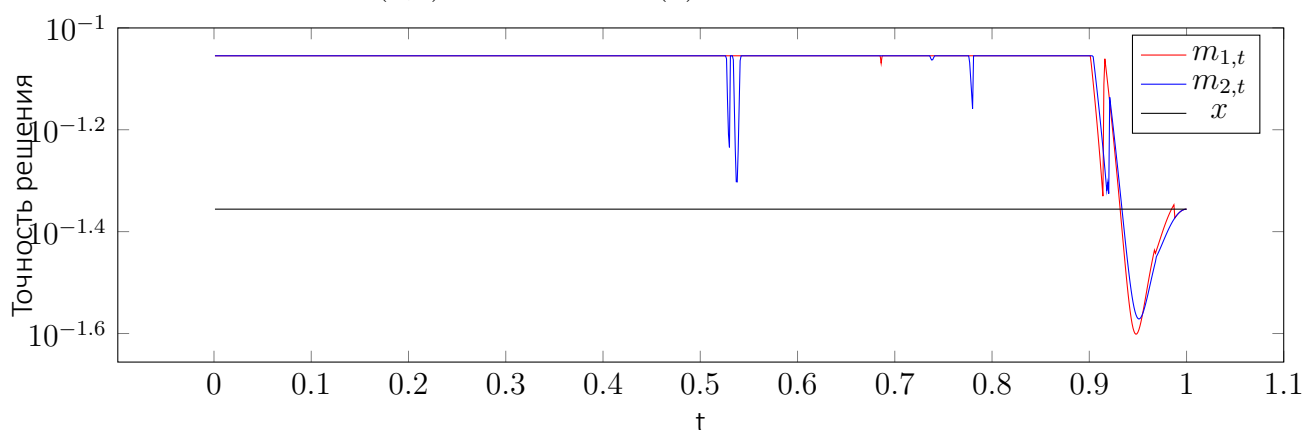
Графики точности и количества итераций обрезались сверху, так как данные о плохой точности и большом числе итераций нас не интересуют, так же эти данные мешают анализировать хорошие зависимости.

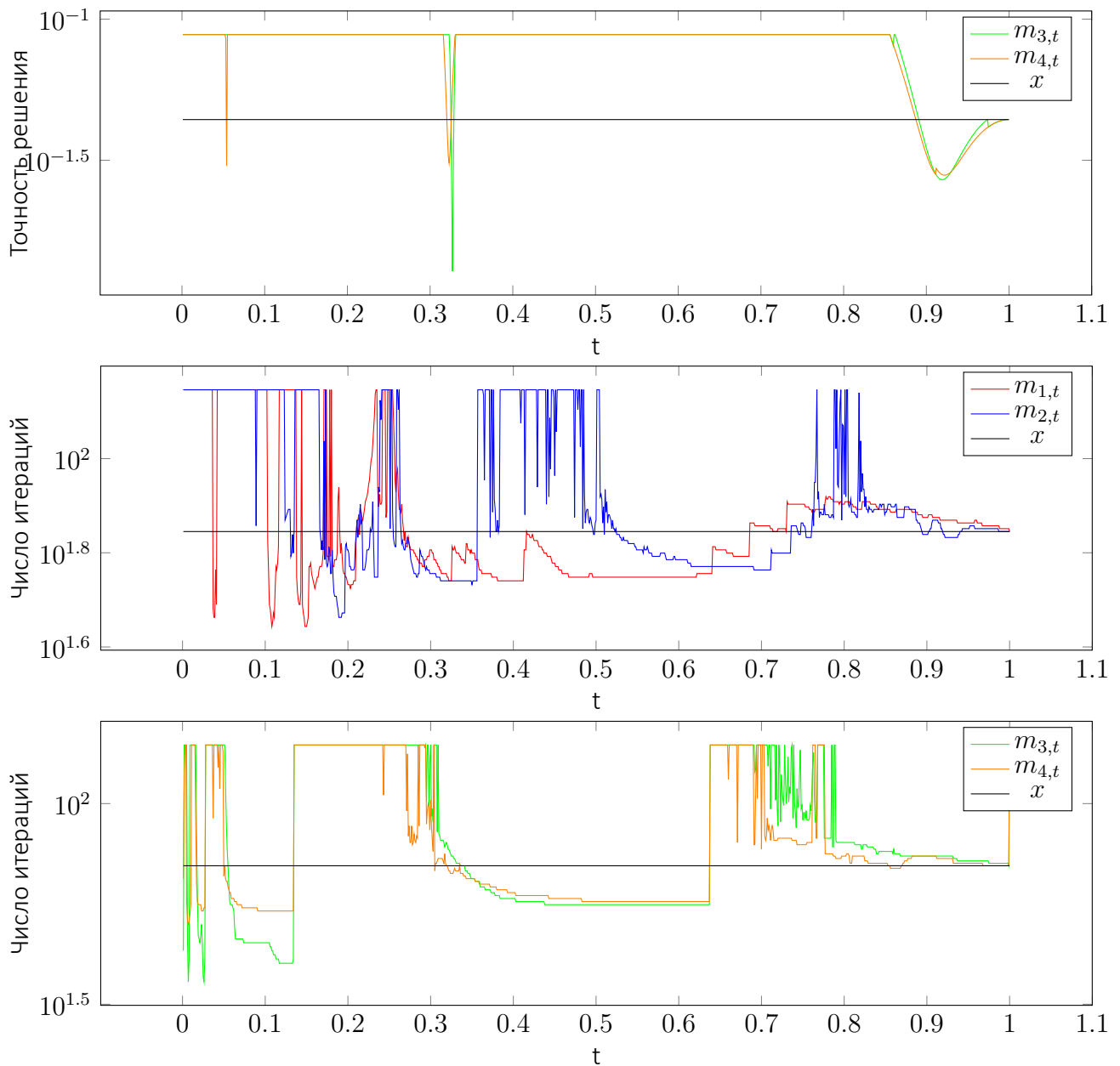
3.2.3 Сетка по пространству

В данных исследованиях неравномерность применяется к сетке по пространству.

3.2.3.1 $u = x^4 + t$

Исследуется функция $u(x, t) = x^4 + t$, при $\lambda(u) = u^2$.

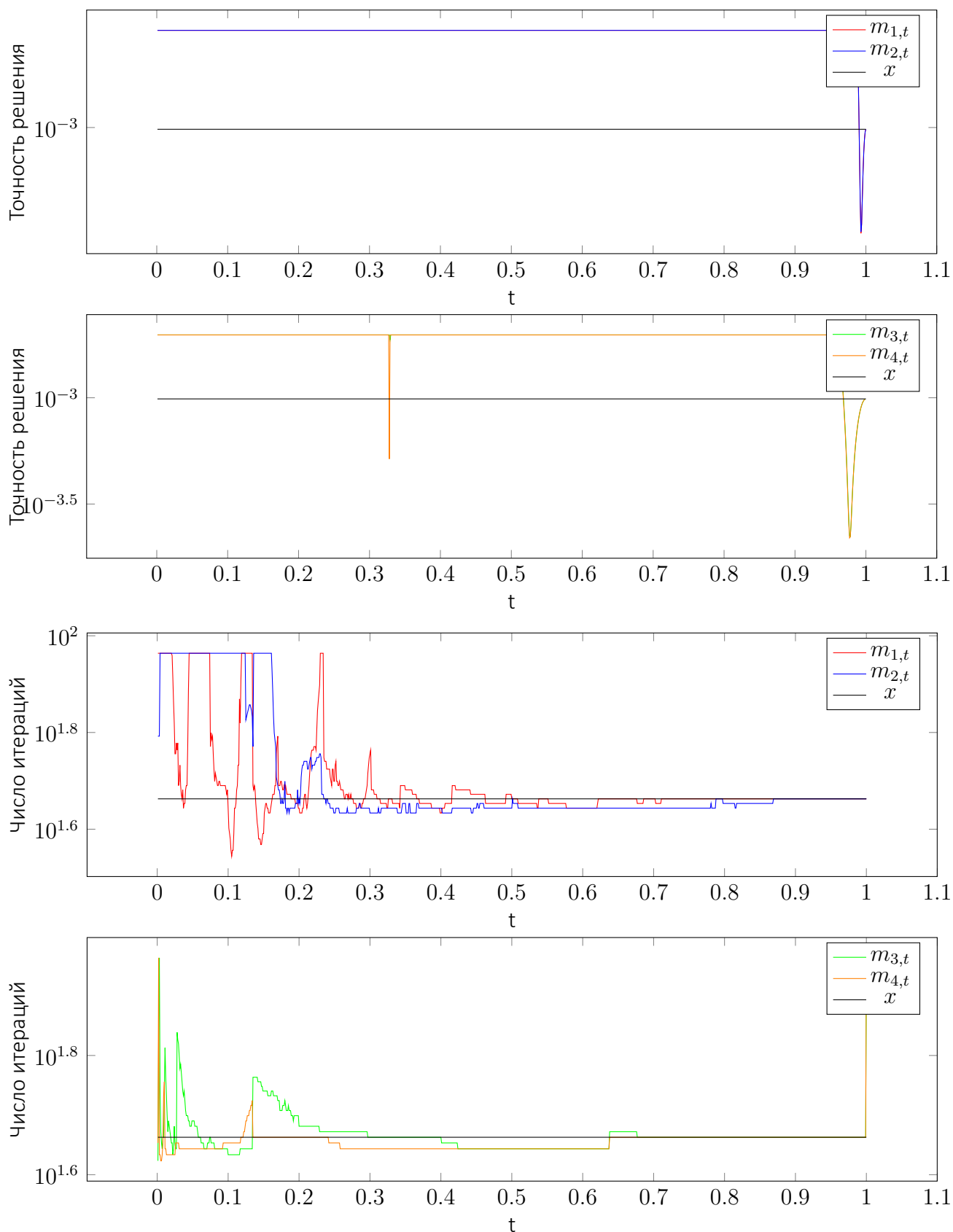




Вывод: у обеих функций имеются точки, где точность больше, чем при равномерной сетке — это точки в окрестности 0.9. Гладкость графика точности в зависимости от параметра позволяет найти эти точки, например, при помощи метода Ньютона, спускаясь из $t = 1$. Но к сожалению точность увеличивается на ничтожно малую величину, поэтому использование неравномерных сеток на основе данных функций может быть нецелесообразно. Так же в этих точках немного увеличивается число итераций.

3.2.3.2 $u = \exp(x) + t$

Исследуется функция $u(x, t) = e^x + t$, при $\lambda(u) = u^2$.



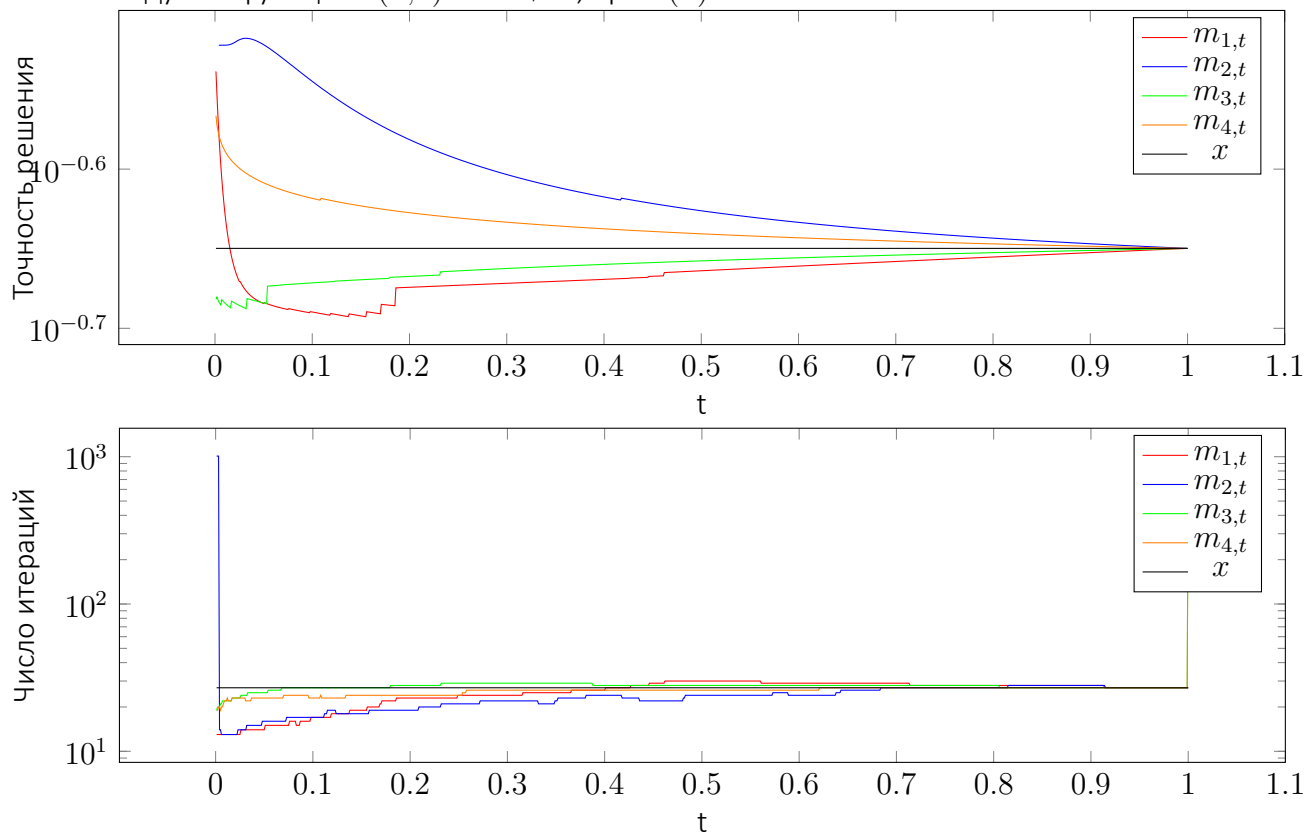
Вывод: предыдущий вывод подтверждается ещё одним примером, разве что теперь точность увеличилась не на ничтожную величину, а на половину порядка.

3.2.4 Сетка по времени

В данных исследованиях неравномерность применяется к сетке по времени.

3.2.4.1 $u = \exp(x) + t^3$

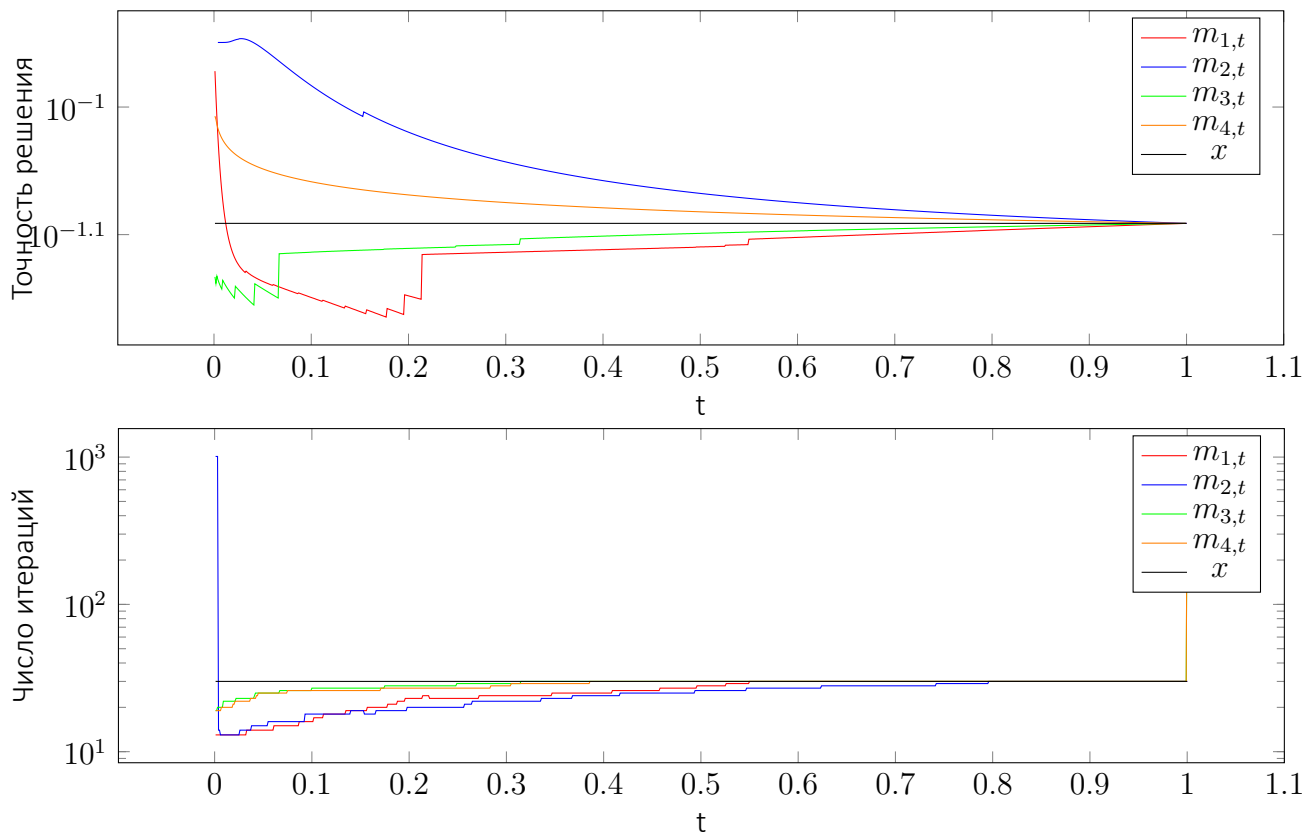
Исследуется функция $u(x, t) = e^x + t^3$, при $\lambda(u) = u$.



Вывод: точность лучше, чем у равномерной сетки, показали 1 и 3 функция, которые сгущаются к последнему моменту времени $t = 1$. Наверное благодаря этому точность и увеличивается. Причем видно, что как и у 1, так и у 3 функции есть явно выраженные минимумы, но к сожалению функция зависимости точности от параметра не идеально гладкая, а с небольшими ступеньками, что не позволяет настолько просто подобрать нужный параметр при помощи градиентных методов. Хотя, к сожалению, точность увеличивается на совершенно ничтожную величину. Так же любая неравномерная сетка требует меньше итераций, чем равномерная, что может быть аргументом в пользу их использования.

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

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



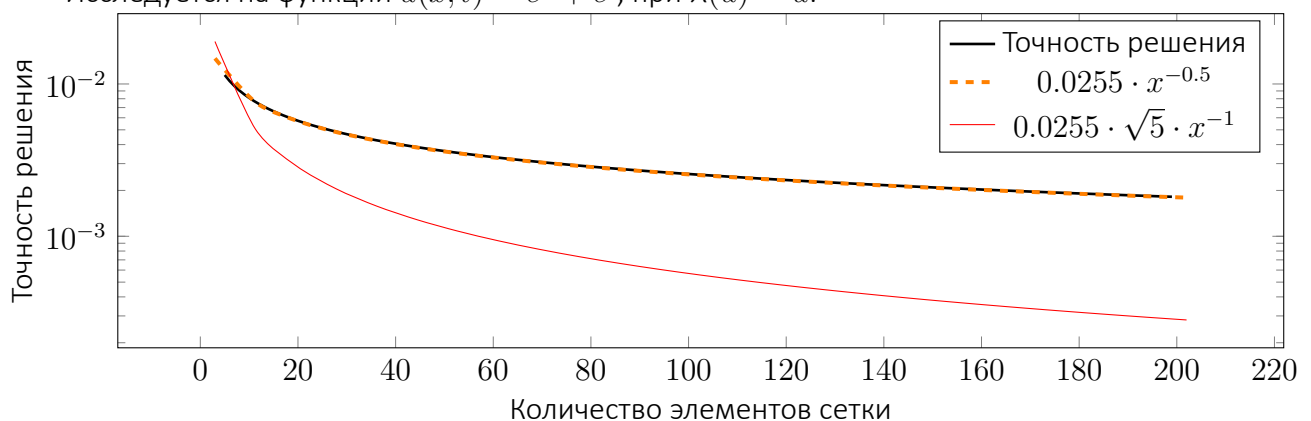
Вывод: предыдущий вывод подтвердился. Так же можно заметить, что при некоторых параметрах сетки число итераций становится вообще минимально возможным — 10.

3.3 Точность в зависимости от размера сетки

В данном пункте определяется *порядок сходимости*. **Порядок сходимости** лучше всего описать на примере. Например, у нас есть сетка, и точность на ней равна 10, мы уменьшаем эту сетку вдвое, и точность становится 5. Тогда порядок сходимости равен 1, если же итоговая точность станет 2.5, то порядок сходимости будет 2. Порядок сходимости — это степень того, насколько сильно увеличивается точность при уменьшении сетки на какой-то параметр. Порядок сходимости можно определить из степени, которая стоит у x , в функции, хорошо аппроксимирующей исходные данные.

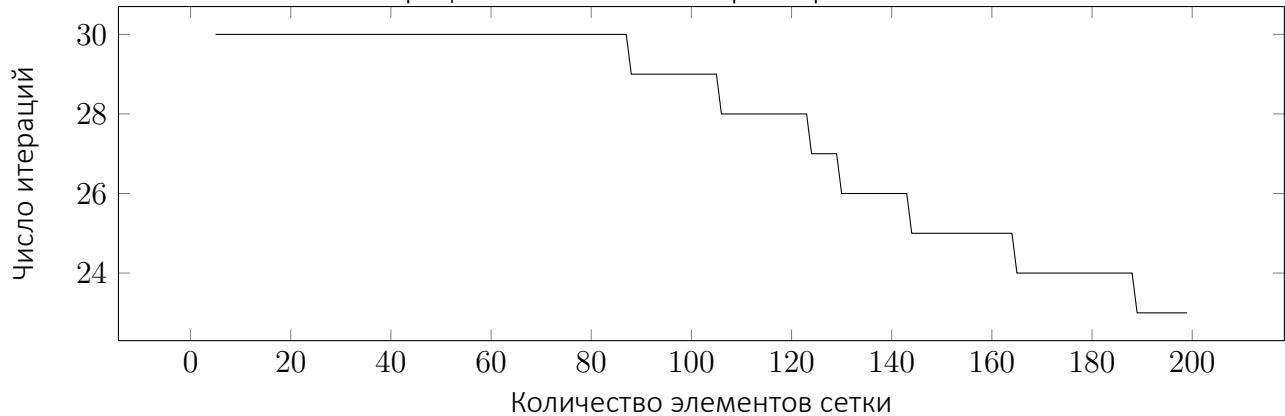
3.3.1 Сетка по пространству

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



Вывод: порядок сходимости по пространственной сетке равен 0.5.

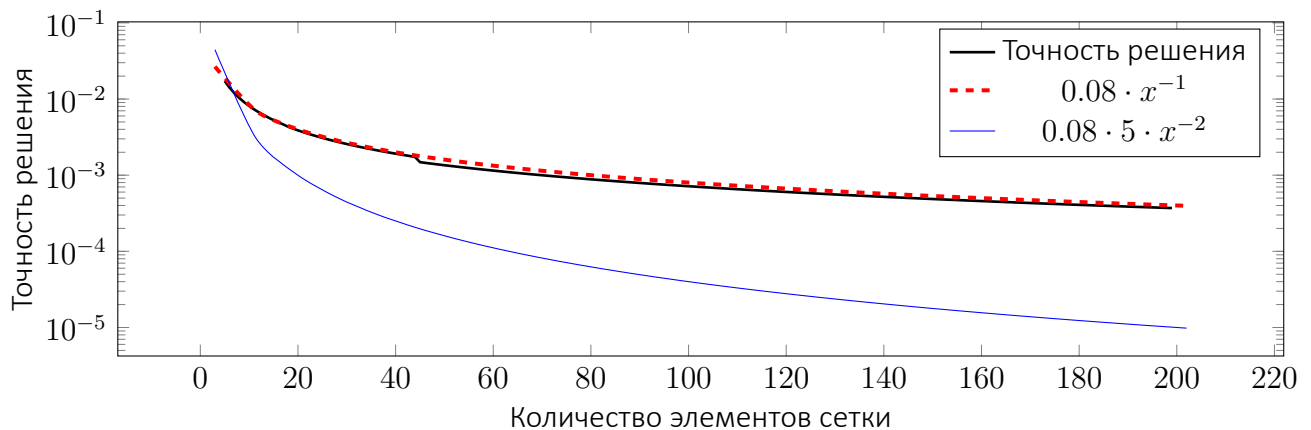
Так же имеется число итераций в зависимости от размера сетки:



Вывод: при увеличении числа элементов пространственной сетки, число итераций падает, хоть и незначительно.

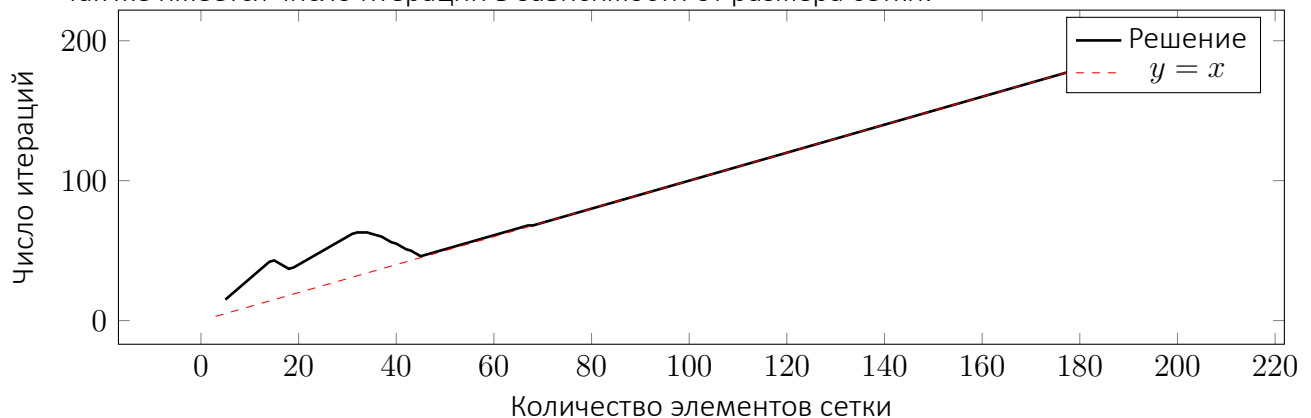
3.3.2 Сетка по времени

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



Вывод: порядок сходимости по временной сетке равен 1.

Так же имеется число итераций в зависимости от размера сетки:



Вывод: начиная с определенного числа элементов временной сетки, число итераций равно числу элементов временной сетки. Что означает, что в методе простой итерации происходит минимальное возможное число итераций — 1.

Интересно было бы сравнить данные результаты с результатами, когда использовалась интегральная норма, а не норма между векторами. Может оказаться, что увеличение размера сетки по пространству может оказаться выгодней увеличения сетки по времени, несмотря на то, что порядок сходимости у временной сетки лучше.

4 Код программы

FILE main.cpp

```
1 #include <iostream>
2 #include <algorithm>
3 #include <functional>
4 #include <iterator>
5 #include <iomanip>
6 #include <fstream>
7 #include <sstream>
8 #include <Eigen/Dense>
9
10 using namespace std;
11
12 typedef Eigen::MatrixXd Matrix;
13 typedef Eigen::VectorXd Vector;
14
15 typedef function<double(double)> Function1D;
16 typedef function<double(double, double)> Function2D;
17 typedef function<double(double, double, double)> Function3D;
18
19 //-----
20 double integral_gauss3(const vector<double>& X, Function1D f) {
21     const double x1 = -sqrt(3.0/5.0);
22     const double x2 = 0;
23     const double x3 = -x1;
24     const double q1 = 5.0/9.0;
25     const double q2 = 8.0/9.0;
26     const double q3 = q1;
27     double sum = 0;
28     double xk = 0;
29     double h = X[1] - X[0];
30     double h2 = h/2.0;
31
32     for (int i = 0; i < X.size()-1; ++i) {
33         xk = (X[i]+X[i+1])/2.0;
34         sum += q1 * f(xk + x1 * h2);
35         sum += q2 * f(xk + x2 * h2);
36         sum += q3 * f(xk + x3 * h2);
37     }
38
39     sum *= h;
40     sum /= 2.0;
41     return sum;
42 }
43
44 //-----
45 void make_grid(vector<double>& X, double a, double b, long n, Function1D move = [] (double x) ->
46     ↪ double {return x;}) {
47     X.clear();
48     double size = b-a;
49     for (double i = 0; i <= n; ++i)
50         X.push_back(a + move(i/double(n)) * size);
51 }
52
53 //-----
54 //-----
55 //-----
56
57 double basicFunction(double left, double middle, double right, double x) {
58     // Базовая линейная функция
59     if (x > left && x < right) {
60         if (x < middle) {
61             return (x-left)/(middle-left);
62         } else {
63             return 1-(x-middle)/(right-middle);
64         }
65     }
66
67     return 0;
68 }
69
70 //-----
71 double basicFunctionGrad(double left, double middle, double right, double x) {
72     // Базовая линейная функция
73     if (x > left && x < right) {
74         if (x < middle) {
75             return 1.0/(middle-left);
76         } else {
77             return -1.0/(right-middle);
78         }
79     }
80 }
```

```

81     return 0;
82 }
83
84 //-----
85 Function1D calcFirstDerivative(const Function1D& f) {
86     return [f](double x) -> double {
87         const double h = 0.001;
88         return (-f(x + 2 * h) + 8 * f(x + h) - 8 * f(x - h) + f(x - 2 * h)) / (12 * h);
89     };
90 }
91
92 //-----
93 Function2D calcRightPart(
94     const Function1D& lambda,
95     const Function2D& u,
96     const Function1D& sigma
97 ) {
98     // f = -div(lambda(u) * grad u) + sigma * du/dt
99     return [=](double x, double t) -> double {
100         using namespace placeholders;
101         auto ut = calcFirstDerivative(bind(u, x, _1));
102         auto ux = calcFirstDerivative(bind(u, _1, t));
103         auto lambda_grad = [lambda, ux, u](double x, double t) -> double {
104             return lambda(u(x, t)) * ux(x);
105         };
106         auto div = calcFirstDerivative(bind(lambda_grad, _1, t));
107         return -div(x) + sigma(x) * ut(x);
108     };
109 }
110
111 //-----
112 double norm(const vector<double>& a, const vector<double>& b) {
113     assert(a.size() == b.size());
114     double sum = 0;
115     for (int i = 0; i < a.size(); i++)
116         sum += (a[i] - b[i]) * (a[i] - b[i]);
117     return sqrt(sum);
118 }
119
120 //-----
121 ostream& operator<<(ostream& out, const vector<double>& mas) {
122     for (auto& i : mas)
123         out << i << " ";
124     return out;
125 }
126
127 //-----
128 string write_for_latex_double(double v, int precision) {
129     int power = log(std::fabs(v)) / log(10.0);
130     double value = v / pow(10.0, power);
131
132     if (v != v) return "nan";
133
134     if (v == 0) {
135         power = 0;
136         value = 0;
137     }
138
139     stringstream sout;
140     sout.precision(precision);
141     if (power == -1 || power == 0 || power == 1) {
142         sout << v;
143     } else {
144         sout << value << "\\cdot 10^{\"" << power << "\"}";
145     }
146
147     return sout.str();
148 }
149
150 //-----
151 //-----
152 //-----
153
154 //-----
155 class lin_approx_t
156 {
157 public:
158     vector<double> q; /// Вектор весов
159     vector<double> x; /// Массив положений каждого элемента
160
161     int size(void) const {
162         return x.size();
163     }
164
165     double left(int i) const {
166         if (i == 0)
167             return x[0] - 0.00001;
168         else
169             return x[i - 1];
170     }
171 }

```

```

170 }
171
172 double middle(int i) const {
173     return x[i];
174 }
175
176 double right(int i) const {
177     if (i == x.size() - 1)
178         return x.back()+0.00001;
179     else
180         return x[i+1];
181 }
182
183 double value(double pos) const {
184     int start = distance(x.begin(), lower_bound(x.begin(), x.end(), pos))-1;
185     if (start == -1) {
186         if (fabs(pos-x[0]) < 0.00001)
187             return q[0];
188         else
189             return 0;
190     }
191     if (start == x.size()) {
192         if (fabs(pos-x.back()) < 0.00001)
193             return q.back();
194         else
195             return 0;
196     }
197     double sum = 0;
198     for (int i = start; i < min<int>(start+2, x.size()); ++i)
199         //for (int i = 0; i < x.size(); ++i)
200             sum += q[i] * basic(pos, i);
201     return sum;
202 } /// Получить значение функции аппроксимации в некоторой точке
203
204 double basic(double pos, int i) const {
205     return basicFunction(left(i), middle(i), right(i), pos);
206 } /// Получить значение базовой функции под номером i в точке pos
207
208 double basic_grad(double pos, int i) const {
209     return basicFunctionGrad(left(i), middle(i), right(i), pos);
210 } /// Получить значение производной базовой функции под номером i в точке pos
211 };
212
213 //-----
214 //-----
215 //-----
216
217 //-----
218 double calc_a1_integral(const Function1D& lambda, const lin_approx_t& u, int i, int j) {
219     if (abs(i - j) > 1)
220         return 0;
221     auto f = [&] (double x) -> double {
222         return lambda(u.value(x)) * u.basic_grad(x, i) * u.basic_grad(x, j);
223     };
224     vector<double> X;
225     if (i != j)
226         make_grid(X, min(u.middle(i), u.middle(j)), min(u.right(i), u.right(j)), 10);
227     else
228         make_grid(X, u.left(i), u.right(i), 10);
229     return integral_gauss3(X, f);
230 }
231
232 //-----
233 double calc_a2_integral(const Function1D& sigma, const lin_approx_t& u, int i, int j) {
234     if (abs(i - j) > 1)
235         return 0;
236     auto f = [&] (double x) -> double {
237         return sigma(x) * u.basic(x, i) * u.basic(x, j);
238     };
239     vector<double> X;
240     if (i != j)
241         make_grid(X, min(u.middle(i), u.middle(j)), min(u.right(i), u.right(j)), 10);
242     else
243         make_grid(X, u.left(i), u.right(i), 10);
244     return integral_gauss3(X, f);
245 }
246
247 //-----
248 double calc_b1_integral(const Function1D& f, const lin_approx_t& u, int i) {
249     auto fun = [&] (double x) -> double {
250         return f(x) * u.basic(x, i);
251     };
252     vector<double> X;
253     make_grid(X, u.left(i), u.right(i), 10);
254     return integral_gauss3(X, fun);
255 }
256
257 //-----

```

```

258 double calc_b2_integral(const Function1D& sigma, const lin_approx_t& u, const lin_approx_t&
↪ u_last_time, int i) {
259     auto fun = [&] (double x) -> double {
260         return sigma(x) * u_last_time.value(x) * u.basic(x, i);
261     };
262     vector<double> X;
263     make_grid(X, u.left(i), u.right(i), 10);
264     return integral_gauss3(X, fun);
265 }
266
267 //-----
268 lin_approx_t approximate_function(Function1D lambda, const lin_approx_t& u) {
269     lin_approx_t result = u;
270     for (int i = 0; i < result.q.size(); ++i)
271         result.q[i] = lambda(u.x[i]);
272     return result;
273 }
274
275 //-----
276 Matrix calcA(const lin_approx_t& u_last, double dt, Function1D lambda, Function1D sigma) {
277     Matrix result(u_last.size(), u_last.size());
278     for (int i = 0; i < u_last.size()-1; ++i) {
279         for (int j = 0; j < u_last.size(); ++j) {
280             result(i, j) = calc_a1_integral(lambda, u_last, i, j) + calc_a2_integral(sigma, u_last, i,
↪ j) / dt;
281         }
282     }
283     return result;
284 } /// Рассчитать матрицу A(q)
285
286 //-----
287 Vector calcB(const lin_approx_t& u_last, double dt, Function1D f, Function1D sigma, const
↪ lin_approx_t& u_last_time) {
288     Vector result(u_last.size());
289     for (int i = 0; i < u_last.size(); ++i) {
290         result(i) = calc_b1_integral(f, u_last, i) + calc_b2_integral(sigma, u_last, u_last_time, i) /
↪ dt;
291     }
292     return result;
293 } /// Рассчитать вектор правой части b(q)
294
295 //-----
296 void write_first_boundary_conditions(Matrix& A, Vector& b, const lin_approx_t& u, Function2D u_true,
↪ double time) {
297     A(0, 0) = 1;
298     for (int i = 1; i < A.cols(); ++i) A(0, i) = 0;
299     b(0) = u_true(u.x.front(), time);
300
301     A(A.rows()-1, A.cols()-1) = 1;
302     for (int i = 0; i < A.cols() - 1; ++i) A(A.rows()-1, i) = 0;
303     b(b.rows()-1) = u_true(u.x.back(), time);
304 }
305
306 //-----
307 //-----
308 //-----
309
310 //-----
311
312 struct Result
313 {
314     enum ExitType {
315         EXIT_RESIDUAL,
316         EXIT_STEP,
317         EXIT_ITERATIONS,
318         EXIT_ERROR
319     } exitType;
320
321     lin_approx_t answer;
322     int iterations;
323     double residual;
324 };
325
326 //-----
327 ostream& operator<<(ostream& out, const Result::ExitType& exit) {
328     switch (exit) {
329         case Result::EXIT_RESIDUAL: out << "residual"; break;
330         case Result::EXIT_STEP: out << "step"; break;
331         case Result::EXIT_ITERATIONS: out << "iterations"; break;
332         case Result::EXIT_ERROR: out << "error"; break;
333     }
334     return out;
335 }
336
337 //-----
338 Result solveFixedPointIteration(
339     Function2D f,
340

```

```

341 Function2D u_true,
342 Function1D lambda,
343 Function1D sigma,
344 const lin_approx_t& u_last_time,
345 double dt,
346 double time,
347 double eps,
348 int maxiter
349 ) {
350     lin_approx_t u = u_last_time;
351
352     Vector last_x(u.q.size());
353     for (int i = 0; i < u.q.size(); i++)
354         last_x[i] = u.q[i];
355
356     Result result;
357     result.iterations = 0;
358     while (true) {
359         auto A = calcA(u, dt, lambda, sigma);
360         auto b = calcB(u, dt, bind(f, placeholders::_1, time), sigma, u_last_time);
361         write_first_boundary_conditions(A, b, u, u_true, time);
362
363         Eigen::JacobiSVD<Matrix> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);
364         Vector x = svd.solve(b);
365         for (int i = 0; i < u.size(); ++i)
366             u.q[i] = x[i];
367
368         result.iterations++;
369
370         if (result.iterations > maxiter) {
371             result.exitType = Result::EXIT_ITERATIONS;
372             break;
373         }
374
375         {
376             auto A1 = calcA(u, dt, lambda, sigma);
377             auto b1 = calcB(u, dt, bind(f, placeholders::_1, time), sigma, u_last_time);
378             write_first_boundary_conditions(A1, b1, u, u_true, time);
379
380             double au = (A1 * x - b1).norm();
381             double bu = b1.norm();
382             result.residual = au/bu;
383             if (au/bu < eps) {
384                 result.exitType = Result::EXIT_RESIDUAL;
385                 break;
386             }
387         }
388
389         if ((x-last_x).norm() < eps) {
390             result.exitType = Result::EXIT_STEP;
391             break;
392         }
393
394         last_x = x;
395     }
396
397     result.answer = u;
398
399     return result;
400 } /// Решить уравнение методом простой итерации
401
402 lin_approx_t calcTrulyApprox(const vector<double>& grid, const Function2D& u_true, double time) {
403     lin_approx_t u_truly_approx;
404     u_truly_approx.x = grid;
405     u_truly_approx.q = grid;
406     for (int i = 0; i < u_truly_approx.x.size(); i++)
407         u_truly_approx.q[i] = u_true(u_truly_approx.x[i], time);
408     return u_truly_approx;
409 }
410
411 //-----
412 void write_iterations_information(double t0, const Result& result, const Function2D& u_true) {
413     cout << "time: " << t0 << endl;
414     cout << "residual: " << result.residual << endl;
415     cout << "iterations: " << result.iterations << endl;
416     cout << "answer: " << result.answer.q << endl;
417
418     lin_approx_t u_truly_approx = calcTrulyApprox(result.answer.x, u_true, t0);
419     cout << "shold be: " << u_truly_approx.q << endl;
420
421     cout << "norm: " << norm(u_truly_approx.q, result.answer.q) << endl;
422
423     cout << endl << "-----" << endl;
424 }
425
426 //-----
427 vector<Result> solveByTime(
428     Function2D f,
429     Function2D u_true,

```

```

430     Function1D lambda,
431     Function1D sigma,
432     const lin_approx_t& u_start,
433     const vector<double>& time_grid,
434     double eps,
435     double maxiter
436 ) {
437     vector<Result> res;
438     lin_approx_t u = u_start;
439
440     for (int i = 1; i < time_grid.size(); ++i) {
441         double t0 = time_grid[i];
442         double t1 = time_grid[i-1];
443         auto result = solveFixedPointIteration(f, u_true, lambda, sigma, u, t0-t1, t0, eps, maxiter);
444         res.push_back(result);
445
446         // Выводим мета-информацию
447         //write_observations_information(t0, result, u_true);
448
449         u = result.answer;
450     }
451
452     return res;
453 }
454
455 //-----
456 //-----
457 //-----
458
459 //-----
460 template<class A, class B, class C>
461 struct triple {
462     A first;
463     B second;
464     C third;
465 };
466
467 //-----
468 vector<triple<Function2D, string, bool>> us;
469 vector<pair<Function1D, string>> lambdas;
470 vector<Function2D> moves;
471 auto sigma = [] (double x) -> double { return 1; };
472
473 double a = 1, b = 2, n = 10; // Характеристики сетки по пространству
474 double at = 0, bt = 1, nt = 10; // Характеристики сетки по времени
475 double na = 0, nb = 1, nn = 1000; // Характеристики сетки по параметру t, в зависимости от которого
476   ↪   меняются неравномерные сетки
477 int sa = 5, sb = 200; // Характеристики сетки по размеру
478
479 //-----
480 void init() {
481     us.push_back({[] (double x, double t) -> double { return 3*x + t; }, "$3x+t$", false});
482     us.push_back({[] (double x, double t) -> double { return 2*x*x + t; }, "$2x^2+t$", false});
483     us.push_back({[] (double x, double t) -> double { return x*x*x + t; }, "$x^3+t$", false});
484     us.push_back({[] (double x, double t) -> double { return x*x*x*x + t; }, "$x^4+t$", false});
485     us.push_back({[] (double x, double t) -> double { return exp(x) + t; }, "$e^x+t$", false});
486     us.push_back({[] (double x, double t) -> double { return 3*x + t*t; }, "$3x+t^2$", true});
487     us.push_back({[] (double x, double t) -> double { return 3*x + t*t*t; }, "$3x+t^3$", true});
488     us.push_back({[] (double x, double t) -> double { return 3*x + exp(t); }, "$3x+e^t$", true});
489     us.push_back({[] (double x, double t) -> double { return 3*x + sin(t); }, "$3x+sin(t)$", true});
490     us.push_back({[] (double x, double t) -> double { return exp(x) + t*t; }, "$e^x+t^2$", true});
491     us.push_back({[] (double x, double t) -> double { return exp(x) + t*t*t; }, "$e^x+t^3$", true});
492     us.push_back({[] (double x, double t) -> double { return exp(x) + exp(t); }, "$e^x+e^t$", true});
493     us.push_back({[] (double x, double t) -> double { return exp(x) + sin(t); }, "$e^x+sin(t)$", true});
494
495     lambdas.push_back({[] (double u) -> double { return 1; }, "$1$" });
496     lambdas.push_back({[] (double u) -> double { return u; }, "$u$" });
497     lambdas.push_back({[] (double u) -> double { return u*u; }, "$u^2$" });
498     lambdas.push_back({[] (double u) -> double { return u*u + 1; }, "$u^2+1$" });
499     lambdas.push_back({[] (double u) -> double { return u*u*u; }, "$u^3$" });
500     lambdas.push_back({[] (double u) -> double { return u*u*u*u; }, "$u^4$" });
501     lambdas.push_back({[] (double u) -> double { return exp(u); }, "$e^u$" });
502     lambdas.push_back({[] (double u) -> double { return sin(u); }, "$sinu$" });
503
504     moves.push_back({[] (double x, double t) -> double { return pow(x, t); }});
505     moves.push_back({[] (double x, double t) -> double { return pow(x, 1.0/t); }});
506     moves.push_back({[] (double x, double t) -> double { return (pow(t, x)-1.0)/(t-1.0); }});
507     moves.push_back({[] (double x, double t) -> double { return (pow(1.0/t, x)-1.0)/(1.0/t-1.0); }});
508
509 //-----
510 void writeFirstInvestigation() {
511     ofstream fout("first.txt");
512     fout << "a\t";
513     for (auto& i : lambdas) fout << i.second << ((i.second != lambdas.back().second) ? "\t" : "");
514     fout << endl;
515

```

```

516 vector<double> time;
517 make_grid(time, at, bt, nt);
518 for (auto& i : us) {
519     fout << i.second << "\t";
520     auto u_true = i.first;
521     auto sigma = [] (double x) -> double { return 1; };
522
523     lin_approx_t u;
524     make_grid(u.x, a, b, n);
525     for (int j = 0; j < u.x.size(); j++)
526         if (i.third)
527             u.q.push_back(u_true(u.x[j], at));
528         else
529             u.q.push_back(1);
530
531     lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);
532
533     for (auto& j : lambdas) {
534         auto lambda = j.first;
535
536         auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
537             ↪ time, 0.001, 500);
538
539         int itersum = 0;
540         for (auto& k : result) itersum += k.iterations;
541
542         double residual = norm(u_truly_approx.q, result.back().answer.q);
543
544         fout << "\scalebox{.55}{\tcell{$} << itersum << "$\\$\" <<
545             ↪ write_for_latex_double(residual, 2) << "$}" << ((j.second != lambdas.back().second) ?
546             ↪ "\t" : "");
547     }
548     fout << endl;
549 }
550 fout.close();
551 }
552
553 //-----
554 void writeGridInvestigation(
555     const Function2D& u_true, string su,
556     const string& file,
557     const Function1D& lambda, string slambda) {
558     // Делаем равномерную сетку по времени
559     vector<double> time;
560     make_grid(time, at, bt, nt);
561
562     pair<int, double> resu = {-1, 0};
563
564     auto test_grid = [&resu, time, u_true, lambda] (const vector<double>& grid) -> pair<int, double> {
565         lin_approx_t u; u.x = grid;
566         for (int j = 0; j < u.x.size(); j++) u.q.push_back(1);
567         lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);
568
569         auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
570             ↪ time, 0.001, 100);
571
572         int itersum = 0; for (auto& k : result) itersum += k.iterations;
573         double residual = norm(u_truly_approx.q, result.back().answer.q);
574         if (residual > resu.second * 2 && resu.first != -1) residual = resu.second * 2;
575         if (itersum > resu.first * 2 && resu.first != -1) itersum = resu.first * 2;
576         return {itersum, residual};
577     };
578
579     ofstream fout(file);
580     fout << "u = " << su << ", lambda = " << slambda << endl;
581
582     // Получаем результаты для равномерной сетки
583     vector<double> x_uniform;
584     make_grid(x_uniform, a, b, n);
585     resu = test_grid(x_uniform);
586
587     fout << "t\tru\tiu";
588     int counter = 0;
589     for (auto& i : moves) {
590         counter++;
591         fout << "\tr" << counter << "\ti" << counter;
592     }
593     fout << endl;
594     for (int i = 1; i <= nn; i++) {
595         double t = na + (nb-na) * i/nn;
596         cout << "\r" << 100 * t << " ";
597         fout << t << "\t" << resu.second << "\t" << resu.first;
598     }
599
600     vector<double> grid;
601     for (auto& move : moves) {
602         make_grid(grid, a, b, n, bind(move, placeholders::_1, t));
603         auto res = test_grid(grid);

```



```

600     fout << "\t" << res.second << "\t" << res.first;
601 }
602
603     fout << endl;
604 }
605     cout << endl;
606     fout.close();
607 }
608 //-----
609 void writeGridInvestigationTime(
610     const Function2D& u_true, string su,
611     const string& file,
612     const Function1D& lambda, string slambda) {
613     pair<int, double> resu = {-1, 0};
614
615     auto test_grid = [u_true, lambda, resu] (const vector<double>& grid) -> pair<int, double> {
616         lin_approx_t u; make_grid(u.x, a, b, n);
617         for (int j = 0; j < u.x.size(); j++) u.q.push_back(u_true(u.x[j], at));
618         lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);
619
620         auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
621             ↪ grid, 0.001, 100);
622
623         int itersum = 0; for (auto& k : result) itersum += k.iterations;
624         double residual = norm(u_truly_approx.q, result.back().answer.q);
625         if (residual > resu.second * 2 && resu.first != -1) residual = resu.second * 2;
626         if (itersum > resu.first * 2 && resu.first != -1) itersum = resu.first * 2;
627         return {itersum, residual};
628     };
629
630     ofstream fout(file);
631     fout << "u = " << su << ", lambda = " << slambda << endl;
632
633     // Получаем результаты для равномерной сетки
634     vector<double> uniform;
635     make_grid(uniform, at, bt, nt);
636     resu = test_grid(uniform);
637
638     fout << "t\tru\tiu";
639     int counter = 0;
640     for (auto& i : moves) {
641         counter++;
642         fout << "\tr" << counter << "\ti" << counter;
643     }
644     fout << endl;
645     for (int i = 1; i <= nn; i++) {
646         double t = na + (nb-na) * i/nn;
647         cout << "\r" << 100 * t << " ";
648         fout << t << "\t" << resu.second << "\t" << resu.first;
649
650         vector<double> grid;
651         for (auto& move : moves) {
652             make_grid(grid, at, bt, nt, bind(move, placeholders::_1, t));
653             auto res = test_grid(grid);
654             fout << "\t" << res.second << "\t" << res.first;
655         }
656
657         fout << endl;
658     }
659
660     cout << endl;
661     fout.close();
662 }
663 //-----
664 void writeGridInvestigationSize(
665     const Function2D& u_true, string su,
666     const string& file,
667     const Function1D& lambda, string slambda) {
668     ofstream fout(file);
669     fout << "u = " << su << ", lambda = " << slambda << endl;
670
671     // Делаем равномерную сетку по времени
672     vector<double> time;
673     make_grid(time, at, bt, nt);
674
675     fout << "size\titerations\tresidual" << endl;
676     for (int size = sa; size < sb; ++size) {
677         lin_approx_t u;
678         make_grid(u.x, a, b, size);
679         for (int j = 0; j < u.x.size(); j++) u.q.push_back(u_true(u.x[j], at));
680         lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);
681
682         auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
683             ↪ time, 0.001, 100);
684     }

```

```

686     int itersum = 0; for (auto& k : result) itersum += k.iterations;
687     double residual = norm(u_truly_approx.q, result.back().answer.q);
688
689     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.
690
691     fout << size << "\t" << itersum << "\t" << residual << endl;
692 }
693
694 fout.close();
695 }
696
697 //-----
698 void writeGridInvestigationSizeTime(
699     const Function2D& u_true, string su,
700     const string& file,
701     const Function1D& lambda, string slambda) {
702     ofstream fout(file);
703     fout << "u = " << su << ", lambda = " << slambda << endl;
704
705     lin_approx_t u;
706     make_grid(u.x, a, b, n);
707     for (int j = 0; j < u.x.size(); j++) u.q.push_back(u_true(u.x[j], at));
708     lin_approx_t u_truly_approx = calcTrulyApprox(u.x, u_true, bt);
709
710     fout << "size\titerations\tresidual" << endl;
711     for (int size = sa; size < sb; ++size) {
712         vector<double> time;
713         make_grid(time, at, bt, size);
714
715         auto result = solveByTime(calcRightPart(lambda, u_true, sigma), u_true, lambda, sigma, u,
        ↳ time, 0.001, 100);
716
717         int itersum = 0; for (auto& k : result) itersum += k.iterations;
718         double residual = norm(u_truly_approx.q, result.back().answer.q);
719
720         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.
721
722         fout << size << "\t" << itersum << "\t" << residual << endl;
723     }
724
725     fout.close();
726 }
727
728 //-----
729 //-----
730 //-----
731
732 int main() {
733     init();
734
735     writeFirstInvestigation();
736
737     writeGridInvestigation(
738         [] (double x, double t) -> double { return x*x*x*x + t; }, "$x^4+t$",
739         "x4_space.txt",
740         [] (double u) -> double { return u*u; }, "$u^2$"
741     );
742
743     writeGridInvestigation(
744         [] (double x, double t) -> double { return exp(x) + t; }, "$exp(x)+t$",
745         "expx_space.txt",
746         [] (double u) -> double { return u*u; }, "$u^2$"
747     );
748
749     writeGridInvestigationTime(
750         [] (double x, double t) -> double { return exp(x) + t*t*t; }, "$e^x+t^3$",
751         "t3_time.txt",
752         [] (double u) -> double { return u; }, "$u$"
753     );
754
755     writeGridInvestigationTime(
756         [] (double x, double t) -> double { return exp(x) + exp(t); }, "$e^x+e^t$",
757         "expt_time.txt",
758         [] (double u) -> double { return u; }, "$u$"
759     );
760
761     writeGridInvestigationSize(
762         [] (double x, double t) -> double { return exp(x) + exp(t); }, "$e^x+e^t$",
763         "expx_expt_size_space.txt",
764         [] (double u) -> double { return u; }, "$u$"
765     );
766
767     writeGridInvestigationSizeTime(
768         [] (double x, double t) -> double { return exp(x) + exp(t); }, "$e^x+e^t$",

```

```
769     "exp_xpt_size_time.txt",  
770     [] (double u) -> double { return u; }, "$u$"  
771 );  
772 }
```